No

Date

マルチコンピュータシステムのための
プログラム言語
Dihybrid

指導教官　森下巖　助教授

1979年 3月

金田　泰

共有メモリをもたない計算機システムのプログラムを書く場合、処理装置ごとにプログラムをかくのは問題が多い。かといって、全く逐次的なプログラムをコンパイラで各処理装置にふりわけるのも難しい。この中間にあって、用途に応じたかきかたのできる柔軟さがあり、しかも効率のよい言語とその処理系はできないものだろうか。

このような問題に関する考察は、過度に逐次的だった従来のプログラミングのありかたに対する再検討をうながすものでもある。この論文は不完全ながらもこの問題に対する1つの解答を与えるものである。

No 11

Date

目次

No

Date

序

研究のテーマは multi(micro)computer system のための言語を設計し、そのコンパイラをつくることであった。そのコンパイラの target machine である M³-system は、1977年度の卒業研究でそのハードウェア（第2章の文献表の[Wa78], [Hi78]）と基本ソフトウェア（monitor, simulator）（同[Og78], [On78]）がつくられた。そして、そのプログラミングも試みられたが、利用できたのはアセンブリ言語だけであった。高級言語で書けば10分ほどで書けるプログラムを書くのにかなりの日数を要したということであり[Og78]、適当なコンパイラが必要とされていたわけである。

M³は共有メモリをもたない複数の処理装置からなる計算機システムであり、従来の言語をそのまま用いることは困難だった。そのため新たな言語 Dihybrid を設計してその implementation を試みた。コンパイラはまだ完成していないが、言語設計はもちろん、implementation もほぼその方策が定まっているので、それらについて述べる。

第1部で Dihybrid の文法を定義したが、これは、
第2部、第3部を読むときに参照するだけでもよい。

No.

Date

第1部

プログラム言語 Dihybrid

No

Date

プログラム言語 Dihybrid II

vers. 1981. 3. 3.

第1部では プログラム言語 Dihybrid の仕様を定義する。Dihybrid は かんたんな構文で様々の並列処理の問題をとくために設計された言語であり、主に multi computer system に implement することをめざしている。言語設計は基本的に Hoare [Ho78] に負っているが、効率をあげるために、手続き、関数など、従来のプログラム言語の制御構造もとりいれている。

この仕様は実験的な system のためのものであり、実用には充分でないと思われるが、やや拡張すれば systems programming から application programming まで、幅広い用途に供しうるものと考えている。

0. Dihybrid IIの特徴

Dihybrid IIの主な特徴は次のとおりである（詳しいことは［第2部］を参照されたい）。

1. 並列命令と入出力命令

並列化された手続きを記述するための手段として並列命令と入出力命令がある。

並列命令は、1つのプロセスを複数のプロセスに分けて共時に走らせる（multi-microcomputer systemにimplementされた場合は、並列命令の実行前にidleであった処理装置が起動される）。分かれたプロセスは並列命令の終わりで再び1つにまとまる。

入出力命令は、並列命令によって分かれたプロセスの間の通信（すなわち値の受けわたし）をおこなうものである。外部との通信（すなわち、普通のプログラム言語でいうところの入出力）も入出力命令によっておこなう。

2. 被護命令と非決定性

Dihybrid IIには普通の意味のif文やwhile文(DO文)はない。そのかわり、選択命令とくりかえし命令がある。選択命令と普通のif文、くりかえし命令と普通のwhile文の主なちがいは、非決定性があるかないかということである。たとえば、次の選択命令をみてみよう(§3.4 2.2.6でも同じ例を用いた)。

```
if   x≧y → m:=x
□    y≧x → m:=y
fi.
```

x>yのときは m:=xが実行され、y>xのときはm:=yが実行される。 x=yのときは m:=x, m:=yのうちどちらか一方が実行されるが、どちらが実行されるかはわからないというたてまえになっている)。 この非決定性はアルゴリズムから不必要な非対称性を除く働きをするが、複数のプロセスを用いる実時間プログラミングの場合に従って、並列

命令、入出力命令と組みあわせて用いる場合）にはもっと本質的な意味をもつ（金田[1979]第2部参照）.

3. マクロと準マクロ機能

Dihybrid IIにはマクロ手続きとマクロ関数がある。これらは手続き、関数が1つのプロセスの中だけで使えるのに対し、プログラム全体で有効であり、手続き、関数の機能をおぎなっている（§1.4参照）。また、同じ形の複数のプロセスや被護命令をまとめて書く記法や、入出力命令の複数のパラメタ列をまとめて書く記法は、プログラムの簡潔さを増すとともに、定数宣言とあわせて使うことにより、プログラムの変更を容易にする。

1. 基本概念

この章ではDihybrid IIIの基本概念を説明する。ただし、被護命令（guarded command）については述べない*ので、それに関しては金田[1979]第2部を参照されたい。また、プロセスとプロセス間の通信に関してはHoare[Ho78]を参照されたい。

なお、この言語はHoare[Ho78]の言語を基にしているが、それと記法上、また概念上異なる部分も多いので、誤解のないように願いたい。

＊§3.4.2.2で、この言語にそくして被護命令の解説をしてある。Dijkstraの被護命令とはやや異なったところがあることを注意しておく。

1.1 量とその種類

次のものを量と称する：

型、定数、変数、マクロ手続き、マクロ関数、手続き、関数、プロセス、路、プロセス名札の範囲パラメタ.

これらには（一部の 型 を除いて）名前をつけることができる。

1.2 可視範囲則

量につけた名前が、その量をあらわすものとして解釈される文面上の範囲を可視範囲といい、それを定める規則を 可視範囲則 という。DihybridIIには基本的に2種類の可視範囲則があり、量の種類と その宣言場所によって どちらの規則をとるかが決まる。

(1) 大域的可視範囲則

この規則が適用される量はプログラムの最も外側

のブロックより前で宣言されるか、または無宣言で使用できるもののすべてであり、それだけである。これらの可視範囲はそのブロック全域である。マクロ手続き、マクロ関数、大域的定数に適用される。

(2) 局所的可視範囲則

この規則が適用される量は、ある構文単位の、その宣言以降の部分が可視範囲である。その構文単位を有効範囲とよぶ。すなわち、量Qの宣言Qを含む有効範囲 αQβ があるとき(α, βは記号列)、Qはβで可視だが、αでは可視でない。

ただし、宣言は、その有効範囲の中にある並列命令の中では、無条件に可視ではない。その並列命令の輸入宣言にあらわれる量は、その並列命令の中でも可視である。

この規則は 大域的定数を除く定数、変数、手続き、関数、マクロとルーティンのパラメタ、プロセス、路に適用される。定数、変数、手続き、関数の場合

には、それがブロックの要素である命令列の要素としてあらわれたときはそのブロックが有効範囲であり、被護命令の要素（護衛列または命令列）の要素としてあらわれたときはその被護命令が有効範囲であり、マクロ、ルーティンのパラメタの場合は、そのマクロ、ルーティンの宣言全体が有効範囲である。路およびプロセスの場合は、その宣言がおこなわれた並列命令が有効範囲である。また、プロセス名札の範囲パラメタの場合は、それがプロセスの要素として（p[i:1..n]:: ... の形で）あらわれたときはそのプロセス本体であり、入力（出力）パラメタ列にあらわれたときは、その入力（出力）パラメタ列である。

上記の2つの規則のうちいずれに従う場合も、同じ名前の2つの宣言の、上の規則による可視範囲が重なってはならない（いいかえると、ひとつの宣言の可視範囲内で同じ名前を再び宣言することはでき

ない*。(ただし、手続き、関数、プロセスのforward宣言の場合は別である)。

* 可視範囲が重ならなければ、同じ名前を別の量をあらわすものとして用いることができる。

例1

```
1.  mfunc add(a,b: int) = a+b;
2.  const  m=2;
3.  begin  const n=5; var v: int;
4.     proc P= begin var w: Bool; ---   end;
5.     v:=n;
6.     var  x:char; ---
7.     co  imports v;
8.        x:: define v; ---
9.     ||y:: use  v; ---
10.    oc
11. end.
```

マクロ関数add,(大域的)定数 m は3~11行で可視、定数nは3,5~6行で可視 変数v は3,5~10行で可視(ただし 9行では代入はできない)。

変数wは4行(の宣言のあと)で可視 変数xは6行で可視である(並列命令内の可視範囲に関しては§3.4.2.1で詳しく述べる。また、プロセス名札の範囲パラメタに関しては§3.4.1.2を参照されたい)。

例　次のプログラムは正しくない:

```
const n=5;
begin
  proc P= begin var n; int; ---  end;
  ---
end.
```

手続きPの本体の中で大域定数nと変数nの可視範囲が重なっているからである。■

例　次のプログラムは正しい(省略したところに誤まりがないとすれば):

```
begin
 proc P= begin const n=5; ---  end;
 proc g(n; Bool) = begin  ---  end;
 func n returns result; int =
   begin ---- . end;
 ---
end.
```

nは3回宣言されているが、その有効範囲は重なっていない。関数nの宣言を手続きgの宣言より前におくと誤まりになる。■

1.3 プロセスとプロセス間通信

一時に1つずつ操作を実行するときの、計算の流れを逐次プロセス（sequential process）といい、略して単にプロセスという。DiyhybridIIのプログラムの実行が開始されるときは、標準プロセスを除けば"ただ"1つのプロセスが存在する。そのプロセスを主プロセスという。

プロセスは並列命令によって複数にわかれ、並列命令のおわりで"待ち合わせて再び1つにまとまる。プロセスがわかれた状態では、それぞれの計算の相対速度は不確定であるが、これらのプロセスの同期をとり、情報をやりとりするために入出力命令がある。

入出力命令(のパラメタ列)は、常に2つのプロセスで対をなしてあらわれる。対をなす一方の実行が開始されると、そのプロセスは、もう一方が実行されるまで"待たされる。そして、出力パラメタの式が、代入文と同様にして入力パラメタの変数に代入される。

対をなす入出力命令を識別するために、路という量が導入されている。　すべての入出力命令は路を指定する。

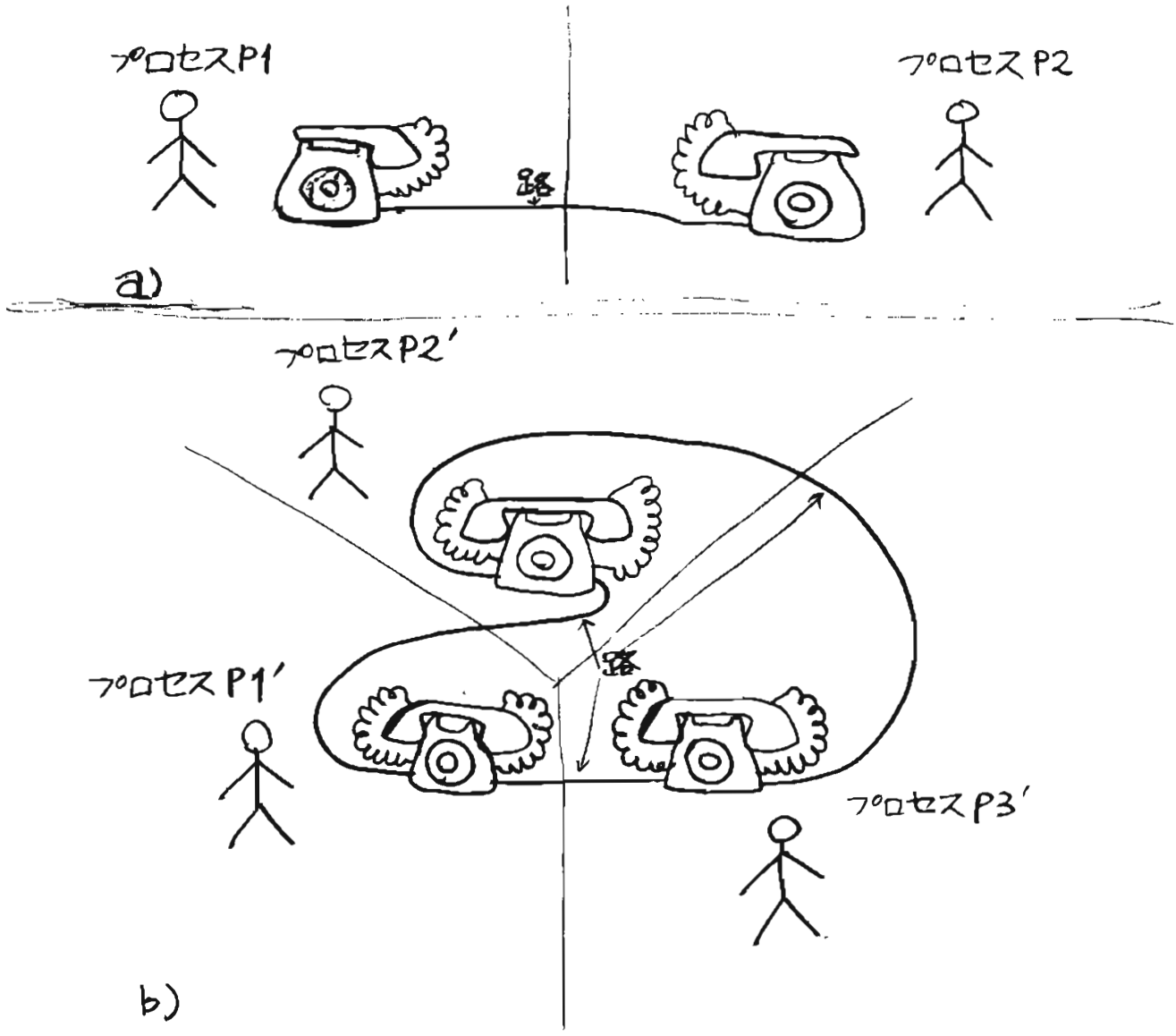


図1.　DiHybrid IIにおける入出力の概念のたとえ

例

x:: ... chan!y(47); ...

y:: ... chan?x(v); ...

2つの入出力命令（chanではじまる命令）が同時に実行される状態にあれば、プロセスyは、chan?x(v)の実行後 v=47 をえる。chanは路につけられた名前である。

1.4 マクロとルーティン

プログラムの一部をモジュール化するために、マクロとルーティンが用意されており、それぞれに手続きと関数とがある。マクロの場合はマクロ手続き、マクロ関数と呼び、ルーティンの場合には単に手続き、関数とよぶ。

（マクロ手続きと手続きを合わせて手続き類、マクロ関数と関数をあわせて関数類とよぶ）。

1.4.1 マクロとルーティンの違い

マクロとルーティンは次のようなちがいがある。これは手続き類の場合にも、関数類の場合にも共通である。

(1) マクロは大域的可視範囲則に従うが、ルーティンは局所的可視範囲則に従う。しかも、並列命令はルーティンを「輸入」*することはできないから、その中で外のルーティンを使うことはできない。だからルーティンは1つのプロセスでしか実行することができないが、マクロは複数のプロセスで同時または共時(以下共時という)に実行することができる。**

(2) ルーティンはその本体からルーティンをよぶことができるが、マクロはその本体からマクロをよぶことはできない。とくに、ルーティンは再帰的に用いることができ

---

* 「輸入」の概念については 3.4.2.1 参照。

** マクロ宣言の本体の中で宣言された変数は、マクロを実行するプロセスそれぞれに対して異なる(共有されるのではない)。

きるが、マクロはそれができない。

2. 記法

構文は BNF (Backus Naur Form / Backus Normal Form) に多少の変更を加えた記法で記す。

< > で囲んだ記号は非終端記号であり、それと超記号を除く記号が終端記号である。終端記号は 3.1 で定義する。超記号には次のものがある;

::= | { } * + #

::= は右辺の記号列が左辺の非終端記号から導出されることをあらわす。右辺に | があらわれたときは、それで区切られた各記号列がそれぞれ左辺の記号から導出されることをあらわす。任意の記号列 ξ, 任意の終端記号 x に対して、

$\{\xi\}^*$ は ξ の 0 回以上のくりかえしをあらわす。

$\{\xi\}^\#$ は <empty> | ξ と同一、

$\{\xi\}^*_x$ は <empty> | $\xi\{x\xi\}^*$ と同一、

$\{\xi\}^+$ は $\xi\xi^*$ と同一、

$\{\xi\}^+_x$ は $\xi\{x\xi\}^*$ と同一のことをあらわす。

ただし、<empty>は空列である。

$A ::= B_1 B_2 \cdots B_n$ が構文規則であるとき、$B_1, B_2, \cdots, B_n$ をAの要素とよぶ。ただし、$B_1, B_2, \cdots, B_n$ は非終端記号または終端記号である。

3. 規準文法

規準文法は Dihybrid を使用するとき、および implement するときの基準となるべき文法である。実際の処理系にはこの文法にない規則、制限が追加されるであろう。

3.1 基本記号

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M
| N | O | P | Q | R | S | T | U | V | W | X | Y | Z
| a | b | c | d | e | f | g | h | i | j | k | l | m
| n | o | p | q | r | s | t | u | v | w | x | y | z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<underscore> ::= _

<special symbol> ::= + | − | * | = | < | > | ≦ | ≧ | ≠ | (|)
| , | [|] | := | . | ; | : | ' | → | □ | ‖ | ! | ? | ..
| :: | div | mod | or | and | not | if | fi | do | od
| co | oc | begin | end | const
| var | func | proc | forward | channel | import
| ref | define | use | mproc | mfunc | returns

(≦ は ≤ , ≧ は ≥ のように記してもかまわない)

規準文法では 太字が使用可能のときは 英字からなる特殊記号は太小文字で記され、使用可能でないときは

細小文字に下線を引いたもので記す。また letter は
下線をひかない細字で記す。

3.2 名前、定数

<identifier> ::= <letter> {{<under score>}# <letter or digit>}*

<letter or digit> ::= <letter> | <digit>

<unsigned integer> ::= {<digit>}+

<string> ::= '{<'を除く任意の文字または''>}*'

<unsigned const> ::= <unsigned integer> | <string>
| <const id>

<const id> ::= <identifier>

名前(identifier)は、量を識別するために、1つの量に対して1つつけられるものである。符号なし整数(unsigned integer)は、それが通常あらわす整数値をあらわし、その型はintである(3.6参照)。文字列(string)は' 'ではさまれた文字列をあらわす。ただし、その中にあらわれる''は'と解釈される。文字列はそれが1文字からなるときchar型であるが、それ以外のときは標準の形(int, char, Bool)とは整合しない(標準プロセスoutへの出力パラメタとしてのみあらわれうる)。

次の名前は その横に記した型をもつ大域定数名である。* これらは宣言せずに使える。

true : Bool

false : Bool

eol : char

* これらの名前を予約語とみなすこともできる。その違いは本質的なものではない。

3.3 変数と式

3.3.1 変数

<variable> ::= <var id> { [<expr list>] }#
| <param id> { [<expr list>] }#

<var id> ::= <identifier>

<param id> ::= <identifier>
<expr list> ::= { <expr> }+,

[<expr list>]（これを添字とよぶ）のついた変数は配列要素であり、配列として宣言された変数のみが、この形をとることができる。添字の値は宣言された範囲内になければならない。名前だけからなる変数は、単純変数または配列である。

変数には int, char, Bool という名前であらわされる型がある（3.6参照）。また、それらの配列型がある。

3.3.2 式

<const expr> ::= <expr> (1)

<Bool expr> ::= <expr> (2)

<expr> ::= <rel expr>{and <relexpr>}*
| <rel expr>{or <relexpr>}*

<rel expr> ::= <simple expr>{<relop><simple expr>}#

<rel op> ::= = | < | > | ≦ | ≧ | ≠

<simple expr> ::= {<sign>}#<term>{<addop><term>}*

<sign> ::= + | −

<add op> ::= + | −

<term> ::= <factor>{<mul op><factor>}*

<mul op> ::= * | div | mod

<factor> ::= <variable> | <unsigned const>
| <func kind designator> | (<expr>) | not <factor>

(1) 定式(const expr)の中の因子(factor)は変数および関数類呼びだしを含んではならない。

(2) 論理式(Bool expr)は、式のうち、その型がBoolであるものをいう(3.3.3参照)。

式にも、変数と対応して、int, char, Bool およびそれらの配列の型がある。式の型を決定する規則は以下に述べる。

3.3.3 演算子

3.3.3.1 演算子の意味

各単項演算子の意味は表1のとおりである。2項演算子の左項と右項の型は等しくなければならない。各2項演算子の意味は表2のとおりである。単項演算子、2項演算子とも、配列には作用しない。

表1 単項演算子の意味

演算子	意味	被演算項の型	結果の型
+	無意味	int	int
−	加法に対する逆元	int	int
not	否定	Bool	Bool

表2　2項演算子の意味

演算子	意　　味	被演算項の型	結果の型
=	左項と右項が等しいときtrue、 等しくないときfalse	int, char	Bool
<	左項が右項より小さいときtrue、 小さくないときfalse	int, char	Bool
>	左項が右項より大きいときtrue 大きくないときfalse	int, char	Bool
≦	左項が右項以下のときtrue、 そうでないときfalse	int, char	Bool
≧	左項が右項以上のときtrue そうでないときfalse	int, char	Bool
≠	左項と右項が等しくないとき true、等しいときfalse	int, char	Bool
＋	左項と右項の和	int	int

−	左項から右項をひいた差	int	int
or	左項、右項のいずれかが true のとき true、ともに false のとき false	Bool	Bool
*	左項と右項の積	int	int
div	左項を右項で割った商(1)	int	int
mod	左項を右項で割った余り(2)	int	int
and	左項、右項がともに true のとき true、いずれかが false のとき false	Bool	Bool

(1) $n \geq 0$, $m > 0$ のとき $0 \leq n - n \underline{div}\ m * m < m$ …①

がみたされる。($n \geq 0$, $m < 0$), ($n < 0$, $m > 0$), ($n < 0$, $m < 0$) の3つの場合に、それぞれ ① または $-m < n - n \underline{div}\ m * m \leq 0$ がみたされる。$m = 0$ のときは 誤りとされる。

(2) $m \neq 0$ のとき $n = n \underline{div}\ m * m + n \underline{mod}\ m$

がなりたつ。$m = 0$ のときは 誤りとされる。

(注意)

以上の表において、意味の欄にあらわれる true, false はそれぞれ「true と同じ値」,「false と同じ値」の意味である。以下でも true とそれがあらわす値、false とそれがあらわす値とを同一視する。

3.3.3.2 演算子の順位

因子(factor)にあらわれる演算子の順位がもっとも高く、項(term)の要素としてあらわれるものがこれにつぎ、以下、単純式(simple expr)、式の順となる。すなわち、次の順である。

not

*, div, mod

+, −

=, <, >, ≦, ≧, ≠,

and, or

順位の等しい演算子が並んでいるときは、左から順に評価するか、またはそれと同じ意味になるように評価する。( ) 内の式はその前後の演算に先だって評価する。すなわち、( ) はその中の演算子の順位を、その外の not より高める働きをする。

3.3.4 関数類呼びだし*

<funckind designator> ::= <funckind id> { ({ <actual param> }+,)# -
<funckind id> ::= <identifier> -
<actual param> ::= <expr>

関数類呼びだし(funckind designator)は、その関数名がマクロ関数として宣言されたものならマクロ関数呼びだし、関数として宣言されたものなら関数呼びだしと解釈される。そのいずれとしても宣言されていないときは誤りとされる。

いずれの場合も、呼びだされた(マクロ)関数は、それを呼びだしたプロセスが実行する。従って、マクロ関数は複数のプロセスで共時に実行することができる(§1.4参照)。

---

* この節の記述は関数類の implementation のしかたを記述するものでないことを注意しておく.

3.3.4.1 マクロ関数呼びだし

マクロ関数呼びだしは その関数名をもつマクロ関数宣言の本体（式）の前に（，後に）をつけたものを、そのマクロ関数よびだしとおきかえたものと同じ意味をもつ。ただし、パラメタがあるときは、後述する マクロ手続き呼びだしの場合と同様におきかえられる（3.4.1.3.1 参照）。

例　4桁のBCDを 2進数に変換するマクロ関数：

```
mfunc  dectobin (dec:int) =
  dec div 4096 * 1000 + dec mod 4096 div 256 * 100
+dec mod 256 div 16 * 10 + dec mod 16
```

と宣言されているとき、 −dectobin(x) は、

−(x div 4096 * 1000 + x mod 4096 div 256 * 100
+x mod 256 div 16 * 10 + x mod 16)

と同義である。■

3.3.4.2 関数呼びだし

3.3.4.2.1 関数呼びだしの意味1

護衛列以外の場所にあらわれる関数呼びだしを含む命令の意味は、次のようにして作られる命令列と等しい。

注目する関数呼びだしを含む命令のうち、その長さが最も短いものについて、その関数呼びだしより左の記号列を l, 右の記号列を r とする。すなわち、その命令は $l\ f(Ap)\ r$ という形をしている(ここで f が呼びだすべき関数名をあらわし、(Ap)は実パラメタ列をあらわす)。呼びだされるべき関数の宣言は

func f(P) returns V:T = Body

とする((P)はパラメタ列、Vは変数名、Tは関数の型、Bodyは関数本体をあらわす)。

このとき、$l\ f(Ap)\ r$ を次の命令でおきかえる。

begin var V':T; Body'; l V' r end

ここで、Body'は、Bodyの一部におきかえをほどこしたものをあらわす。これに関してはパラメタの実パラメタによるおきかえと有効範囲の衝突をさけるための名前のおきかえが必要になるが、その機構は手続き呼びだしの場合と同じであり、手続き呼びだしのところで後述する。ただし、その場合のおきかえに加えて、Bodyの中にあらわれるVをV'でおきかえる必要がある。V'はプログラムの中でまだ使われていない適当な名前とする。(上の命令列にあらわれるV'はこれと同じ名前である。)

例

次の関数宣言があるとする.

```
func add(a,b:int) returns sum:int =
begin sum := a+b end.
```

このとき

1) 代入命令の場合1.

```
v := a + add(b,c)
```

は

```
begin var sums:int; begin sums := b+c end end;
v := a + sums
```

と同じ意味である(sumsは新しい名前とする)。//

2) 代入命令の場合2

```
v[add(b,c)] := a
```

は

```
begin var sums:int; begin sums := b+c end end;
v[sums] := a
```

と同じ意味である(sumsは新しい名前)。//

3) 入出力命令の場合

c!p(a, f(b,c))

は、それが譲渡引列にあらわれる場合を除けば、

```
begin
  var sums: int; begin sums := b+c end;
  c!p(a, sums)
end
```

に等しい。(sumsは新しい名前)。//

関数呼びだしは、上の例のほか入出力文の入力パラメタ列にあらわれる変数の添字の中、手続き命令の実パラメタにあらわれるが、それらも、上の例と同様にしておきかえられる。■

3.3.4.2.2 関数呼びだしの意味2

護衛列に関数呼びだしを含む被護命令の意味は次のようにして作られる命令列と等しい(この節は「くりかえし命令と選択命令」の節を読んでから読まれたい)。

3.3.4.2.2.1 選択命令の場合

注目する関数呼びだしを含む選択命令の、その関数呼びだしより左の記号列を ℓ, 右の記号列を r とする。すなわち、その選択命令は $\ell f(Ap) r$ という形をしている。呼びだされるべき関数の宣言の形は 3.3.4.2.1 と同じとする。このとき、$\ell f(Ap) r$ を次の命令でおきかえる。

~~begin~~ ~~var~~ v' : T ; Body' ; $\ell v' r$ ~~end~~

ここで、v', Body' は 3.3.4.2.1 と同じである。

例

3.3.4.2.1 の例と同じ関数宣言が与えられているとする。このとき、

if .... a = f(b,c) → ... fi

は

var sums : int ; begin sums := b+c end ;

if ... a = sums → ... fi

と同じ意味である（sums は新しい名前）。■

例 3.3.4.2.1 の例と同じ関数宣言のもとで、

if [i : 1..2] C[i] ! P[i](f(b,c)) → CS(i) fi

は

var sums1 : int ; begin sums1 := b+c end ;

var sums2 : int ; begin sums2 := b+c end ;

if C[1] ! P[1](sums1) → CS(1)

□ C[2] ! P[2](sums2) → CS(2)

fi

と同じ意味である（sums1, sums2 は新しい名前）。■

後者は前者に対して おきかえ規則を2回適用した結果である。■

3、3、4、2、2、2 くりかえし命令の場合

前節と同様に、注目する関数呼びだしを含むくりかえし命令を l f(Ap) r とする。呼びだされるべき関数も同様とするとき、このくりかえし命令を次の命令でおきかえる。

begin var V'; T; Body'; l'V'r' end

ここで、V'; Body' の意味は 3、3、4、2、1 と同じである。l', r' は、それぞれ l, r に次のようなおきかえをほどこしたものである。

そのくりかえし命令のすべての □ を、; Body' □ でおきかえる（この Body' は、Body にそれぞれの場所で必要なおきかえをしたものであって、互いに等しいとは限らない）。Body から Body' へのおきかえは手続き命令に関して後述するのと同じ手順でおこなう。

また、rの最後の od を、;Body' od でおきかえる。

例 3.3.4.2.1 の例と同じ関数宣言が与えられているとき、

```
do a=f(b,c) → CS od
```

は

```
begin
 var sums : int ; begin sums := b+c end ;
do a = sums →
      CS ; begin sums := b+c end
od
end
```

と同じ意味である (sums は新しい名前)。 ■

例 3.3.4.2.1の例と同じ関数宣言のもとで、

```
do [i : 1..2] c[i]!p[i](f(b,c)) → CS(i) od
```

は

```
begin
 var sums1 : int ; begin sums1 := b+c end ;
begin
 var sums2 : int ; begin sums2 := b+c end ;
do c[1]!p[1](sums1) →
```

CS(1);

begin sums1 := b+c end; begin sums2 := b+c end

[] c[2]!p[2](sums2) →

CS(2);

begin sums1 := b+c end; begin sums2 := b+c end

od
end end
と同じ意味である（sums1, sums2は新しい名前）。■

3.3.4.2.3 再帰呼びだし

関数は再帰的に呼びだすことができる。また、関数の中から他のルーティンやマクロをよびだすことができる。

例　階乗の計算

```
func factorial (n : int) returns result : int =
  begin
    if  n=0  →  result := 1
    □   n>0  →  result := factorial (n-1) * n
    fi
  end
```

[注意]　n<0のときは実行時にあやまりとされるかデッドロックをおこす。■

3.3.4.3 標準マクロ関数

3.3.4.3.1 標準マクロ関数の定義

プログラムの中で宣言しなくても使用することができるマクロ関数を標準マクロ関数という。プログラムの中で標準マクロ関数をあらわす名前を別の意味で使うことはできない。ユーザ・プログラムは次のようなプログラムの一部と考えることができる。

```
mfunc abs(x: int)= ---;
mfunc ord(x: char)= ---;          } 標準マクロ関数
mfunc chr(x: int)= ---;
 begin
   co channel read, write;
     in :: ---
     out :: ---                   } 標準プロセス (3.4.1.2, 3.4.2.1参照)
     user :: USER PROGRAM
   oc
end.
```

ただし、これは正確な表現ではない。

3.3.4.3.2 標準マクロ関数の種類

標準マクロ関数としては次のものが用意されている*。

abs(x)： xの絶対値を求める（ int → int ）。

ord(x)： charから0,1,…,n-1（nは使用可能な文字の数）への1対1の写像（ char → int ）。

chr(x)： ordの逆関数（ int → char ）。

ただし、x<0, n-1<xに対しては値は定義されていないから、誤りとされる。

ord, chrに関しては 4 を参照のこと。

＊標準マクロ関数は必ずしもいわゆる"マクロ"として、すなわち open subroutine という形で implement されるとは限らない。

3.4 命令

<command> ::= <simple command> | <structured command>

命令(command)は、単純命令と構造化命令に分けられる。構造化命令は、その中に再び命令列を含んでいる。

3.4.1 単純命令

<simple command> ::= <proc kind command>
| <assignment command>
| <I/O command>
| <null command>

単純命令には手続き類命令(proc kind command)、代入命令(assignment command)、入出力命令(I/O command)、空命令(null command)の4つがある。

3.4.1.1 代入命令

<assignment command> ::= <variable> := <expr>

代入命令は、右辺の式の値を左辺の変数に代入する。両辺の型は等しくなければならない。

例

1) var b : Bool, i, j : int; ... b := i = j

2) var a : [1..5] int, b : [0..4] int; ... a := b

1) は正しい例であり、2) は誤った例である。ただし、1) の代入文の前に i, j の値が定義されている必要がある。2) を

var a : [1..5] int, b : [1..5] int; ... a := b

とすれば正しい。■

3.4.1.2 入出力命令

<I/O command> ::=
  <channel> {<output param sequence>}$^{+}$ {<input param sequence>}*
 | <channel> {<input param sequence>}$^{+}$

<channel> ::= <channel id> {[<const expr>]}$^{\#}$

<channel id> ::= <identifier>

<output param sequence> ::= !<process label> ({<expr>}$^{*}_{,}$)

<input param sequence> ::= ?<process label> ({<variable>}$^{*}_{,}$)

<simple output command> ::= <channel id><output param sequence>

<simple input command> ::= <channel id><input param sequence>

<process label> ::= <process id> {[<range>]}$^{\#}$

<range> ::= <range param> | <const subrange type>
 | <const expr>

<range param> ::= <const id>

<process id> ::= <identifier>

<const subrange type> ::= <lower bound>..<upper bound>

<lower bound> ::= <const expr>

<upper bound> ::= <const expr>

{<simple input command>, <simple output command>}$^{+}$

3.4.2.2で使われる。}

3.4.1.2.1 入出力命令の意味

入出力命令は、複数のプロセス間の同期をとり、情報を交換するための命令である。入力パラメタ列(input param sequence)は、名札で指定したプロセス(以下相手プロセスとよぶ)から同期信号と情報をうけとることを指定し、出力パラメタ列は名札で指定したプロセスへ同期信号と情報を送ることを指定する。

通信が成立するためには、入出力命令の実行中にそれが指定するすべての相手プロセスが同じ路(channel)を冠した入出力命令を実行し、*,+

---

* 同じ路名をもつすべての入出力命令が実行されなければならないわけではない([第3章]参照)。

+ (次ページ)

かつ パラメタが 正しく対応しなければならない。同じ路とは、路名が等しく、かつ路配列の場合には添字の値も等しい路ということである。路が添字をともなうのは それが路配列として宣言された場合に限り、添字は宣言された範囲内になければならない。

パラメタの対応は次のように定める。*

(1) 任意の入出力命令の任意の出力パラメタ列について、次の条件がなりたつ。その入出力命令が属するプロセスをx、その出力パラメタ列が指定する相手プロセスをyとすると、yにあらわれる同じ路をもつすべて

---

* (1), (2)がみたされていないとコンパイル時誤りとなる。

+ 入出力命令は相手プロセスの対応するすべての入出力命令が実行されるまで終了しない。従って、デッドロックがおこりうる。

例
```
co channel c1,c2 ;  y:: forward
 || x:: var v: int ; c1?y(v) ; c2!y(ex)
 || y:: var v: int ; c2?x(v) ; c1!x(ey)
oc
```
(ただし、ex, eyは適当な式とする) ■

の入出力命令は 相手プロセスを文とする入力パラメタ列」をもっていること。

(2) これらのパラメタ列にあらわれる変数と式の数は等しく、左から順に対応づけると、対応する変数と式の型は等しい。

1つの相手プロセスに対するパラメタ列」は、入力パラメタ列、出力パラメタ列とも1個以下でなければならない。

従って、(1)は「入力」と「出力」を交換しても真である。
上の対応則の意味については〔付録3〕でさらに述べる。

入出力命令は 相手プロセスのうち1つでもすでに停止したものがあれば、通信は成功せず、護衛列にあらわれる単純入力命令と単純出力命令の場合を除いて、誤まりとされる。*
また、相手プロセスは、その入出力命令を含むプロセスであってはならない。

1つの入出力命令の中のパラメタはすべて1度に送受される。

---

* 単純入力命令と単純出力命令　はくりかえし命令と選択命令の中にだけあらわれる。従ってそこで解説する(3.1.2.2)。

従って 次の入出力命令では デッドロックはおこらない。*

co channel c) y :: forward || z :: forward
|| x :: var v : int ; c ? y(v) ! z(e_x)
|| y :: var v : int ; c ? z(v) ! x(e_y)
|| z :: var v : int ; c ? x(v) ! y(e_z)
oc.

(e_x, e_y, e_z は適当な式とする)。

* これに対し、次の入出力命令列ではデッドロックがおこる。

co channel c1, c2, c3 ; y :: forward || z :: forward
|| x :: var v : int ; c1 ? y(v) ; c3 ! z(e_x)
|| y :: var v : int ; c2 ? z(v) ; c1 ! x(e_y)
|| z :: var v : int ; c3 ? x(v) ; c2 ! y(e_z)
oc.

(いずれかのプロセスの 2つの入出力命令の順序を逆にするとデッドロックは防げる)。

入出力命令にあらわれる路名は、それを含む並列命令のうち、もっとも内側のもので宣言された路名でなければならない(すなわち、路名は局所的有効範囲規則に従う)。
(以下の例は[付録3]で一般化されている.)

例1 プロセスPの中に c!g(x)!r(y) なる入出力命令があるとき、Pの中に c!r(y)!g(x) があらわれてもよい。しかし、c!g(x), c!r(y), c?g(x)?r(y), c!s(z), c!g(x)!r(y)!s(z) などがあらわれると、他のプロセスの如何にかかわらず、それらは誤まりとみなされる。e_1 が x と同じ型、e_2 が y と同じ型であれば、もちろん c!g(e_1)!r(e_2) も正しい。■

例2 プロセスPの中に c!g() なる入出力命令があるとき、Pにあらわれる c を冠する入出力命令はすべてこの形でなければならない。また、プロセスgにあらわれるcを冠する入出力命令は、?p() という入力パラメタ列をもたなければならない。■

例3　プロセスPの中に　c!r(e)?g(v)　なる入出力命令があり、プロセスgの中に c!p(e')?r(v') なる入出力命令があるとする。このときプロセスrにあらわれるcを冠する入出力命令はすべて !g(e') なる出力パラメタ列と、?p(v") なる入力パラメタ列をもたなければならない。p, g, rのほかに、同じ並列命令で宣言されたプロセスが存在しないときは、この入出力命令は c!g(e")?p(v") の形である。ただし、eとv", e'とv, e"とv'は同じ型である。■

3.4.1.2.2 プロセス配列との入出力

出力パラメタのプロセス名札はプロセス配列名であってもよい。そのときは、配列要素のすべてのプロセスとの間で通信がおこなわれる。たとえば、プロセスPが P[i:1..3] :: --- と宣言されていれば、c!p(e)は

c!P[1](e)!P[2](e)!P[3](e)

と等しい。

ただし、その入出力命令を含むプロセスが、その配列要素であるときは、自分は相手プロセスの中から除かれる。すなわち、前の例で、c!p(e)をおこなうのがP[2]であれば、これは c!p[1](e)!p[3](e) と等しい。

入力パラメタのプロセス名札はプロセス配列名であってはならない。*

また、 c!P[i:1..3](v[i]) は

c!P[1](v[1])!P[2](v[2])!P[3](v[3])

に等しい。この場合もプロセス名札がプロセス配列名のときと同様に扱われる。(注意: 上の例で、この入出力命令のある場所が別のiという名前の有効範囲にはいっていてはならない。iはプロセス名札の中で宣言されたものとみなされ、それを要素とする 出力 パラメタ列の中で有効である。)

例 次の入出力命令は誤まりである。なぜなら、前節例1と同様、ひとつの路を指定する入出力命令が異なるプロセスを指定する入力パラメタをもっているからである。

if [i; 1..5] c?p[i]() → CS fi.

これは次のように直せばよい。

if [i; 1..5] c[i]?p[i]() → CS fi.

もちろん、Cの宣言や、対応する入出力命令をかきなおす必要がある。■

3.4.1.2.3 標準プロセスとの入出力

外部装置との間の入出力は、標準プロセスを通しておこなうことができる。標準プロセスとは、プログラムの中で宣言しなくても存在するプロセスのことをいう(3.3.4.3参照)。標準プロセスをあらわす名前を別の意味で使うことはできない。

標準プロセスおよびそれと他のプロセスとの間をつなぐ標準路には次のようなものがある:

プロセス in　　関連する路 read

プロセス out　　関連する路 write

以下これらを個別に解説するが、これらの路は普通の路とちがって、パラメタの数や型は自由であり、またこれらをどのような順序でおこなってもデッドロックはおこらない。なお、いずれも相手プロセスが1つであるから、パラメタ列は1つしかとらないことに注意されたい。

なお、標準プロセスおよび標準路は局所的有効範囲則に従う(すなわち、並列命令の中では使えない)。

3.4.1.2.3.1 in

in は標準入力装置からの入力をおこなうプロセスであり、入力がつきると停止する。次の語がこれに関して使える。

- read: 入力パラメタとして変数をかくと、それに標準入力装置から値が入力される。入力された値がパラメタの変数型のものでないときは誤りとなる。* パラメタの個数は1個以上の任意個数、型は int または char である。パラメタの個数が1個で、その型が char であるとき以外は read を用いる入出力命令が護衛列の中にあらわれてはならない。

  出力パラメタはあってはならない。

---

* 内部プロセスとの入出力においては型の誤りはコンパイル時に検出されるが、外部プロセスからの入力の場合は、型の検査をコンパイル時におこなうことはできないので、実行時に誤りが検出される。ただし、int, char 以外の変数がパラメタとしてあらわれると、コンパイル時誤りとなる。

例　次のプロセスは in からの入力を out へそのままコピーし、それが終わると停止する*。 (3.4.1.2.3.2)

x:: do var c:char; read?in(c) → write!out(c) od.

char 型のパラメタがあらわれたときに入力が行末に達すると、その入力パラメタには値 eol が代入される。(eol については3.2参照)。eol は入力行1行につき1回だけあらわれる。

* すなわち護衛行列で read を使えば、入力がつきたかどうかをためすことができる。

3.4.1.2.3.2 out

outは標準出力装置への出力をおこなうプロセスであり、他のすべてのプロセスが停止するとoutも停止する。
次の路がこれに関して使える。

- write : 出力パラメタとして式をかくと、その型がintのときは整数として、charまたはstringのときは文字として出力される。他の型は許されない。パラメタの個数は1個以上の任意個数である。入力パラメタがあってはならない。

出力パラメタがeolのときは1行改行する。eolが陽に出力パラメタにあらわれなくても、標準出力装置の1行(1レコード)の長さに制限があるときは、出力行がその長さに達すると1行改行する。

writeを用いる入出力命令は護衛列の中にあらわれてはならない。

例は 3.4.1.2.4.1 を参照のこと。

3.4.13 手続き類命令

<proc kind command> ::= <proc kind id> {({<actual param>}$^{+}_{,}$)}$^{\#}$

<proc kind id> ::= <identifier>

手続き類命令(proc kind command)は、その手続き名(proc id)がマクロ手続きとして宣言されているものならマクロ手続き命令、手続きとして宣言されているものなら手続き命令と解釈される。いずれの宣言もされていないときは誤まりとされる。

いずれの場合も呼びだされた(マクロ)手続きは、それを呼びだしたプロセスが実行する。従って、マクロ手続きは複数のプロセスで共時に実行することができる(1.4 参照)。

3.4.3.1 マクロ手続き命令

マクロ手続き命令は、その手続き名をもつマクロ手続き宣言(3.7)の本体(=ブロック)

を、そのマクロ手続き命令とおきかえたものと同じ意味をもつ。ただし、そのマクロ宣言に

マクロパラメタ(macro param)があるときは、それらが、マクロ手続き命令の実パラメタと左から1つずつ対応づけられ、マクロ本体にあらわれた前者をすべて後者(<variable>の形をしていないときは、その両側に"('と")"をつけたもの)*
とおきかえてからマクロ手続き命令へのおきかえをする。

従って、マクロ手続き命令、マクロ手続き宣言のいずれか一方のみにパラメタがあるか、両者のパラメタの数が異なるときは誤まりとなる。また、おきかえの結果正しくないプログラムになる場合も誤まりとなる。

マクロを再帰的によぶことはできない。

---

*演算の順序を保存するために( )が必要な場合がある。

例 次のマクロ手続き宣言があるとする。

mproc GPS(i, n, z, v) =

begin i:=1; do i≦n → z:=v od end

{ Knuth, Merner [KM61] の general problem solver に基く }

これを用いてベクトルの内積を求めるには

z:=0; GPS(i, n, z, z+a[i]*b[i])

とする。このプログラム部分は、次のプログラム部分と同義である。

```
z:=0;
Begin
  do i≦n → z:= z+a[i]*b[i] od
end
```

■

3.4.1.3.2 手続き命令

3.4.1.3.2.1 手続き命令の意味

手続き命令の意味は、次のようにして作られるプログラム部分と等しい。*(次ページ)

3.4.1.3.2.1.1 パラメタの対応

手続き命令はそこにあらわれる手続き名と同じ手続き名をもつ手続き宣言と対応づけられる。手続き命令が実パラメタをもつとき、それに含まれる式が、それに対応する宣言のルーティン・パラメタに含まれるパラメタ名と、左から1つずつ対応づけられる。

3.4.1.3.2.1.2 参照パラメタ

宣言において ref が冠されたパラメタ（ref から次の型名までの間にあるパラメタ名）を参照パラメタという。参照

パラメタに対応する式（参照実パラメタという）は<variable>の形をしていなければならない。その実パラメタが配列要素でないときは、そのまま手続き本体にあらわれるパラメタ名と同一の名前をすべてその実パラメタにおきかえる。その実パラメタが配列要素のときには、プログラムの中にあらわれていないある名前をint型の変数とする宣言と、それにその配列要素の添字の値を代入する命令を手続き命令の直前におく。そして、その配列要素に対応する手続き本体内の名前をすべて

<その配列名>[<あらたに宣言した変数名>]

でおきかえる。ただし、その結果手続き本体で宣言された変数の有効範囲が、実パラメタに含まれる同一の名前の有効範囲と重なるときは、変数名をあらかじめ組織的に新しい名前とおきかえる必要がある。

＊　ただし、以下の記述は手続き命令のimplementationを反映したものではない。手続きは通常closed subroutineとしてimplementされる。

3.4.1.3.2.1.3 値パラメタ

パラメタ名にrefが冠されていない場合はそのパラメタを値パラメタという。値パラメタの場合、パラメタ名を変数とする宣言と、それに対応する実パラメタの式の値を代入する命令を、次節でおきかえをすべき本体の左端のbeginの直後におく。ただし、その結果前節で述べたような有効範囲の衝突がおこるときには、前節と同様にこれを組織的におきかえる必要がある。

3.4.1.3.2.1.4 本体のおきかえ

手続き本体で宣言された名前の中で、手続き命令のあるブロックで宣言されているのと同じものがあれば、前者を新しい名前と組織的におきかえる。また、前節までの操作をすべてのパラメタに対して実行する。その後、手続き本体を

手続き命令とおきかえる。

例

```
begin
  var n:int, a:[1..5]int;
  proc p(x:int, ref y:int) =
    begin var z:int;
      z:=y; x:=x+z; y:=x+1
    end;
  n:=1; a[1]:=2; p(2, a[n])
end
```

は次のようにかきかえたものと同じ意味をもつ。

```
begin
  var n:int, a:[1..5]int;
  begin n:=1; a[1]:=2;
    var x:int; nn:int; x:=2; nn:=n;
    var z:int;
    z:=a[nn]; x:=x+2; a[nn]:=x+1
  end
end
```

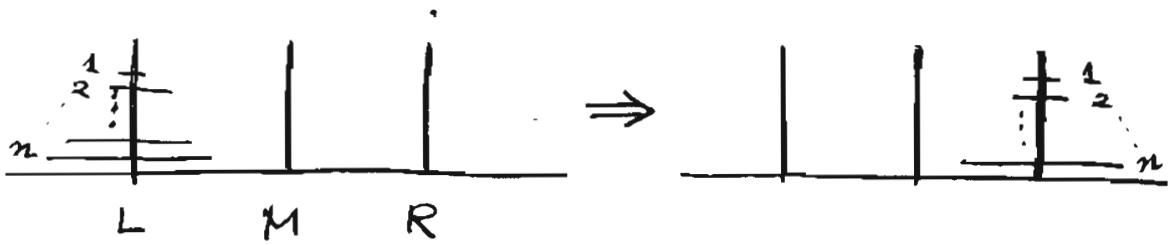
■

3.4.1.3.2.2 再帰呼びだし

手続きは再帰的によびだすことができる。

例　Hanoi の塔

```
begin
  proc Hanoi (n : int, l, m, r : char) =
   begin
     if n>0 → Hanoi (n-1, l, r, m);
              write!out ("Move piece", n, "from", l, "to", r, eol)
              Hanoi (n-1, m, l, r)
     □ n=0 →
     fi
   end;
  var n : int; read?in (n);
  Hanoi (n, 'L', 'M', 'R')
end.
```



■

3.4.1.4 空命令

<null command> ::= <empty>

空命令は何もしない命令である.

例　3.4.1.3.2.3 の □n≦0→ のところを参照されたい。■

例　;;;;;;■

3.4.2 構造化命令

<structured command> ::= <parallel command>
| <repetitive command>
| <alternative command>
| <compound command>

構造化命令には並列命令(parallel command)、くりかえし命令(repetitive command)、選択命令(alternative command)、複合命令(compound command)の4つがある。

3.4.2.1 並列命令

<parallel command> ::=
co {<imports list>;}# {<channel dec>;}#
{<process dec>}+|| oc

<process dec> ::= <process label>:: <process body>

<process body> ::= forward
| {<define list>;}# {<use list>;}# <command sequence>

<channel dec> ::=
channel {<channel id>{[<const subrange type>]}#}+;

<imports list> ::= imports {<data>}$_{,}^{+}$

<define list> ::= define {<define item>}$_{,}^{+}$

<use list> ::= use {<use item>}$_{,}^{+}$

<define item> ::= <variable>
| <var id> [<const subrange type>]

<use item> ::= <define item> | <const id>

{名札に関する構文は 3.4.1.2 を参照のこと}

3.4.2.1.1 並列命令の意味

並列命令の各プロセスは共時的、いいかえると並列に実行される。すなわち、並列命令の前に1つであったプロセスが、命令が終わるまで複数にわかれる。並列命令内のプロセスが交信する場合には路(channel)が使われて(3.4.1.2)、それを識別するための路名(channel id)と、路配列の場合にはその添字の範囲(上限、下限は定式)を路宣言(channel dec)で宣言しなければならない。入出力命令を使わないときは、路宣言は不要である。路は局所的有効範囲則に従う。

プロセスには名札をつけることができる。

プロセスの本体がforwardであるときは、あとで同名の、本体に命令列を含んだプロセスの宣言があらわれる。前の宣言があらわれてから後の宣言があらわれるまでの間にそのプロセス名があらわれると、それは後の宣言を参照するものとみなされる。この「forward宣言」は、命令列をともなった宣言の前であれば、何度あらわれてもよい。

3.4.2.1.2 プロセス配列

プロセス名札に添字がついているときは、プロセスの配列がとられ、各配列要素は並列に実行される。配列の添字の範囲は定範囲型(const subrange type)であらわされ、プロセスの本体では、範囲パラメタ(range param)が添字の値をもつ量として使える。* たとえば、

X[i:1..n]:: CS(i) は

X[1]:: forward || --- ||X[n]:: forward
||X[1]:: CS(1) || --- ||X[n]:: CS(n)

と等しい。

範囲パラメタは局所的有効範囲規則に従う。すなわち、そのプロセスの中だけで有効である。

---

* <lower bound>..<upper bound> を定範囲型とよぶのは Pascal [JW75] との類推からである。しかし、変数の型をあらわすのにこの形は使えない。下限(lower bound)および上限(upper bound)の型は int でなければならない。

3.4.2.13 プロセスに付随する宣言

使用リスト(use list)、定義リスト(define list)がないときには、そのプロセスは、それが属する並列命令の外部の名前を参照することはできない(たとえ輸入リスト《imports list》がなされていても)。使用リストにあらわれた定数、変数は式の中に使用することができ、定義リストにあらわれた変数はその上に値を変更することができる(定義リストに定数名をかいてはならない)。いずれの場合も、配列要素の場合はその添字が定式(const expr)でなければならない。*

---

* プロセス名札の範囲パラメタ(range param)は、定数として扱われ、この定式の中にあらわれうる。

例) x[i:1..n]:: use v[i]; define w[i+1]; ---

(次頁へつづく)

Vを任意の変数名、e_1, e_2を任意のint型の定式（$e_1 \leq e_2$）とするとき、 V[e_1..e_2]は

V[e_1], V[e_1+1], ---, V[e_2]

に等しい。

使用リストによって参照される変数の値は、他のプロセスに依存しない。また、並列命令の実行

（前ページから）

この例の意味は次のプログラム部分と同じである。

```
x[1] ;; use v[1]; define w[2]; ---
x[2] ;; use v[2]; define w[3]; ---
  :                  :
x[n] ;; use v[n]; define w[n+1]; ---
```

後も、それが定義リストにあらわれない、またはあらわれても代入がおこなわれない限りはじめの値が保存される。* また、代入がおこなわれた場合、最後に実行された代入の結果が保存される。

例　P:: define x[1]; use x; ---

x[1] は Pの実行によって値がかわりうるが、xの他の要素の値はかわらない。■

各プロセスの定義リストにあらわれる変数,定数は重複してはならない(すなわち、同じ変数の値を2つ以上のプロセスが定義することはできない)。使用リストの場合は、重複が許される。

---

* このように定義したのは、Dihybrid を、共有記憶のない system (multi-computer system) に implement することをめざしたからである。

なお、定義、使用が可能なのは定数、変数のみであり、手続き、関数などの他の局所的な量は常に参照することができない。

プロセスの使用リスト、定義リストにあらわれる変数、定数は並列命令のはじめの輸入リスト(*import list*)にかかなければならない。また、輸入リストにあらわれた変数、定数は、必ずいずれかのプロセスの使用リスト、定義リストにあらわれなければならない。

3.4.2.1.4 例

階乗の計算

```
co  channel knock[0..limit], reply[0..limit],
      knocksub[0..limit], replysub[0..limit];
  x :: USER  ||  fac[limit+1] ::
||  fac[i:1..limit] ::
  do  knocksub[i] ? fac[i+1]() →
      var r:int; CALLCHILD(r);
      replysub[i]!fac[i+1](r*i)
  □  knock[i]?x(y) →
      var r:int; CALLCHILD(r);
      reply[i]!x(r*i)
  od
||  fac[0] :: do knocksub[0]?fac[1]() → replysub[i]!fac[1](1)
            □ knock[0]?x() → reply[0]!x(1)
            od
oc
```

xは fac を使うプロセスであり、

knock[k]!fac[k](); reply[k]?fac[k](y)

によって y = k! がえられる。ただし、CALLCHILD

は次のようなマクロである。

```
mproc CALLCHILD(r) =
begin knocksub[i-1]!fac[i-1](); replysub[i-1]?fac[i-1](r)
end
```

{ この例において fac[0] はいわゆる番兵(sentinel)のような役を果たしていると思われる。同じ例を再帰的な関数でかくと(3.3.4.2)、この部分は他(fac[i:1..limit])と混ざりあってしまう。

Dihybridでは入出力の相手を必ず指定しなければならないが、この制限がないとすれば、fac[i:1..limit]の2つの被護命令はまとめることができる。}

なお、[Ho78]にも階乗を求めるプログラムが示されている。それも「はかけた」プログラムだといえるが、上のプログラムは更にはかけたものということができる。

3.4.2.2 くりかえし命令と選択命令

< repetitive command > ::= do {< guarded command >}$^{+}_{口}$ od

< alternative command > ::= if {< guarded command >}$^{+}_{口}$ fi

< guarded command > ::=
{< range >}$^{\#}$ < guard sequence > → < command sequence >

< guard sequence > ::=
{< Bool expr > ;}$^{\#}$ {< data dec > ;}* < I/O guard >
| < Bool expr >

< I/O guard > ::= < simple input command >
| < simple output command >

[注意] "guarded command" の訳としては「ふた付き命令」というのがあるが、この論文では「被護命令」と訳す。"guard"は「護衛」、"guard sequence"は「護衛列」である。

3.4.2.2.1 護衛の値

護衛列(guard sequence)にあらわれる論理式、データ宣言、入出力護衛(I/O guard)をあわせて護衛という。護衛はtrue, falseのいずれかの値をとる。護衛がデータ宣言のときは常にtrueをとる。入出力護衛の場合、trueをとるのは入出力が成功しうる場合(すなわちその入出力護衛の相手プロセスが、対応する入出力命令の実行にさしかかっている場合)であり、falseをとるのは相手プロセスが入出力護衛の実行前または実行中に停止した場合である(そのどちらでもない場合——相手プロセスが対応する入出力命令以外のところを実行している間は護衛の値は定まらない)。護衛が論理式のときは、その値は論理式の値に一致する。

なお、入出力護衛と対応する入出力命令は入出力護衛であってはならない(命令列の要素としてあらわれる入出力命令でなければならない)。*

3.4.2.2.2 護衛列の値

護衛列（guard sequence）の値は、その要素の護衛の値から、次のようにして定められる。

すべての護衛の値がtrueであるとき、護衛列の値はtrueである。falseをとる護衛が1つでもあると、護衛列の値はfalseである。値の定まらない護衛がある間は護衛列の値もまた定まらない。

3.4.2.2.3 選択命令の意味

選択命令（alternative command）は、それが含む護衛列のうち値がtrueであるものの中から任意の1つを選んで、その護衛列が入出力護衛を含むならその入出力を完了させてから、それに続く命令列を実行する（選択されなかった護衛列が含む入出力護

＊ この禁止事項を設けたのは、入出力護衛対入出力護衛の通信を正しくimplementすることは困難だと考えられるからである。

衛は実行されない。つまり通信はおこなわれない).
すべての護衛列がfalseをとるときの効果は定義されない。

3.4.2.2.4 くりかえし命令の意味

くりかえし命令(repetitive command)は、その要素である護衛列のうち、値trueをとるものの中から任意の1つを選んで、その護衛列が入出力命令を含むならその入出力を完了させてから、それに続く命令列を実行するという操作をくりかえす。(選択されなかった護衛列が含む入出力護衛は実行されない)。
すべての護衛列の値がfalseと

なったとき、この操作は終了する（はじめからこの条件がみたされていれば何もしない。 いいかえれば、くりかえし命令は、その護衛行列がすべてfalseならば何も せず、trueのものがあれば、そのうち1つを選んで、それに続く命令列を実行した後、そのくりかえし命令をはじめから もう一度実行する）。

3.4.2.2.5 範囲つき被護命令

範囲(range)のついた被護命令は、その範囲パラメタの値だけが異なる被護命令の列と等しい。 すなわち、

[i : 1..n] G(i) → CS(i)

は

G(1) → CS(1) □ ・・・ □ G(n) → CS(n)

に等しい。

3.4.2.2.6 例

例1　x, y のうち小さくない方を m に代入すること

(Dijkstra [Di75] による) ;

if $x \geq y \rightarrow m := x$
□ $y \geq x \rightarrow m := y$
fi

この例に関しては 0. ですでに述べた. ■

例2　Euclid の互減法 (?). (Dijkstra [Di75] による)

—— x, y の最大公約数を x および y に代入する.

do $x > y \rightarrow x := x - y$
□ $y > x \rightarrow y := y - x$
od

($x = y$ となったとき、くりかえし命令の実行が

おわる。) ■

例3　Sort：

整数 g[1], g[2], ---, g[n] を g[1] ≦ g[2] ≦ --- ≦ g[n]

となるように並べかえること：

```
do [i: 1..n-1]
   g[i] > g[i+1] → var t: int;
      t := g[i]; g[i] := g[i+1]; g[i+1] := t
od
```

■

なお、「関数呼びだしの意味2」を参照されたい。

3.5 命令列と複合命令(ブロック)

<command sequence> ::= {<CS item>}*

<CS item> ::= <data dec> | <routine dec>
| <command>

<compound command> ::= begin <command sequence> end

命令列(command sequence)の要素は左から順に実行される。begin, end は命令列に対するかっこの役割りをはたす。複合命令内の宣言は複合命令外では有効でないが、複合命令外の宣言は複合命令内で有効である。すなわち、複合命令外の宣言は、複合命令内に自動的に輸入される。なお、複合命令は、この論文でもこれまで使ってきたように、ブロックとよんでもよい。

3.6 データ宣言と標準の型

<data dec> ::= <const dec> | <var dec>

<const dec> ::= const {<const id> = <const expr>}$^{+}_{;}$

<var dec> ::= var {{<var id>}$^{+}_{,}$: <var type>}$^{+}_{;}$

<var type> ::= {[<const subrange type>]}$^{\#}$ <type id>

<type id> ::= <identifier>

定数宣言(const dec)は、定数名を右辺の定数式と同じ意味にする。従って、それ以降宣言された定数名があらわれたときは、それをその定数式に(,)をつけたものとおきかえた場合と意味は等しい(ただし、この仕様書で規定しない細部に関しては、両者の効果は異なっていてもよい)。

変数宣言(var dec)は、変数型(var type)であらわされる型をもつ変数を宣言する。型名として使えるのは次の3つである。*

int : 整数型（値には機械依存の上限、下限がある）

char : 文字型（implementation 依存の文字集合。eol であらわされる「行末文字」を含む†）。

Bool : 論理型（値は true, false のいずれかをとる）

[<const subrange type>] のついた変数型は、それにつづく名前の要素からなる配列であって、その添字の下限が 定範囲型（const subrange type）の中のはじめの定式で、上限が 次の定式で与えられる。**

---

* Pascal [JW75] のような型宣言（type declaration）を導入することによって、多様なデータ構造を定義し、型名を定義できるように言語を拡張することができる。

** 定範囲型については 並列命令の節を参照のこと。

† eol は 外部プロセスとの入出力において特別の意味をもつが、それについては 3.4.1.2.3 を参照されたい。

定数宣言、変数宣言とも、局所的有効範囲則に従う。ただし、最も外側のブロックの外にあらわれる定数宣言だけは、大域的有効範囲規則に従う(3.7参照)。

例

```
const n = 10, T = true , star = '*'
var   i, j : int , c : char
var   array : [1..10] int
```

3.7 マクロ宣言

<macro dec> ::= <macro proc dec> | <macro func dec>
| <const dec>

<macro proc dec> ::=
mproc <proc kind id> {<macro param sequence>}# —
= <compound command> —
<macro func dec> ::=
mfunc <func kind id> {<macro param sequence>}# = <expr> —

<macro param sequence> ::= ({<param id>}+,)

マクロ宣言（macro dec）は、マクロ手続き命令 および マクロ関数呼びだしでよばれる マクロ手続き、マクロ関数 および 大域的定数を宣言する。

マクロ宣言は 大域的有効範囲規則に従う。また、マクロのパラメタは、そのマクロの中だけで有効である（すなわち、局所的有効範囲則に従う）。

例は 3.4.1.3.1 および 3.3.4.1 に示す。また、3.4.2.1 の例も参照されたい。

3.8 ルーティン宣言

<routine dec> ::= <proc dec> | <func dec>

<proc dec> ::=
proc <proc kind id> { <routine param sequence>$^{\#}$ } = <routine body>

<func dec> ::=
func <func kind id> {<routine param sequence>}$^{\#}$
returns <var id> : <func type> = <routine body>

<routine param sequence> ::=
({{ref}$^{\#}$ {<param id>}$^{+}_{,}$: <var type>$^{+}_{;}$)

<routine body> ::= <compound command> | forward

<func type> ::= <type id>

手続き宣言(proc dec)は、手続き命令でよばれる手続きを宣言し、関数宣言(func dec)は、関数呼びだしでよばれる、関数型(func type)であらわされる型の値をかえす関数を宣言する。いずれも局所的有効範囲則に従う。また、ルーティンのパラメタはそのルーティンの中だけで有効である(すなわち、局所的有効範囲則に従う)。

refのついたパラメタは参照パラメタ、ついていないパラメタは値パラメタとよばれる。その意味については3.4.3を参照のこと。

関数宣言の中では参照パラメタに対する代入、参照パラメタを参照実パラメタとして手続きをよぶこと、あるいは参照パラメタを入出力命令の入力パラメタとすることは許されないし、関数宣言の外の変数に関しても同じ制限が加えられる(すなわち、関数の副作用は禁止されている)。また、関数は入出力命令を含んではならない*。

関数宣言において、returnsにつづく変数名は、関数本体の中で変数として使われるが、本体の実行終了直後のこの変数の値が関数の値となる(関数宣言の外では、この名前を参照することはできない)。

---

* この禁止事項を設けたのは、入出力命令を含んだ関数を護衛列の中からよびだすとき、問題が生じうるからである。

ルーティン宣言の本体が"forward"のときは、同名の手続き／関数が同じ命令列の要素として あとで再びあらわれなければ"ならない。その宣言はパラメタをもたないが、その本体では、前の宣言におけるパラメタがあらわれる。前の宣言がおこなわれた以降に この手続き名／関数名があらわれると、2度目に宣言された手続き／関数に対するよびだしとみなされる。ただし、実パラメタは最初の宣言におけるパラメタと結合される。

例　3.3.4.2, 3.4.1.3.2 を参照, されたい ■

例

```
proc mutual recursion2 (i: int) = forward;
proc mutual recursion1 (j: int) =
begin --- mutual recursion2(j-1) ... end;
proc mutual recursion2 =
begin --- mutual recursion1(i-1) ... end
```

■

3.9 プログラム

<program> ::= {<macro dec> ;}* <compound command>

プログラムは コンパイル、実行の単位であり、これ以外の記号列は 正しく コンパイル、実行されない。

3.10 注釈

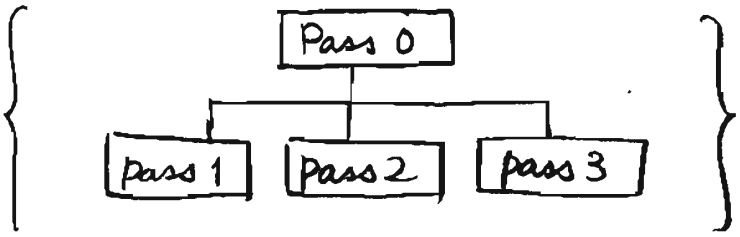
任意個の }を除く記号を { と }の間にはさんだものを注釈といい、名前、符号なし定数の中を除く任意の場所にそう入することができる。* これは言語の要素ではなく、構文および意味に無関係である(付録2参照)。

例

{ comment }

{ 1組のかっこの中の注釈は、
2行以上にわたってもかまわない。 }

{ かっこは 同じ行に
なくてもよい }



■

* 予約語(英字からなる特殊記号)はそれぞれ1つの記号であるから、その中に注釈をかくことはもちろんできない。

文献

[Di75] Dijkstra, E.W.: Guarded command, non-determinancy and formal derivation of programs, Comm ACM18:8 [1975], 453–457

[Di76] Dijkstra, E.W.: A dicipline of programming, Prentice-Hall [1976]

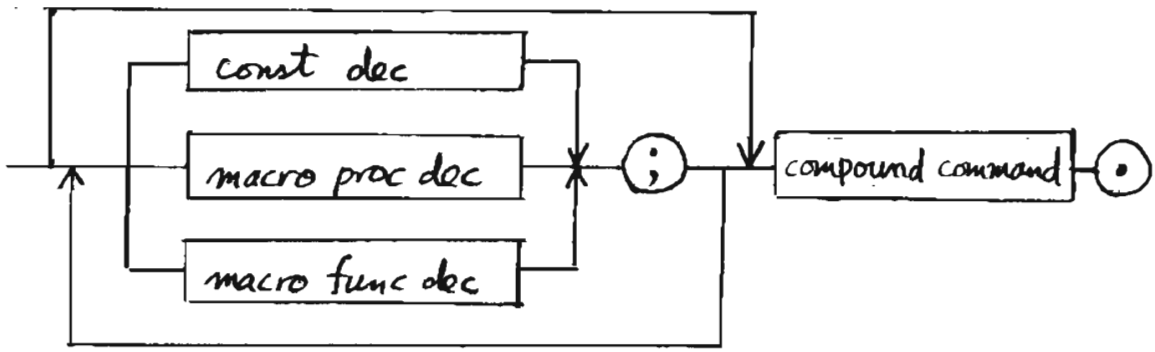
[JW75] Jensen, K. and Wirth, N: Pascal: User manual and Report, Springer [1975]

[KM61] Knuth, D.E., and Merner, J.N.: Algol 60 confidential, Comm. ACM 4 [1961], 268–272

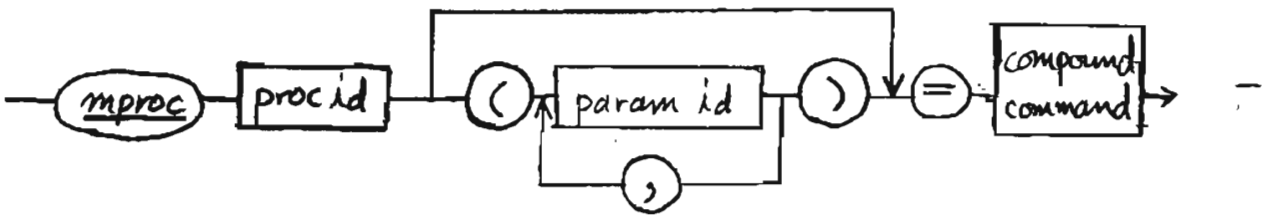
[Ho78] Hoare, C.A.R.: Communicating sequential processes, Comm. ACM 21:8 [1978], 666–677

付録1　Dihybridの構文図 *

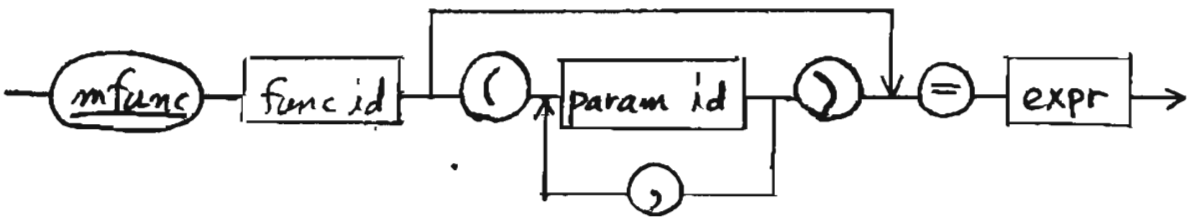
program



macro proc dec

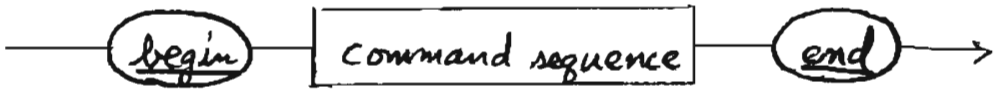


macro func dec

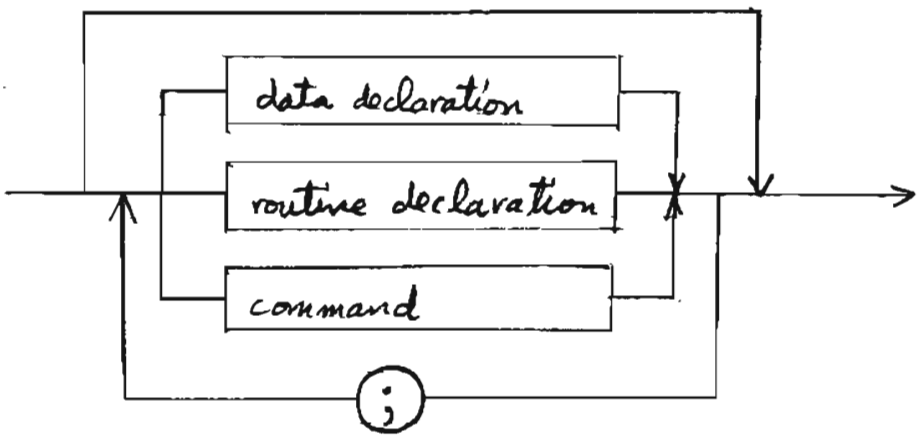


* 構文を top-down に記述する。なお、この記述はformalなものではない。

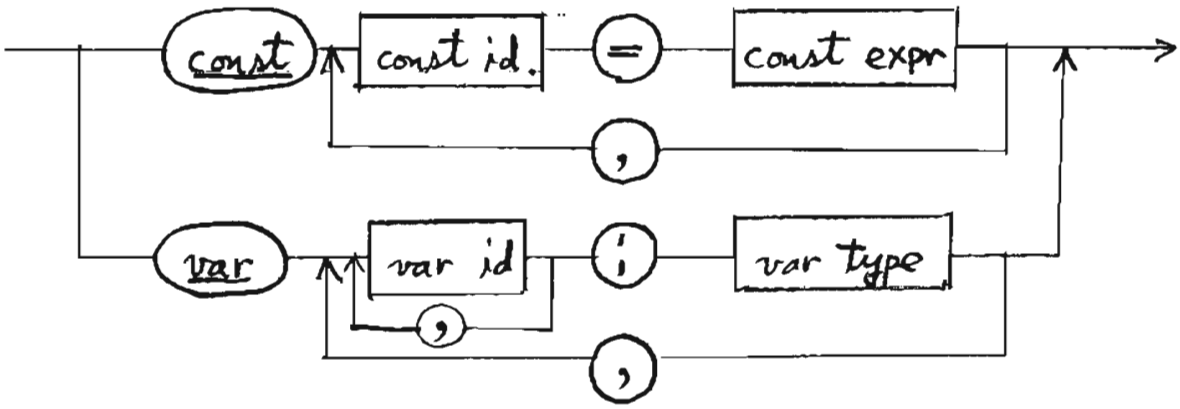
compound command (block)



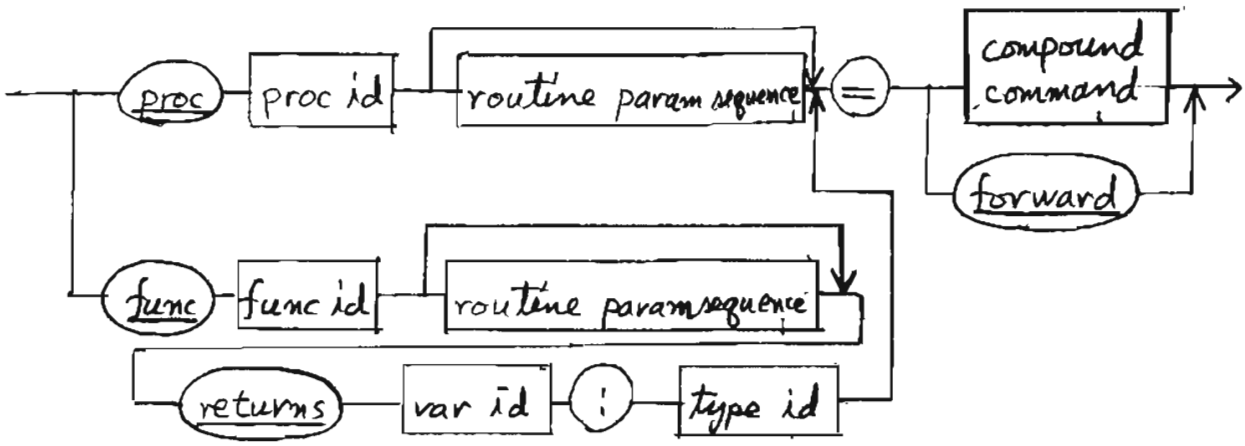
command sequence



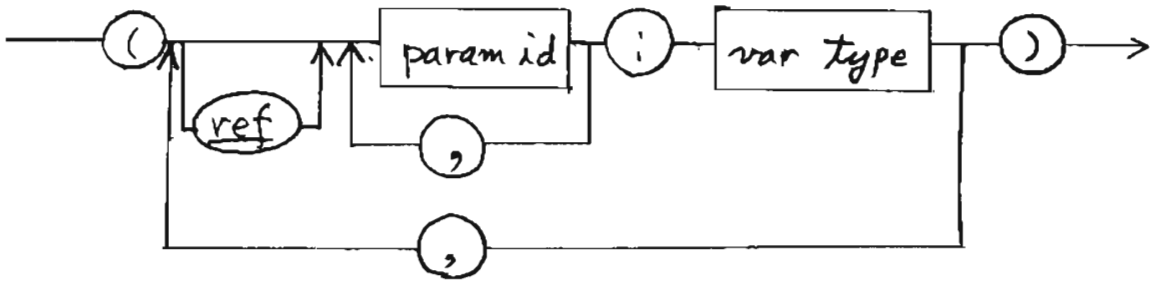
data declaration



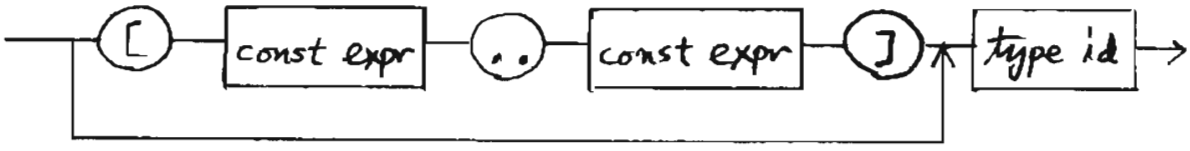
routine declaration



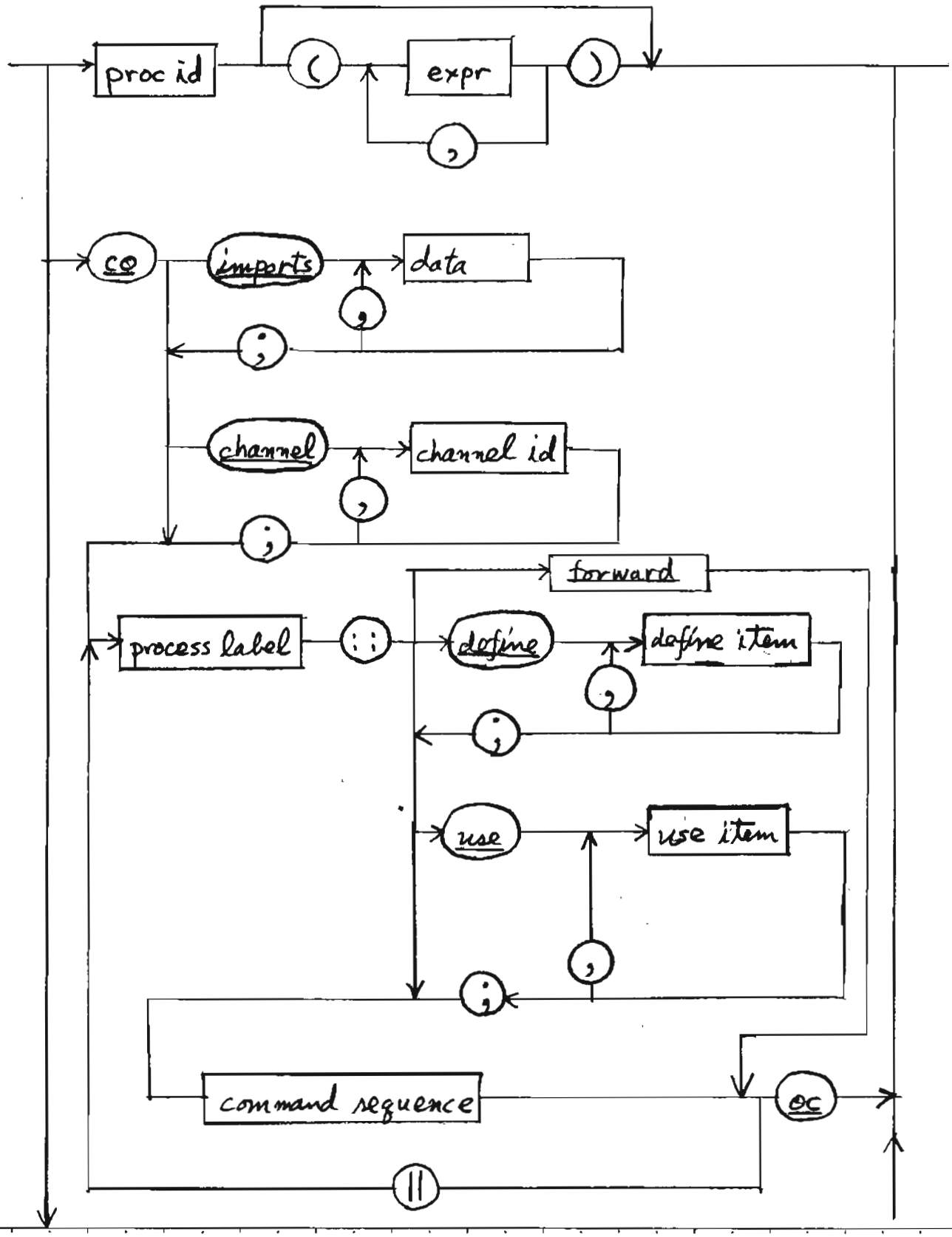
routine param sequence



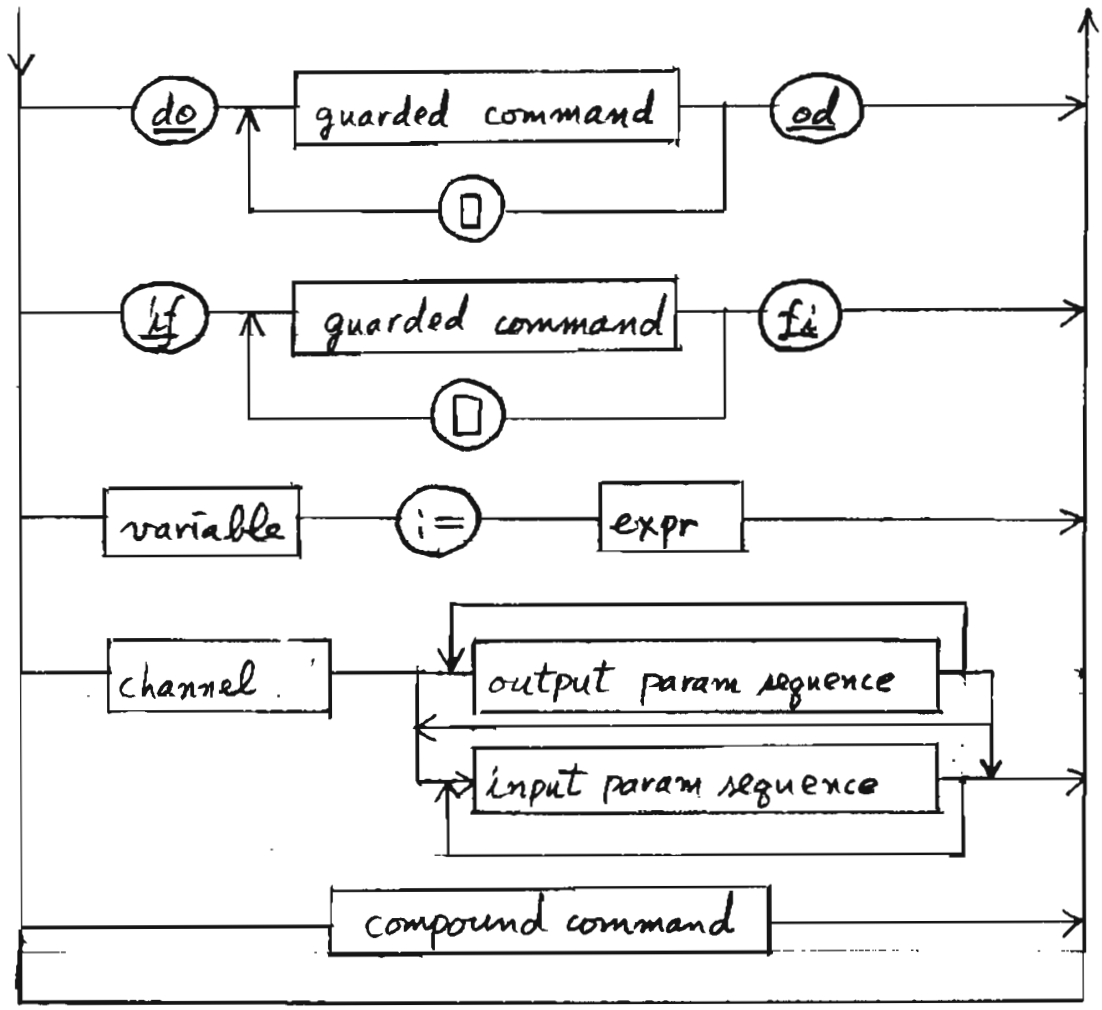
var type



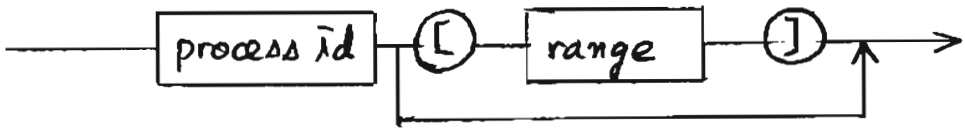
command


proc id
(
expr
)
,
co
imports
data
,
;
channel
channel id
,
;
forward
process label
:
define
define item
,
;
use
use item
,
;
command sequence
oc
||

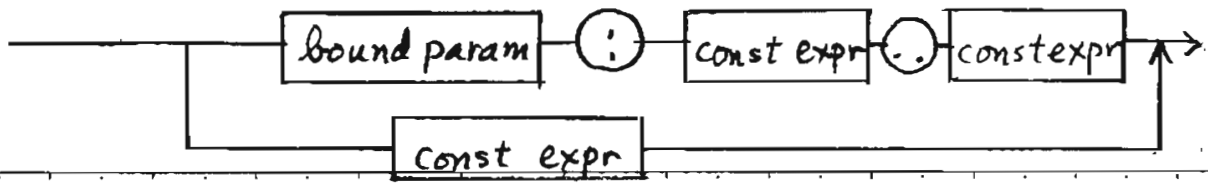
command (つづき)



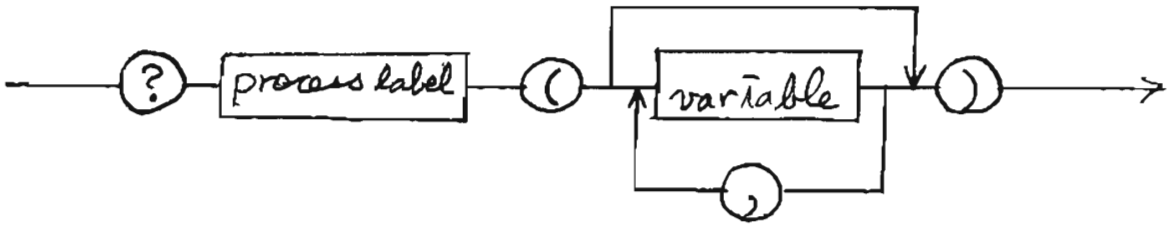
process label



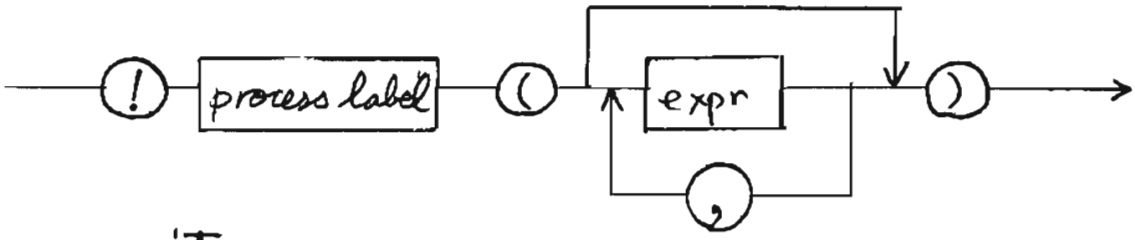
range



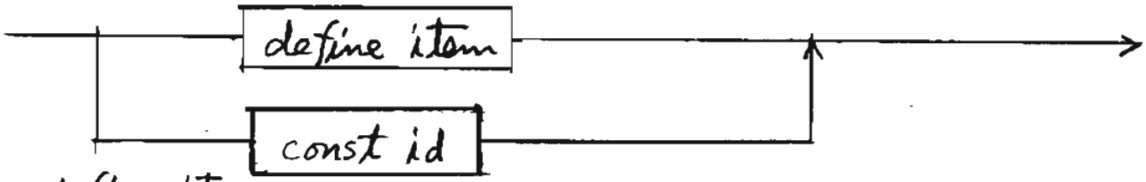
input param sequence



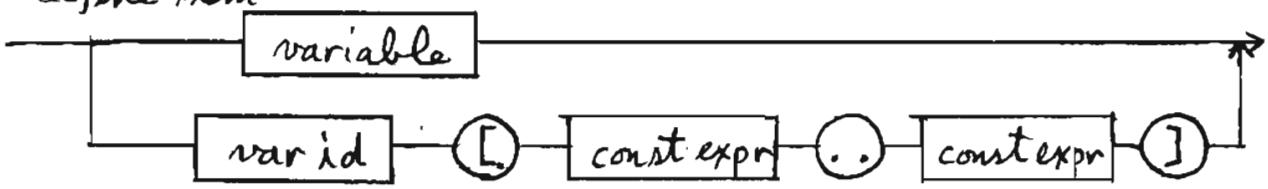
output param sequence



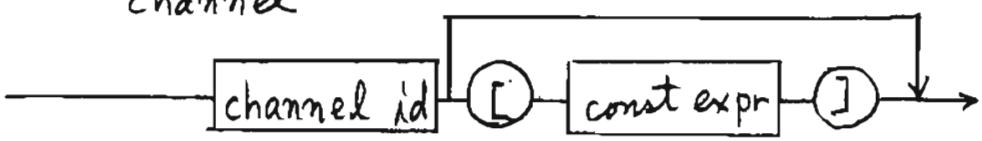
use item



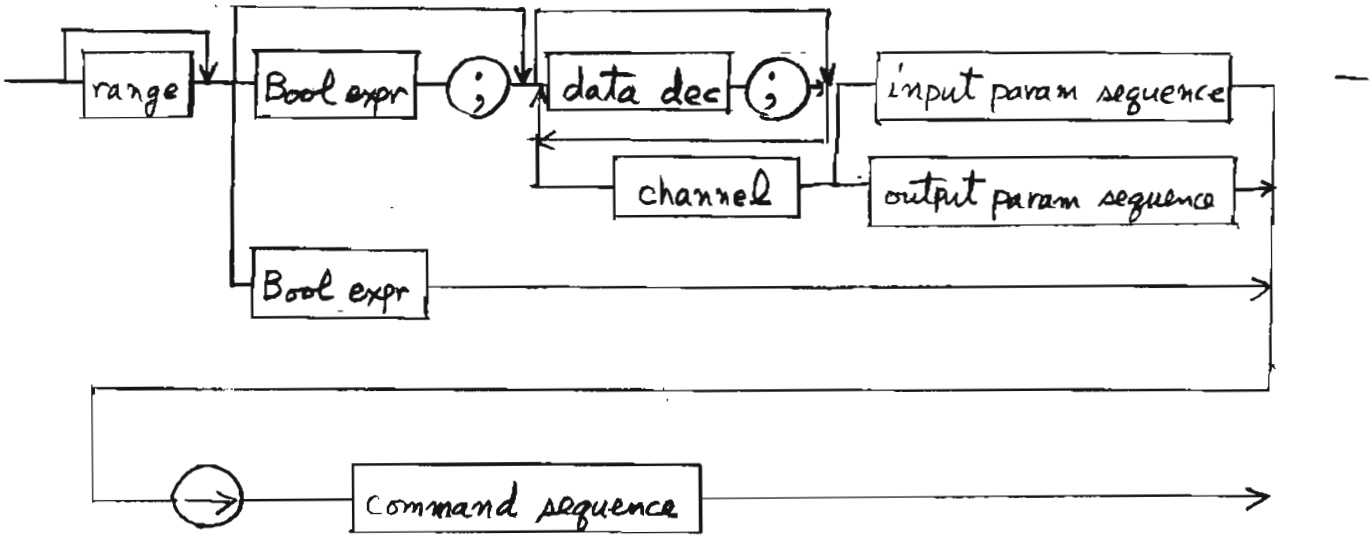
define item



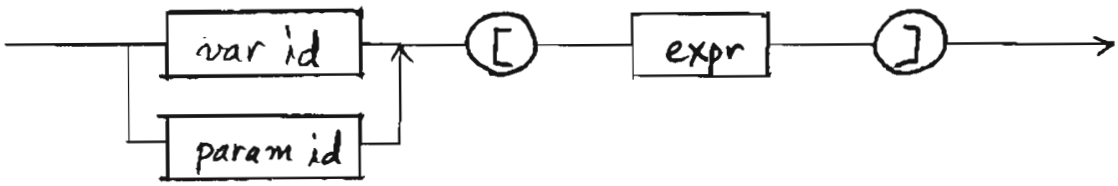
channel



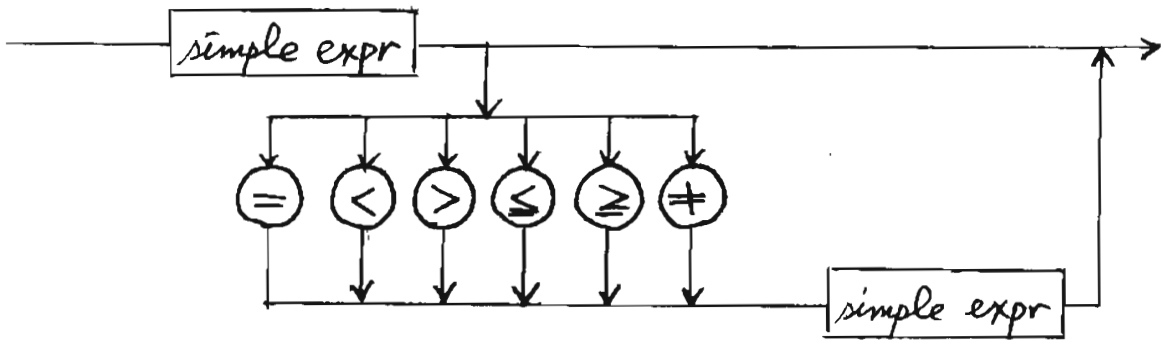
guarded command



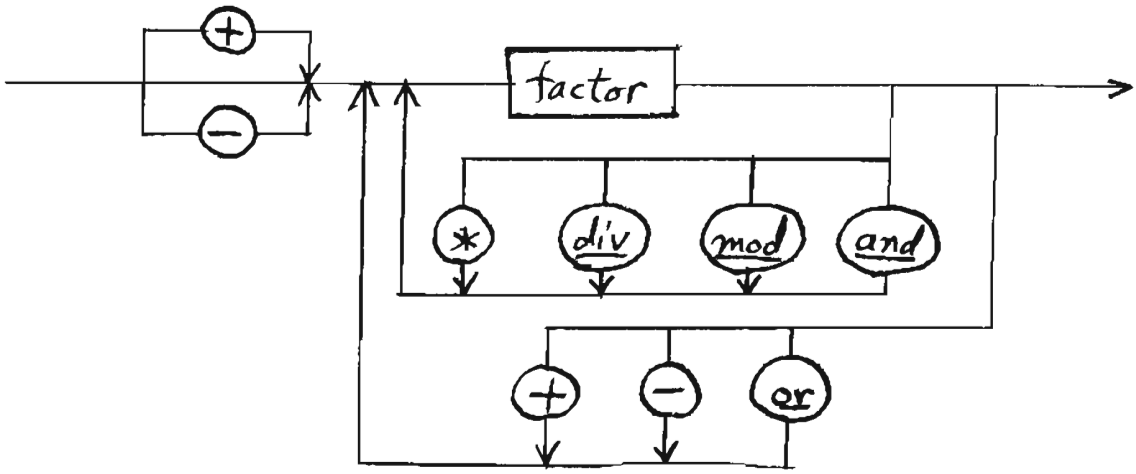
variable



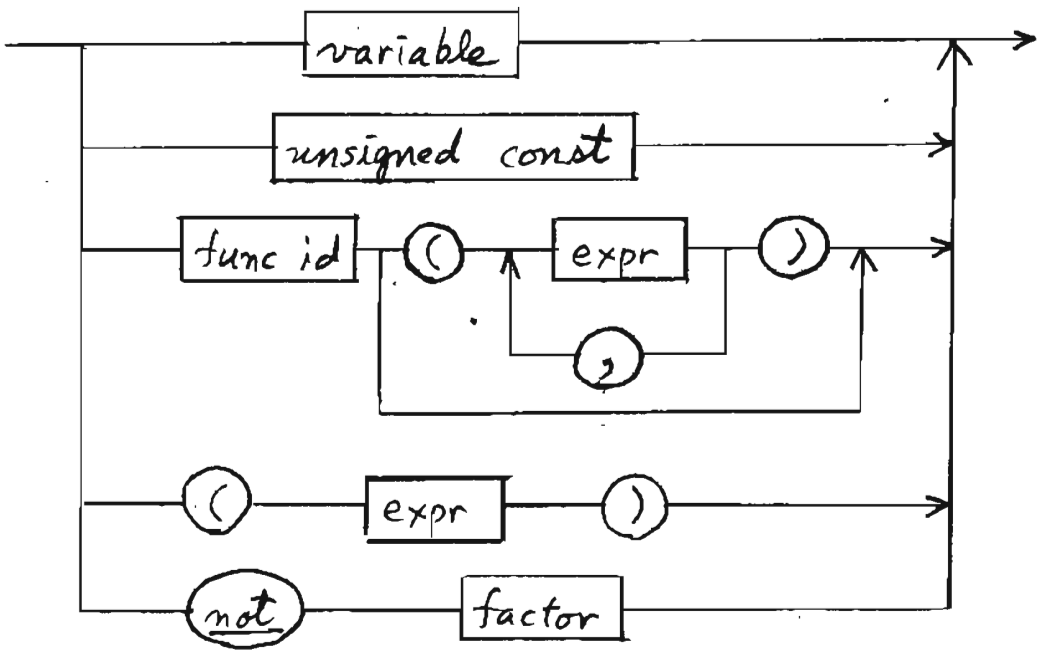
expr



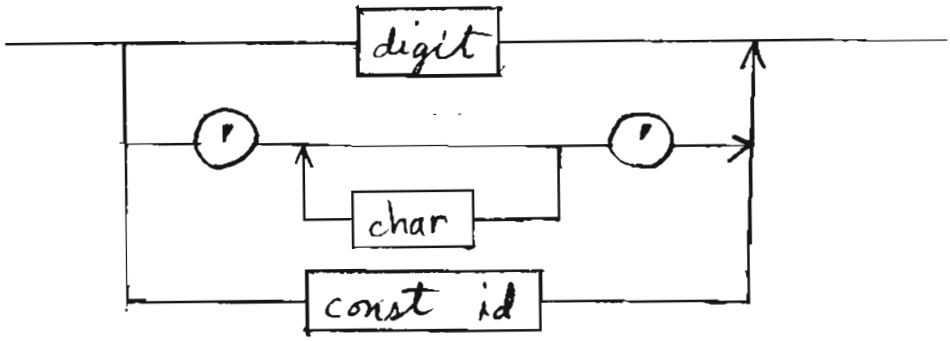
simple expr



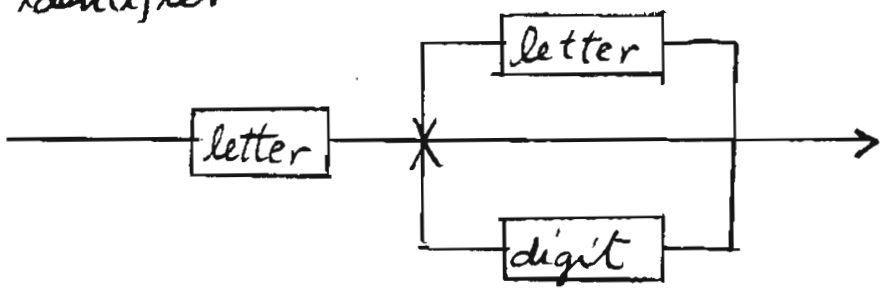
factor



unsigned const



identefier



付録2　金物表現

金物表現においては、小文字と大文字は同一視する。また、小文字は使えなくてもかまわない。また、英字からなる特殊記号は、英数字以外の文字（空白も含む）で前後を区切られた、英字だけからなる列としてあらわされる。*（すなわち、escape character は用いない）。これらの英字列を名前として用いることはできない。また、次の各記号が金物にないときは、それぞれその右に示した文字列によってあらわす。

≦	<=
≧	>=
≠	<>
‖	//
‥	..
≔	:=
→	->
□	#
［	(.
］	.)

基本記号の中には空白を含んではならない。基本記号と基本記号の間には1個以上の任意個の空白を含みうる。基本記号の間の空白は区切り

以外の意味をもたない（区切りとして、少なくとも1個の空白を含まなければならないことがある*）。

* 英字からなる特殊記号および名前は、特殊文字からなる特殊記号とつづけてかく（空白を含めない）ことができる。しかし、英字からなる特殊記号どうし、または英字からなる特殊記号と名前をつづけてかくことはできない。

付録3. 入出力命令のパラメタ列の対応

入出力命令のパラメタの対応則については「入出力命令の意味」のところで述べたが、まずそれを再び記そう。

- パラメタの対応則

(1) パラメタ列の対応

任意の入出力命令の任意の出力パラメタ列について、次の条件がなりたつ：その入出力命令が属するプロセスをp、その出力パラメタ列が指定する相手プロセスをqとすると、qにあらわれる同じ路を用いるすべての入出力命令は相手プロセスをpとする入力パラメタ列をもっていること。

(2) パラメタ列内のパラメタの対応

これらのパラメタ列にあらわれる変数と式の数は等しく、左から順に対応づけると、対応する変数と式の型は等しい。

(1)は「入力」と「出力」を交換しても真である。■

パラメタ列の対応則から、次の定理が導かれる。

定理1

Cを通して通信する任意の2つのプロセス p, q 内のすべての入出力命令について、入出力パラメタの対応則がなりたっていれば、

(1)(2) pにあらわれる、Cを用いるある入出力命令が、qを相手プロセスとする入力パラメタ列、出力パラメタ列のうち入力(出力)パラメタ列だけをもつとき、pにあらわれる、Cを用いる他の入出力命令はやはりqを相手プロセスとする入力(出力)パラメタ列だけをもち、qにあらわれる、Cを用いるすべての入出力命令は、pを相手プロセスとする出力(入力)パラメタ列だけをもつ。

(3) pにあらわれるCを用いるある入出力命令が、qを相手プロセスとする入力パラメタ列、出力パラメタ列の両方をもつとき、pにあらわれるCを用いる他の入出力命令はやはり、qを相手プロセスとする入力パラメタ列と出力パラメタ列をもち、qにあらわれる、Cを用いる

すべての入出力命令は、pを相手プロセスとする入力パラメタ列と出力パラメタ列をもつ。

また、(1),(2),(3)の条件がみたされれば、入出力パラメタ列の対応則がなりたつ。

[証明]

(1) pにあらわれる、cを用いるある入出力命令Cが、qを相手(=相手プロセス)とする入力パラメタ列をもつとき、qにあらわれる、cを用いるすべての入出力命令は、pを相手とする出力パラメタ列をもたなければならない。それから逆に、pにあらわれる、cを用いるすべての入出力命令は、qを相手とする入力パラメタ列をもっていなければならない。

もしCがqを相手とする出力パラメタ列をもたず、qにあらわれる、cを用いる入出力命令がPを相手とする入力パラメタ列をもっているとすれば矛盾がおこるから、qにあらわれるcを用いるすべての入出力命令は、pを相手とする入力パラメタ列をもたない。同様に、pにあら

われる、cを用いるすべての入出力命令が、Sを相手とする出力パラメタ列をもたないこともいえる。

(2), (3) は (1) と同様.

逆は明らかである. ■

定義; 入出力命令の同型

1つのプロセスの中にあらわれる2つの入出力命令は、その一方のすべての入力パラメタ列について、それと同じ相手プロセスをもつ入力パラメタ列を、もう一方の入出力命令がもち、かつ前者のすべての出力パラメタ列について、それと同じ相手プロセスをもつ出力パラメタ列を、後者がもっているとき同型であるという.* ■

定理2

1つのプロセスの中にあらわれる、同じ路を用いるすべての入出力命令は同型でなければならない.

[証明]

定理1から容易に導かれる. ■

定理1, 定理2に関する例は「入出力命令の意味」で 例・ としてあげた。

＊ たとえば、1つのプロセスの中に

c!q(v)!r(w) と c!r(x)!q(y) があらわれるとき、この2つの入出力命令は同型である。

第2部

Dihybridの設計

Dihybridの設計

第2部では、新しいプログラム言語を設計した理由は何かということを述べる。また、なぜ入出力命令、並列命令、被護命令(guarded command)(これらについては[第1部]を参照されたい)などをとりいれる必要があったのかという点、および、より細かい設計上の決定を下した理由に関して述べる。そして、Dihybridの設計の基になったHoareの言語案[Ho78]とのちがいについても述べる。

1. 新たな言語を設計した理由

Dihybrid は M³-system ([Hi78], [Wo78], [Og78], [Om78]) に implement することを前提に設計された。M³-system は、micro computer TM-990 を5台使用した、共有メモリのない multicomputer system である。このようなシステムに従来の言語をそのまま implement するのは困難である。従来の言語で書かれたプログラムを効率よく実行するには、データの流れの解析や、実行時間の見積りをおこない、その結果に基いて手続きや変数を各処理装置に割りわけ、各処理装置がいつどういう方法で通信するかをすべてコンパイラが決めることが必要であろう。それでも充分な効果があげられるとは考えられない。このような努力を払ってまで、従来のプログラミングのやりかたを踏襲する必要があるだろうか。従来のプログラミングは、完全に逐次的な機械によってつちかわれてきたために、過度に逐次的になっていたといえる。このような反省の上にたつならば、この困難を、従来の言語につぎをあてて乗り切ろうとするべきではなく、新たな言語を作るのが適切と考えられ、並列処理を記述することができるようにし

る。*

---

* 従来の言語につき"をあてて並列処理を記述しようとした例はいくつかある([Ku68]など")。新たな言語を覚えるよりは、つき"をあてられた従来の言語を使いたくなるのは人情であろうが、いつまでもこの状態にとど"まっていては進歩を遅らせるであろう。

2. 2つの言語案

Hoare [Ho78] と Brinch Hansen [Br78] は、あいついで M³-system のようなタイプの計算機システムに合わせた言語案を示した。(これらは、新たな言語を設計する場合に重要な道標となるものである。) Hoare は並列化と、同期をとるための手段として、並列文 [Di68] と message buffer [Br73] の一種を用い、Brinch Hansen は monitor [Ho74] を用いている (これらに関しては 4. で簡単に述べる)。いずれも、操作システムやその他の実時間プログラムを書く場合に必要になる非決定性を導入するために guarded command [Di75] を使っている (guarded command については 5. で述べる)。

Dihybrid の設計時には、このうち [Ho78] だけを参照することができた。他に目的に充分かなった言語をみいだせなかったので、Hoare の言語案を基にして、新しい言語を設計した。従って、それとの比較を中心にして Dihybrid の設計に関して述べるが、[Br78] とも比較してみることにする。

3. 並列処理に関する設計方針

並列処理を容易に記述できるためには、設計すべき言語は次のような条件をみたすべきだと考えられる。

1) プログラマを処理装置のわりつけから解放する:

プログラミングは、実際の処理装置がどういう構成になっているかなどということにはわずらわされずに、論理的なレベルでおこなわれるべきである。プログラマが自ら処理装置のわりつけをしなければならないとすると、次のような問題が生じる。

- データ、ハードウェアのいずれが変化しても、プログラムを大幅にかきかえなければならない(データの変化の例としては配列の大きさを変えること、ハードウェアの変化の例としては処理装置の数が変わることなどはしばしばおこるであろう)。このことは、システムの安定性を著しく悪化させる。
- アルゴリズムにとって本質的でない点、とくに、ハードウェア構成にプログラマが注意を払わなければ

ならない。プロセスの同期化という論理的な問題は、処理装置の同期化という問題にすりかえられてしまう。そのために、アルゴリズムの論理的構造がぼかされたり、ねじまげられたりされて、プログラマはそれから遠ざけられてしまう。その結果、プログラムは書きにくくなり、また読みにくくなる。■

共有メモリのないシステムにおいて、処理装置を実行時にプロセスにわりあてることは困難である。また、それができたとしても、実行時に操作システムのようなおおがかりなソフトウェアが必要であろう。従って、コンパイル時にわりあてをしてしまうのを基本とする。コンパイル時にわりあてをおこなう場合、コンパイラはその分複雑になるが、全体としては、またとくに実行時のソフトウェア（モニタ）やハードウェアは比較的簡単なものでもすむと考えられる。コンパイル時のわりあてが可能であるためには、言語じたいがそのように設計されなければならない（たとえば、プロセスの数はコンパイル時に決まらなければならない）。

2) プロセスの同期化とデータ交換のための機能に関して:

まず、プロセスの同期化とデータ交換のための充分な機能をもっていなければならない。そして、それは共有メモリのないシステムにimplementしやすいものでなければならない。この点に関しては次節で更に述べる。

3) 自由な通信ができること.

M^3-systemのハードウェアの思想は、通信は全処理装置の間でいっせいにおこなうというものである。そのため、当初はその思想に従った言語を作ろうと考えた。しかし、これはアルゴリズムの記述の上からは不便なことが多い。もっと柔軟な通信(たとえば2つのプロセスの間だけで、他のプロセスをわずらわせない通信)ができるべきである。

自由な通信という中には次のような通信がある.

① 互いに相手プロセスを指定しての1対1(または多対多)の通信.

② いくつかの指定されたプロセスのうちの1つ(またはいくつか)との通信.(通常必要になるのは、最もはやく通信を

望んだ"ものとの通信である)

③ 相手を指定しない通信 (1対1 または多対多). ■

このうち ③ は、Dihybrid では Hoare [Ho78] にならって認めていない。しかし ① ~ ② はとりいれられており、多くの場合は ① ~ ② でまにあう。* このように決定したのは、相手プロセスが走る処理装置を特定できる方が implement しやすいからである。

② は非決定性の導入を要求する。

この点に関しては 5. で述べる。

* [Br78] では逆に ② のみが可能である。

4. プロセスの同期化とデータ交換のための手段に関して

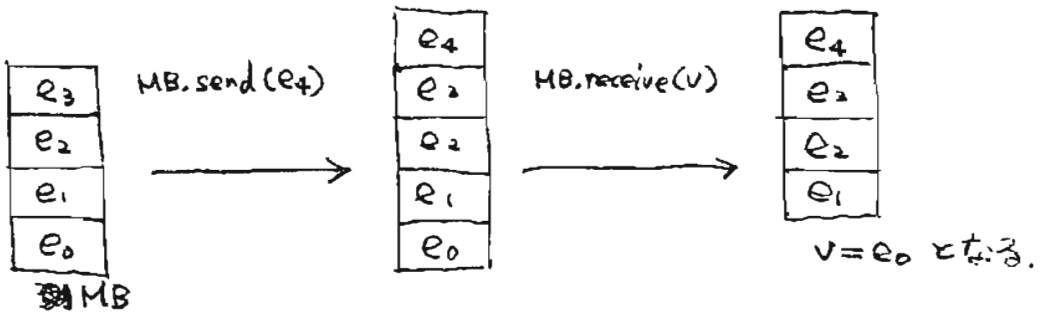
プロセスの同期化とデータ交換のためにはいくつかの方法がある。それらを比較検討することによって、Hoareの言語案およびDijkstraに入出力命令と並列命令がとりいれられたことの意義を論ずることにしよう。

4.1. 同期化の手段 —1
—— とくにメッセージ・バッファについて

共有変数が唯一の手段であってもプロセスの同期化を達することはできる[Di68]し、セマフォ([Di68], [Ha72])はよく使われている。また、臨界領域[Br73]や条件付臨界領域[Br73]、それを発展させたguarded region[BS78](これについてはS.で別の観点から述べる)、あるいはイベント待ち行列[Br73]などという手段がある。しかし、主な目的がデータ交換である場合には、メッセージ・バッファ*が適している。つまり、データ交換の手続きを最も単純明解に表現することが

＊ メッセージ・バッファについて簡単に解説することにしよう。メッセージ・バッファは、同じ型の要素（変数）からなる列（その長さの最大値が定められている）である。メッセージ・バッファMBに対しては2つの手続きが定義される。手続きMB.send(e)は式eの値をその列の一番上に加える。もしMBがすでに満（列の長さが最大値に達している）ならば、次に述べるMB.receiveが実行開始されるまで待つ。手続きMB.receive(v)は、バッファMBの一番下の要素をとりだして、その値をvに代入する。もしバッファが空ならば、MB.sendが実行開始されるまで待つ。待ちが生じたあとはMB.sendとMB.receiveは同時に実行されるので、列の長さはその実行中も変化しない。

更にくわしくは[Br73]などを参照されたい。



できる。

メッセージ・バッファを手段として選ぶことは、他の理由によっても支持される。共有メモリのないシステムへのimplementationを考えているのであるから、共有変数を使う方法は適当でない*が、(条件付)臨界領域は、その中では共有変数を1つのプロセスが排他的にアクセスできるようにするものだし、セマフォが様々の目的で用いられうるのは共有変数との組みあわせられたときである。これに対し、メッセージ・バッファは共有変数の助けを必要としない。また、イベント待ち行列は、プロセスのスケジュリングという特別の目的に用いられるものであるから、我々の目的には ふさわしくない。

* ただし、4.2で述べ [Br78]の方法のように、処理装置間の同期化には別の方法を使うものとして、1つの処理装置上にある複数のプロセスの同期をとるために共有変数を使うことはできる。

また、メッセージ・バッファは、セマフォなどより安全である。なぜなら、セマフォを用いる場合、プロセスの相互排斥はプログラマの責任でおこなわれるが、メッセージ・バッファの場合、これは自動的に達成される(言語のしくみとしてその機能がそなわっている)からである。

だが、メッセージ・バッファの採用は、Hoareの言語案およびDiyhybridにとって、もっと重要な意味をもっている。メッセージ・バッファを用いると、subroutineやcoroutineを模倣することができる[Ho78]。さらに、プロセス配列と組みあわせることによって(深さが配列の大きさでおさえられるが)再帰的手続きを、また、guarded commandと組みあわせることによって、SIMULA67([Da72],[Da67])のclassや、monitor[Ho74]を模倣することができる[Ho78]。これらのことがDiyhybridの汎用性を高めているのである。

Date

メッセージ・バッファによる方法は、さらに

① バッファリングをする方法：**

この方法では、複数のメッセージをバッファにたくわえることができる。理想化すれば、バッファは任意個のメッセージを含むことができ、通信のためにバッファが空くのを待つ必要がないということになる。

② バッファリングをしない方法：†

この場合、通信をしたいとき、相手が通信状態にはいるまで待たなければならない。*

の2つに分けられる。

Hoareは、①を選ばなかった理由を、(1)分散型のmultiprocessorsにimplementするのはあまり現実的でないこと、(2)バッファリングをするメッセージ・バッファは、バッファリングをしないそれを用いて書くことができること、

---

* この場合、実はバッファをもっていないのだから、これをメッセージ・バッファとよぶのは適当でないかもしれない。

** P.124.5の脚注において、

$1 \leq$ 列の長さの最大値 $< \infty$ の場合である

(理想化された場合は、列の長さの最大値 $= \infty$ である)。

† P.124.5の脚注において 列の長さの最大値 $= 0$ の

場合である。

をあげている[Ho78]。②はM³-systemのハードウェア
にも適した方法だと考えられる。*

4.2. 同期化の手段 —2
Distributed Processes

同期化のために~~もっと手の~~こんだ方法も考えられた。
Brinch Hansen [Br78] は、monitorの概念[Ho74]を基に
した(従って Concurrent Pascal [Br75C][Br75P](monitor, class の)や Modula
[Wi76](module の)に近い)、共有メモリのないmulticomputer system
向きの言語概念 Distributed Processes を示しているが、これ
では、(異なった処理装置上にある)プロセスの同期
とデータ交換のために、相手プロセスに属する手続きを起
動するという方法をとっている。たとえば、次のようなプロ
セスがあるとする。

```
process name
  x: own variable;
  proc p; begin ... end;
  initial statement
```

* M³のハードウェアについては[第3部]でかんたんに述べる。

ここで、xをアクセスできるのは手続きPとinitial statementだけであり、プロセスnameの外からは直接アクセスすることはできない。手続きPは、他の(Pをもっている処理装置とは別の処理装置上で走る)プロセスから起動することができる。手続きは値パラメタと結果パラメタ(result parameter)**をもちうる。Pを起動したプロセスはPの実行が終わるまで待たされる。

この方法では各処理装置上に、他の処理装置上のプロセスに対応する「無名のプロセス(anonymous processes)」があって、他の処理装置上のプロセスからの手続きよびだしは、これらのプロセスによって処理される。* これらの無名のプロセスと、同じ処理装置上の名前をもつプロセスは、変数を共有することができる。従って、セマフォや臨界領域を用いることができるが、Brinch Hansenはこれらのプロセスの同期化のために guarded regionを用いている。

* 必ずしもこのようにimplementされなければならないわけではないが。

※ 値パラメタは、周知のように手続き（または関数）の実行を開始するときに、その実パラメタの値を仮パラメタに代入するに従って、その手続きの実行後、実パラメタの中にあらわれた変数の値は変化していない）だが、結果パラメタは、これとは逆に手続き（または関数）の実行が終了するときに、その仮パラメタの値を実パラメタ（＝変数）に代入するものである。手続きの実行前には結果仮パラメタの値は定義されておらず、実行後の実パラメタの値は実行前および実行中のときのそれに無関係である。

4.3. プロセスの分流・合流のための手段

4.1～4.2で述べたのは狭い意味の同期化の手段だが、「同期化は、操作を時間的に順序づけるための制約全般に対することばである。」[Br73, 日本語版] という定義からすれば、プロセスの分流・合流（あるいは生成・消滅）を指定することも同期化の一種である。

データ交換という観点からみるならば、データ交換が必要なときにはプロセスが合流し、そうでないときは分流するのがこの方法である。表現法はいくつかある。

① 並列文 [Di68]

並列文とは次のようなものである。

parbegin S_1; ... ; S_n parend.

parbegin でプロセスは n 個にわかれて $S_1, \cdots, S_n$ を並列に実行した後、parend で1つに集まる。特殊な形として

for i := m to n do S oc がある。これは、$i = m, m+1, \cdots, n$ について S を共時に実行することをあらわす（m, n は定数――すなわちコンパイル時に決まる）。

② Fork, Join [Co63]

手続きFork(place)はプロセスを2つ(またはそれ以上)に分け、そのうちの1つはそのFork呼びだしの次を実行し、残りは指定された場所(place)から実行される。これらは手続きJoinで再び1つにまとまる。

この方法はPL/Iなどにとりいれられている。

③ actor model ([De75], [Ru77])

actor modelにおいては、プロセスという概念はなくなっているといえるが、②を発展させたものと考えることができる。

このモデルではプログラムはactorとよばれる要素をlinkでつなぎあわせたものとしてあらわされる(図1)。actorはinput linksとoutput linksをもち、そのすべてのinput linksにデータが到着すると、output linksのいずれにもデータが存在しな

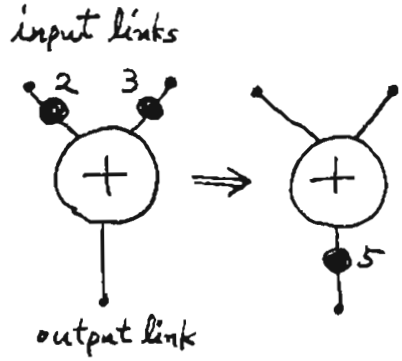


図1 actor

ければ、入力データをとりいれて output links に出力データを出す。1つの link には1つのデータ(構造のある場合もある)しか存在を許されない。この原理によって、同期をとりながら共時的な計算をすることができる。

このモデルのもとでは、同期とか相互排斥といった問題をとりたてて考える必要がなく、安全であるが、このモデルに従ったプログラムをどのように表現するか、どうimplementするかという問題が満足に解決されていない。

これら3つの手段のうち、②はやや危険な方法であり、③はimplementationが難しい。①は安全であり、目的にかなっていると考えられる。

4.4. どれをとりいれるべきか.

これまで述べてきた広義の同期化の手段のうち、4.1～4.2で述べたものは、プログラムの実行中ずっと特定のプロセス群が存在しつづけ(プロセスの生成、消滅がない)というモデルに合ったものであり、4.3で述べたものは一アルゴリズムの各段階にあわせてプロセスが分流・合流する結果、それらのプロセス群を巨視的にみれば"1つのプロセスのようにみえる"というようなモデルに合ったものだと考えられる。時と場合により この2つを使いわけることができればより便利であろう。

たとえば、操作システム(の場合)は様々のスケジュラがその生涯にわたって存在しつづけ、不変の目的をめざして主体的に処理をおこなっていくから、第1のモデルをあてはめるのが妥当と考えられる。一方、たとえば"数値計算のアルゴリズムを並列化したものを記述する場合には、並列化のしかたに対する要求が段階によってかわり、また、全体として1つのアルゴリズムをあらわしているの

だから、そのことがよくわかるようにかくべきだが、そのためには第2のモデルをあてはめるべきだろう。

両者をうまく言語設計の中にとりいれることができるだろうか。4.2で述べたdistributed processesは操作システムの記述というような目的には最適と考えられるが、並列文とはうまくなじまないように思われる。そう思う理由は(1) monitorのいかめしさはこれを「文」の一部としてとりいれることを拒否する。(2) distributed processesで使われているguarded regionは、いったんこれにはいるとぬけだせなくなる。[5.3参照]従ってそれを含む並列文からぬけだせなくなるということである。いずれも解決不可能だとは思わないが、これらの問題を解決して、なおもとのelegance を維持できるかどうかは疑問である。

並列文となじませることが難しいとなると、distributed processesは上記の観点から見た場合不利である。メッセージ、バッファと並列文はなじませることができる。このことはHoare [Ho78]によってはっきり示された。

Dihybrid における"並列文"(すなわち 並列命令)と、"メッセージ・バッファ"(すなわち 入出力命令)は、このHoareの言語案にほとんど全面的に依存することにした。

5. 非決定性の必要性と guarded command

5.1. 非決定性の必要性

入出力命令は指定したプロセスとの入出力をおこなう。従って、全く相手のわからない通信は不可能であるが、いくつかの指定されたプロセスのうちの1つとの通信ができないと、かなり記述できるアルゴリズムのクラスを せばめることになる。

たとえば n 台の端末があって、それを使ってユーザが中央の計算機にコマンドを送るという場合を考えてみるとよい。あるユーザは10秒間に1回の割合でコマンドを送ってくるかもしれないし、別のユーザは何時間もコマンドを送ってこないかもしれない。次にサービスすべきユーザを決めてしまうと、そのユーザがコマンドを送ってくるまで、他のユーザはコマンドを送れないし、計算機は遊ぶことになる。これをさけるには、最初にコマンドを送ってきたユーザにまずサービスするようにすればよい（厳密に到着順にするのは難しいし、またその必要性もないが）。

今の例は、1つの内部プロセスと多数の外部プロセスか

らなる例だったが、ユーザ(端末)という外部プロセスのかわりに内部プロセスをおいてもよい。このように、非決定性は実時間プログラミングには不可欠なものであるが、それをelegantに表現する手段がguarded commandである。*

ただし、M³-systemの目的からすると、このような非決定論的なアルゴリズムを記述する必要はあまり生じないかもしれない。

5.2. Dijkstra の guarded command

guarded commandを発明したのはDijkstraである。この節ではDijkstraのguarded command [Di75]について、簡単に説明する。

* ただし、Dijkstraがguarded commandを考えだした目的は、このような、同期化とからんだ問題をとくためではなくて、「決定する必要のないことは決定しない」ためといえるであろう。[Di75].

Dijkstraに従うと、guarded command に関する構文は次のとおりである（ {...}* は ... の0回以上のくりかえしをあらわす）。

<guarded command> ::= <guard> → <guarded list>

<guard> ::= <boolean expression>

<guarded list> ::= <statement> {; <statement>}*

<guarded command set> ::= <guarded command> {□ <guarded command>}*

<alternative construct> ::= if <guarded command set> fi

<repetitive construct> ::= do <guarded command set> od

<statement> ::= <alternative construct> | <repetitive construct>

| "other statements"

選択構文（alternative construct）は、それに含まれるguardのうち値が真である任意の1つにつづくguarded listを逐次的に実行する。値が真のguardが存在しないときは実行を"放棄"する。

たとえば、

if $x \geq y \rightarrow m := x$

□ $y \geq x \rightarrow m := y$

fi

([Dij75]による) は、 $x > y$ のときは最初の guarded command が選択されて $m := x$ が実行され、 $y > x$ のときは2番目の guarded command が選択されて、$m := y$ が実行される。$x = y$ のときは2つの guarded command のうち任意のひとつが選ばれるが、どちらの guarded list を実行しても結果は変わらない。結局、いずれの場合も x, y のうち小さくない方の値が m に代入されることになる。

くりかえし構文 (repetitive construct) は、もし、すべての guard が偽ならただちに終了し、そうでなければ値が真である任意の guard につづく guarded list を逐次的に実行した後、再びそのくりかえし構文をはじめから実行する。すなわち、すべての guard が偽となるまでくりかえし実行する。

たとえば、

do $x>y \rightarrow x := x-y$

□ $y>x \rightarrow y := y-x$

od

(〔Di75〕による) は、x, y の初期値を X, Y とするとき、もし $X>Y>0$ であれば $x \leq y$ となるまで x から y をひきつづけるから、その結果 $x=y$ または $0<x=X \bmod y<y$ となる。$x=y$ となると実行は終了するが、これは、$X \bmod y=0$ のときである。$x=X \bmod y$ となると、上と同様のこと (x と y が逆になる) がくりかえされ、結局 Euclid の互除法と同じことがおこなわれて、最後には x, y がともに X, Y の最大公約数となる。

5.3. guarded region

Dijkstra の guarded command は、プログラム言語に非決定性を elegant な形で導入したが、5.1. の注釈で述べたように、Dijkstra の目的は同期化の問題をとくことではなかったので、5.1. で述べたような問題をとくには、そ

それでは不充分である。guarded command の elegance をそのまま受けつぎ、これを同期化の問題をとくのに適するように変更を加えたのが guarded region [BS78] である。

guarded region は、guarded command の if, fi, do, od のかわりに別の記号を用いるほかは、guarded command と同じ形をしている。*

if-statement (= alternative construct) に対応するものとして when-statement、do-statement (= repetitive construct) に対応するものとして cycle-statement がある。

when-statement は、その、値が真である guard のうち任意の1つを選んで、それにつづく guarded list を実行する点で if-statement と同じであるが、すべての

---

* Brinch Hansen は、

if ··· → ··· □ ··· fi とかくかわりに if ··· ; ··· | ··· end、

do ··· → ··· □ ··· od とかくかわりに do ··· ; ··· | ··· end

とかいている。when-statement は when ··· ; ··· | ··· end、

cycle-statement は cycle ··· ; ··· | ··· end という形である。

guardの値が偽でも実行を"放棄"せず、いずれかが真になるまで待つ。それらのguardsに含まれる変数を、(同じ処理装置上にある)他のプロセスと共有し、他のプロセスがその値を変更するのを待つわけである。

cycle-statementの場合も、すべてのguardsの値が偽になっても、いずれかが真になるまで待つから、cycle-statementの実行は終わりがないことになる。

cycle-statementを用いると、5.1の問題をとくことができる。user i.sent ($1 \leq i \leq n$) をユーザiがコマンドを送ってきているかどうかをあらわす変数とすると、

```
cycle  user1.sent → service to user1; user1.sent := false
□      user2.sent → service to user2; user2.sent := false
       :
□      user n.sent → service to user n; user n.sent := false
end
```

のようにかくことができる。

5.4. Hoare の guarded command

Hoare の guarded command [Ho 78] は、Dijkstra の guarded command を拡張して、guarded region と同様の機能をもたせたものといえる。Dijkstra の guarded command とのちがいは、guard として、次のような形のものをかけることである。*

<guard> ::= <guard list> | <guard list> ; <input command>
| <input command>

<guard list> ::= <guard element> {; <guard element>}*

<guard element> ::= <boolean expression> | <declaration>

<input command> ::= <source> ? <target variable>

<source> ::= <process name>

入力命令 (input command) は相手のプロセス (source) から値を受けとり、それを target variable へ代入する。

---

* 記法上の問題だが、Hoare は if と do のかわりにそれぞれ [と *[、fi と od のかわりに ともに] を用いている。

すべてのguardが1つの論理式からなるとき、それを含む選択構文、くりかえし構文（Hoareはalternative command, repetitive commandという用語を用いている）の意味はDijkstraのそれに等しい。一般には次のように解釈される。

選択構文の場合は、次の条件をみたすguardがあれば（もしそのguardが"入力命令を含めば"）それが含む入力命令を実行した後、それに続くguarded listを実行する。その条件とは、そのguardに含まれるすべての論理式の値が真であって、さらに入力命令が含まれれば、その相手プロセスが通信準備のできた状態にあるということである。すべてのguardに値が偽の論理式が含まれるか、相手プロセスが停止した入力命令が含まれれば"実行を"放棄"する。しかし、次のようなguardが存在するときは、上のいずれかの条件がなりたつまで"待つ。すなわち、そのguardに含まれるすべての論理式の値は真だが、それは入力命令を含み、その相手プロセスは通信準備が"できていない（あるいは別の通信をおこなっている）ようなguardが存在するときである。

くりかえし構文の場合は、もしすべてのguardに値が偽の論理式が含まれるか、相手プロセスがすでに停止した入力命令が含まれているとき、その実行は終了する。そうでなければ、選択構文と同じようにして1つのguarded commandを選択・実行した後、そのくりかえし構文をはじめから実行する。

すべてのguardが（変数宣言と）入力命令だけからなる場合を考えてみると、他のプロセスとの結合が共有変数によるか、メッセージバッファによるかという違いはあるが、他のプロセスが適当な処置をおこなうまで先へ進めないという点でguarded regionと一致する。

5.1の問題の解は次のようにかくことができる。

```
c: command;
do user1?c → service to user1
[] user2?c → service to user2
   :
[] usern?c → service to usern
od
```

No.

Date

guarded command は、

このように Hoare の Dijkstra の guarded command と、guarded region をたくみに組みあわせたものと考えることができるが、特に注目すべき点は、cycle-statement を実行するプロセスは永久に止まらないのに対し、Hoare のくりかえし構文の実行は終わりうるということである。これは、アルゴリズムを記述する手段であるプログラム言語にとっては重要な条件であり（なぜなら、止まらない手続きはアルゴリズムではないから）、実際問題としても重要である（自発的に止まらない実行は、人間が止めてやらなければならないから）。

Hoare の guarded command は、Dijkstra の guarded command と、guarded region という本来異なった点のある概念を1つに合わせてしまったという点で、アルゴリズムの本質をややぼかすことがあるかもしれないが、比較的単純な構文で様々の手続きをかきあらわすことができるという点ですぐれている。

5.5. Dijkstraの被護命令(guarded command)

Dijkstraの被護命令は、Hoareのそれを基にしているが、次の点で異なっている。それは すべてのguard(Dijkstraの用語では護衛列)の値が偽のときは選択命令の効果を定義しないとしたことである。これは、実行の効率をあげることができるようにするための決定である。このため、Dijkstraの選択命令は、implementationのしかたしだいではwhen-statementのようにもなる。実際問題としては、debug modeのときはすべてのguardの値が偽のときerror messageを出すようにし、no-debug modeのときはguardの評価をつづけるようにすればよい。完成されたプログラムにおいては選択命令の実行が失敗するような事態をひきおこしてはならない——そのような事態がおこらないことが保証されているのであれば、わざわざ高価なコードを出す必要はないのである。

なお、くりかえし命令の方はHoareのそれと同じである。

6. 細かい設計上の決定について.

以下. Hoare の言語案と Dihybrid とを比較しながら、反省も含めて まだ述べていない Dihybrid の特徴について述べていく。

6.1. Hoare の入力/出力命令と Dihybrid の入出力命令

6.1.1. 通信相手の識別法

Hoare [Ho78] の入力命令/出力命令は、5.4でもふれたが次のような形をしている。

p?v　　p!e

pはプロセス、eは式、vは変数である。通信が成立する条件は、等しい型の変数または式をもつ入力命令と出力命令がともに実行開始されることである。constructor を用いることによって通信の相手命令を制限することができる。たとえば、プロセス q の p!constr(e) なる出力命令は、(ここで constr(・) が constructor である)、プロセス p の q?constr(v) のような入力命令としか整合しない。

Dihybridの入出力命令は路(channel)によって相手を識別する。たとえば、

c!p(e)　　c?p(v).

ここで、cは路、pはプロセス、eは式、vは変数である。異なった路を用いる入出力命令の間では通信はおこらない。路は、どのプロセスとどのプロセスが、どの型の値をいくつ入出力するかをすべて(コンパイル時に)決する。

constructorと路の機能は似ているが、後者の方がプログラムの誤りによる予期しない通信を防ぐ効果があると思われる。たとえば、プロセスpはその中の2か所P_1, P_2で型Tの式eの値をqへ送る、プロセスqはやはり2か所Q_1, Q_2で型Tの値をpからうけとるとしよう。Hoare流のかきかたでは、pにあらわれる出力命令はq!e、qにあらわれる入力命令はp?vの形となろう。Dihybridでは、1) $P_1 \to Q_1$, $P_2 \to Q_2$ のような通信しかおこなわなければ、

P_1には　c1!q(e)、　P_2には　c2!q(e)

Q_1には c1?p(v), Q_2には c2?p(v)

のようにかくである。 2) $P_1 \to Q_1$, $P_2 \to Q_2$ だけでなく、$P_1 \to Q_2$, $P_2 \to Q_1$ のような通信もおこなわれるとすれば、

P_1, P_2には c!g(e), Q_1, Q_2には c?p(v)

のようにかくであろう。 上に述べた Hoare流のかきかたは、Dihybrid流のかきかたの後者の場合(2)に相当する。constructionを用いることによって前者(1)と同じ意味を実現することはできるが、前者のような場合にも上に述べたようなかきかたをしやすいと考えられる。 はじめから、あるいはプログラムの一部を直したことによってプログラムに誤りが生じて、P_1とQ_2, P_2とQ_1が同時に実行されうるようになることがありうるとすれば、Dihybridの方が危険が少ないと考えられる。

しかし、Dihybridの記法は、ときにはわずらわしく感じられる。 たとえば、プロセス g[i] ($1 \le i \le n$) のうちのいずれかから送信があったとき CS(i) を実行するという操作をくりかえすとき、Hoare流では、これを

do [i:1..n] q[i]?v → CS(i) od *

とかくが、Dihybrid では

do [i:1..n] c[i]? q[i](v) → CS(i) od

のようにかく。路配列 c を用意することが必要になるのである（c[i] とかくかわりに単純な路 c をかくと、路によって相手プロセスが決まるという規則に反することになり、コンパイル時誤りとなる）。

6.1.2. 複数のパラメタ列

Hoare の入力命令、出力命令は、ただ1つの相手に対し入力のみ、または出力のみをおこなうものである。これに対し、Dihybrid では1つの入出力命令で複数の相手に対し、入力と出力を同時におこなうことができる。これは、そうした方が効率を上げられる可能性があり、また思わぬ dead lock をさける効果もある [第1部 3.4.1.2.1]

---

* Hoare の記法に忠実にかくと次のようになる。

*[(i:1..n) q(i)?v → CS(i)]

~~無理に~~

＊ たとえば、Hoareの方法を拡張して次のようなプログラムをかいたとしよう。

例 P:: $q!e_1 \& r?v_1$;

Q:: $p?v_2 \& r!e_2$;

R:: $p!e_3 \& q?v_3$; ∎

'&' はそれで結ばれた入出力命令を同時におこなうことを意味する。

上の3つの入出力命令は、e_i と v_i $(i=1,2,3)$ の型が等しければ対応しうるが、そのうちいずれかの型がちがっていればこれらは対応しない。従って、どの命令が対応するかは容易にわからない。

目でみて

と考えたからである。もっとも、第1の点に関しては、少なくとも今回のM³へのimplementationにおいてはやや否定的な結果が出た［第3部3.1.6.2］。

なお、6.1.1で述べたdynamic type checkingによる方法では、このような拡張は困難だということに注意されたい。*(←)

6.1.3. 入出力護衛のパラメタ列を1つに限ったこと。

普通の入出力命令の場合は6.1.2で述べたように複数のパラメタ列を許したにもかかわらず、入出力護衛の場合はパラメタ列を1個しか許さないことにした理由は、それほど明確ではないが、入出力護衛の意味をはっきりさせること（2つ以上のパラメタ列をもつ入出力護衛の実行は、2つ以上の議案をだきあわせで議会を通過させるようなものである）と、implementation上の困難を防ぐことが目的であった。

6.1.4. 路の宣言

Dihybridでは、路は宣言しなければならない。路名がプログラムの中にはじめてあらわれたときに、それとわかる

＊　例えば、プロセスpの中にc1!g(e)とかくべきところを
c2!g(e)とまちがえてかいたとする。路宣言がない場合、この
誤りは検出されない（gの中にc2を使う入出力命令が含まれ
なくても誤りでないからうが、路宣言でc2が宣言されていれ
ば検出される。

ようにするためである。最初の記号が名前であるような命令がいくつかあるが、語の宣言をすることによって、それが何の名前であるかによって、何の命令がはじまるかがわかる(宣言されていない名前があらわれたとき入出力命令がはじまると考えることもできるが、あまり良い方法ではない)。
また、これにより、語名のつづりの誤りを検出しやすくなる。(↔)

6.2. 有効範囲則(scope rule)

6.2.1. 大域的有効範囲則と局所的有効範囲則

Algol的な有効範囲則は様々の問題をひきおこす。Algol的有効範囲則をもつ言語Pascalについて考えてみよう。たとえば、標準の名前('integer, true, ord などで)はプログラマが再定義することができる。このことは、便利なこともあるが、よしろプログラムの読みやすさを妨げることが多いであろう。とくに、Pascalにはスカラ型(scalar type あるいはenumeration type)というのがあるが、あるレベルで定義されたスカラ型に属する名前と同一の名前を別の

レベルで"再定義したような場合に奇妙な現象がおこる。

たとえば" true は システムの中で"

type Boolean = (false, true)

として定義されたものと考えることが"できるが、これをプログラムの中で"再定義すると、 true とかくと あとの定義の true を意味するが、 succ(false) とかくと もとの true をアクセスすることが"できる。Pascal が one-pass でのコンパイルをめざして設計されていることと考えあわせると、さらに奇妙な点がいくつかあるということが[WSH77]で指摘されている)。

これらの問題をさけるには、標準の名前は予約する(すなわち再定義を許さない)とともに、プログラマにも予約された名前の定義を許すのがよいと考えられる。しかし、すべての名前の有効範囲を大域的にするのは好ましくないから、 Dihybrid の有効範囲則は、大域的——局所的 の2本立てとした。

しかし、Dihybrid の有効範囲則には重大な問題点があ

る。有効範囲則が1つでないといっても、真に2つだけでかたづけば比較的かんたんである。しかし、局所的有効範囲則というのは複雑な規則をまとめたものへの名称である。共通な特徴はあるとはいうものの、それぞれの構文が独自の有効範囲則をもっており、文法ではこれらを別々に記述するしかなかった[第1部]。このことがDiHybridという言語の見通しをかなり悪くしていることは否定できない。

6.2.2. マクロの有効範囲に関する問題

マクロは、容易にimplementできるように、複写規則をあまり複雑にしないようにした。そのため、奇妙な現象がおこりうる。マクロの中では、それが呼びだされる場所で使える名前が使えるといったことがおこるであろう。たとえば、

```
mpros m = begin a := b+c end;
begin var a,b,c : int ; read?in(b,c); m;
```

write!out(a, eol)

end.

のようなプログラムが許されうる。Dihybridの仕様は、このようなプログラムを許してはいないが禁止してもいない。(このようなプログラムはかくべきでないがコンパイラはそれをチェックしないであろう)。

6.2.3. forwardの問題

Dihybridでは宣言はそれより前では有効でないとした。このため、同じ規則をもつPascalと同じようにforward宣言が必要になった。forward宣言は、同じ名前が2度以上"宣言"されるという点であまり気持ちのよいものではないが、手続き、関数の場合には、mutual recursionのあるとき必要になるだけだからそれほど問題ではない。しかし、並列命令におけるプロセスの宣言の場合、プロセス間の入出力があればたいがいforward宣言が必要になる。これはプログラムをきたなく

するといわざるをえない。

6.2.4. 並列命令内への輸入

Hoare は並列命令の外のものをその中へもちこむこと（すなわち輸入）には言及していない。この問題にはおおまかにいって、3つの答えがありうる。

① あらゆるものを無宣言で輸入できるようにする。

② 輸入はいっさい禁じる。

③ 宣言をすれば（制限つきで）輸入を認める。

①は、共有メモリのない計算機では非現実的であるとともに、同期化の道具として並列命令と入出力命令だけを入れようという決定をくつがえすものである。

②は最もかんたんな解ではあるが、並列命令は外部との通信の手段を全く失ってしまい、無意味になる（他の方法で通信を認めることは考えられるが、すなわち、入出力命令）

Dihybrid ではそれを許していない。

従って③が適当な方法ということになる。Dihybrid

では変数と定数だけ輸入を認め、しかも変数に関しては、共有メモリを必要としないようにその意味を定めている。

Dihybridでは並列命令のはじめで、いずれかのプロセスが必要とする外部の定数、変数を宣言し（輸入リスト）、各プロセスのはじめで再び、そのプロセスが必要とするものだけ宣言する（定義リスト、使用リスト）形をとっている。これはone-passで輸入のための処理ができるようにするためである。言語上の要求としては、輸入リストは不可欠なものではない。

6.2.5. 路の輸入を許さない理由

並列命令内へ路を輸入することを認めれば、現在より柔軟な通信が可能になる。それにもかかわらず路の輸入を禁止したのは次の2つの理由からである。

① 標準プロセスとの入出力に関する問題：

M³やCm*[SFS77]のような計算機システムにおい

ては、入出力装置は処理装置の局所的なbusに接続されている。従って、それらは他の処理装置から直接アクセスすることはできないか、あるいはできるとしてもかなりのoverheadがかかる。従って、並列命令の中では特定の1つのプロセスだけが標準プロセスを使いうる《あるいはいずれも直接使うことはできない》ということになるが、論理的整合性を保つためには、標準プロセスとの入出力は並列命令の中では禁止するのがよいと考えられる。*

② 相手プロセスがactivateされていないときの扱い：

路の輸入が許されたとして、次のようなプログラムを考えてみよう。

* この制限はきつすぎて、Dihybridの用途をせばめてしまうということも確かである。

```
co channel c;    g:: forward
|| p:: ---
    do B →                    exports p1;
        co imports c;  p2:: forward
        || p1:: ---
        || p2:: ---
        oc
    od
|| g:: ---        c?p1(---); ---
    do c?p1(…) → ---    od
oc.
```

P内の並列命令が実行されていないときに、g内の入出力命令が実行開始されたとすれば、どのような処置をとればよいだろうか。また、その並列命令が何回か実行されるとき、p1, p2は生成、消滅をくりかえすことになるが、それを同じものとみなして通信してよいだろうか。g内のくりかえし命令で、あるときp1が停止してそれからぬけたとしても、再びそれを実行すると、p1は再生成されていて、通信は成立するかもしれない。これは奇妙であるばかりではなく、危険ではないだろうか。これらの困難をさけるには、路の輸入を禁止するのが最善だと思われる。

6.2.6. ルーティンの輸入を許さない理由

ルーティンの目的コードは1つの処理装置のみにloadされる。従って、それが実行するプロセス以外のプロセスからよびだされるべきではない。従って輸入は禁止すべきであるということになる。

さらに深刻な問題は、ルーティンの中で並列命令を使う場合におこる。ルーティンの中の並列命令の中で、そのルーティンを再帰的に呼びだすと、生成されるプロセスの数は有界ではなくなる。同じ並列命令の同名のプロセスが、再帰の各レベルごとにactivateされるからである。それらのプロセスは別の処理装置にわりあてられるか、同じ処理装置上に時分割的にわりあてられなければならない。この事態はなんとしてもさけなければならない。

6.3. マクロと準マクロ機能

6.3.1 準マクロ機能

Dihybridにはいくつかの「省略記法」がある.

① 被護命令に冠した範囲 ([i:1..3]のようなもの).

例 if [i:1..2] c[i]?p[i](v) → CS(i) fi

は

```
if c[1]?p[1](v) → CS(1)
□ c[2]?p[2](v) → CS(2)
fi
```

に等しい.■

② 入出力命令における範囲

例 c!p[i:1..3](e[i]) は

c!p[1](e[1])!p[2](e[2])!p[3](e[3])

に等しい.■

③ プロセスの宣言における範囲

例 P[i:1..3]::PB(i) は

P[1]::forward || P[2]::forward || P[3]::forward
|| P[1]::PB(1) || P[2]::PB(2) || P[3]::PB(3)

に等しい。

④ 輸入リスト、定義リスト、使用リストにおいて、

例 imports ひ[1..3] は

imports ひ[1], ひ[2], ひ[3] に等しい。■

これらの記法は単にプログラマやパンチャの手を省くだけでなく、データやハードウェアが変化したとき対処しやすいようにするという意味がある。[第3章]で述べる方法で処理装置のわりつけをすると、プロセスの数は処理装置の数によって制限されてしまうが、たとえば"処理装置が5台のとき P[1], ···, P[5] が各処理装置にわりあてられているとして、

const n=5

if [i: 2..n] c[i]?P[i](v) → CS(i) fi

は自分(P[1])を除くすべてのプロセスのうちの1つからの送信を受けてCS(i)を実行するが、n=5をn=16にかえれば、処理装置が16台の場合の同様の意味をもったプログラムを得ることができる。

6.3.2. マクロ

マクロは並列命令でルーティンの輸入を認めないこの理由は共有メモリがなければコードの共有ができないことにある）ことの代償として、すなわちルーティンの機能を補うために導入されたもので"ある。従って、その機能はプロセス名札の範囲（[i: 1..2]など）と似ている。

例　プロセス p[1] と p[2] が同じ手続き pr を利用したいときには、

p[i: 1..2] :: --- proc p(v: int) = begin BODY end; ---

のようにかけば"よいが"、pとqがともにprを使いたいときは、

mproc mpr(v) = begin BODY end

というマクロ手続きを宣言しておいて、

p:: --- proc pr(v: int) = begin mpr(v) end; ---

q:: --- proc pr(v: int) = begin mpr(v) end; ---

のようにかけば"よい。■

6.3.3. マクロ手続きの機能の限定

Hoare [Ho78] は彼の言語で"マクロ"が使えれば"便利だ"ということを述べているが、[Ho78]で使われているマクロは、マクロの中での宣言が、マクロ手続き命令のあとでも有効となるようなものである。たとえば、

```
mproc mp \= begin var v int end
mp;  v := 10 {vはmp内で宣言されたv}
```

のようなことが可能である。この機能は悪用するとプログラムを非常にわかりにくくするので、Dihybrid ではそのようなことがおこらないようにしたが、そのために、せっかくマクロがあっても、わずらわしいかきかたをしなければならない場合がありうる(6.3.2の例もその1つである)。

6.3.4. マクロ関数の機能の限定

マクロ関数は 関数にくらべると全く不充分な機能しかもっていない。それは、その本体が命令列ではなく、式だからである。 たとえば

```
func F(i: int) returns f: int =
  begin
    if  i=0 → f:=23
    □   i=1 → f:=17
    fi
  end
```

のようなかんたんな関数でも、これをそのままマクロ関数にかきなおすことはできない.*

マクロ関数を作ったのは、むしろ ord, chr, abs といった標準マクロ関数を特別なものとみなしたくなかったからである。

---

* この例においては、次のようにすれば、 i=0, 1 に対し 同じ意味をあらわすことはできる.

```
mfunc F(i) = 23 - i*6
```

Date

6.4. その他

6.4.1. ルーティンの導入

Dihybrid では、4.1で述べたように、並列命令と入出力命令を使えば" ルーティンを使わなくても関数や手続きを模倣することができる。しかし、この方法で"「関数」や「手続き」を記述した場合、multi computer system へのimplementationにおいては、特に凝った方法を使わない限り、「関数」、「手続き」とそれらをよびだすプロセスとは別の処理装置上で走ることになり、特に再帰的手続きの場合は、その深さだけの処理装置が必要になって非現実的である。従って、よびだすプロセス自らが それを実行するような関数、手続きを導入することが必要だと考えたのである。

6.4.2. 被護命令、並列命令の記法

Hoareは if, fi, do, od ca, es ではなく、それぞれ [,], *[,], [,] を用いているが、前者の方が意味がわかりやすいと思われる。また、Hoareの記法では、選択命令と並列命令の区別がつきにくいが、最初の1記号だけでこれらの区別ができるようにしたかった。

6.4.3. 宣言詞と宣言の区切り記号

Hoareの言語では、変数宣言は宣言すべき変数名ではじまるが、Dihybridでは宣言詞 var をつけた。これも最初の1記号だけで変数宣言とわかるようにするためである。const, proc, func なども同様である。

変数、定数の宣言はPascalに似た形にしてあるが、区切り ; のかわりに , を使っているところがある。

たとえば、Pascalでは

var v1, v2 : integer ; v3 : Boolean

とかくところを Dihybridでは

var v1, v2 : int; v3 : Bool

とかく。これは、;がくれば"ただちに"その宣言は終わりということにしたかったためである。こうしないと、宣言詞 var や const をつけた意味がなくなってしまう（コンパイラは 'v3' をよんだところで、それが何のはじまりなのかわからなくなってしまう）。

6.4.4. 最初の1記号で次の構文単位がわかるようにしたこと.

すでに強調したように、文法はあらゆる場所で最初の1記号をみるだけで、次にくる構文単位を予測できるように設計した。ただし、構文だけでは区別のつかないところがある（たとえば引数のない関数呼びだしと変数、マクロ手続き命令と手続

き命令など)。しかし、これらも意味を調べると区別できる。

6.4.5. 命令列中の任意の場所で"宣言ができること

変数、定数はそれが必要になるまで"宣言しない方がプログラムは書きやすく、読みやすくなると考えられる。とくに入出力命令の入力パラメタにあらわれる変数はその直前で"宣言したいことが多い。そのためHoareは任意の場所で"変数が宣言で"きるようにしており、Dihybridもこれをそのままとりいれた。しかし、手続きや関数が任意の場所にあらわれるのはむしろ適当で"ないかもしれない。譲渡汚れの中だけは、ここに手続きや関数があらわれると著しく読みやすさを阻害・すると考えられるので、定数、変数だけ宣言で"きるようにした。

7. 評価

implementationが完成していない現在、完全な評価はできないが、Dihybridの設計にあたって、成功した点と失敗した点をあげよう。

成功したと思われるのは、設計中、implementationのことを常に意識していたので、言語設計ができたあと、implementation上問題をひきおこしたところは特になかったということである。

失敗したと思われるのは、次のような点である。Dihybridはその名前にもあらわされているように、いくつかの言語の構文や概念を借りてきて寄せ集めてできたようなものである。これはよい言語を生みだすために必要な第1段階ではあるかもしれないが、その後充分な洗練の過程を経ることが必要である。その点でDihybridは全く不充分だったといえるだろう。有効範囲則に代表される言語の複雑さはこれを象徴するものだといえよう。

文献

[Br73] Brinch Hansen, P.: *Operating System principles*, Englewood Cliffs, NJ: Prentice-Hall [1973]

邦訳 田中穂積 他訳:
オペレーティングシステムの原理, 近代科学社[1976]

[Br75C] Brinch Hansen, P.: Concurrent Pascal Report, Information Science, CIT [1975]

[Br75P] Brinch Hansen, P.: The programming language Concurrent Pascal, IEEE Trans. on Software Engineering SE1:2 [1975], 199-207

[Br78] Brinch Hansen, P.: Distributed processes: A concurrent programming concept, Comm ACM 21:11 [1978], 934-941

[BS78] Brinch Hansen, P., and Staunstrup, Jørgen: Specification and implementation of mutual exclusion, IEEE Trans. on Software Engineering, SE4:5 [1978], 365-370

[Co63] Conway, M.E.: A multiprocessor system design, Proc. AFIPS FJCC [1963], 139–146

[Da72] Dahl, O.J.: Hierachical program structures, in Dahl, Dijkstra, Hoare: Structured programming, Academic Press, New York [1972]

邦訳 野下浩平 他訳: 構造化プログラミング, サイエンス社

[Da67] Dahl, O.J. et.al.: SIMULA 67, common base language, Norwegian Computing Centre, Forsknigreien, Oslo [1967]

[De75] Dennis, J.B.: First version of a data flow procedure language, MIT/LCS/TM-61 [1975]

[Di68] Dijkstra, E.W.: Cooperating sequential processes, in Programming Languages, F. Genuys. ed., Academic Press [1968], 43–112

[Di75] Dijkstra, E.W: Guarded commands, nondeterminancy and formal derivation of programs,

Comm. ACM **18**:8 [1975], 453-457

[Ha72] Habermann, A.N.: Syncronization of communicating processes, Comm. ACM **15**:3 [1972], 171-176

[Ho74] Hoare, C.A.R.: Monitors: An operating system structuring concept, Comm. ACM **17**:10 [1974], 549-557

[Ho78] Hoare, C.A.R.: Communicating sequential processes, Comm. ACM **21**:8 [1978], 666-677

[Og78] 小笠原 司: マルチマイクロプロセッサシステムのプログラミング手法, 東京大学計数工学科卒業論文 [1978]

[On78] 小野輝: ____________________
____________________, ____________________
____

[Ru77] Rumbaugh, J.: A data flow multiprocessor, IEEE Trans. on Computers, **C26**:2 [1977], 138-146

[SFS77] Swan, R.J., Fuller, S.H., and Siewiorek, D.P.: Cm* - a modular, multi-microprocessor, Proc. AFIPS NCC [1977], 637-644

[Wa78] 渡辺 淳: マルチマイクロプロセッサシステムの試作, 東京大学計数工学科卒業論文 [1978]

[Hi78] 平沢裕: ———————————— ———, ————————————

[Wi76] Wirth, N.: MODULA A language for modular multiprogramming, Software - Practice & Experience 7:1 [1977], 3-35

[WSH77] Welsh, J., Sneeringer, W.J., and Hoare, C.A.R.: Ambiguities and Insecurities in Pascal, Software - Practice & Experience, 7 [1977], 685-696

[Ku68] Kuck, D.: ILLIAC IV Software and application programming, IEEE Trans. on Computers, C17:8 [1968], 758-770

第3部

Dihybrid の implementation

Dihybrid の implementation

第3部では、Dihybrid の M³-system への implementation について述べる。Dihybrid の主な特徴は 被護命令、並列命令、入出力命令にあるから、ここでも それらを中心として述べることにしよう。

1. はじめに

Dihybrid を設計するにあたっては、これを implement すべきハードウェアがすでに完成していた。従って、そのハードウェアの上にいかにすれば"implement できる"かが、設計のときから念頭にあった。時間的余裕がなかったため、他の、より一般的なハードウェアにどのようにすれば"implement できる"ということを充分に考察することはできなかった。

以下に述べる方法は、M³-system に implement するための方法であり、他のハードウェアにおいては、これとは全く異なった方法でなされるべきであるかもしれない。また、M³に対しても、他にもっとよい方法があるかもしれない。

2. 被護命令

2.1. 任意性の実現

選択命令、くりかえし命令が普通のif文、while文(DO文)とちがう点は、非決定性あるいは任意性にあるが、その「任意性」の解釈には任意性がある。それをもっともきびしく解釈すれば、さいころをふる必要がでてくるが、それはばかばかしい非効率的である。どの程度の任意性を必要とするか考えてみよう。

選択命令およびくりかえし命令の実行開始直後にどの護衛列を選択するかという問題についても「任意性」がある方がよいが、常に同じ護衛列が選択されたとしてもそれほど不都合はないであろう。また、護衛列の値がfalseであったとき、次にどの護衛列をえらんで評価するかも決まっていてさしつかえないと考えられる。入出力護衛で待たされた場合も同じである。

しかし、今の場合もっと巨視的には任意性の保証を(正確には公平さ[Br73])

する必要がある。すなわち ひとつの護衛列を評価したあと、同じ選択命令あるいはくりかえし命令内にとどまるならば、次に同じ護衛列を評価するのは他の護衛列をすべて調べたあとにする。くりかえし命令の場合、命令列が実行されたあと再び護衛列の評価にもどるが、この場合も同じようにする。つまり、実行したばかりの命令列に先行する護衛列を評価するのは、他のすべての護衛列を評価したあとにする。

この決定に基いて、次のようにimplementすることにした。

2.2. 選択命令

if $C_1 \to S_1$ □ ··· □ $C_n \to S_n$ fi

は（no-debug modeのとき）次のような形に翻訳する

（ここで loop ··· end は無限のくりかえしをあらわす。

when C do S exit は Cの値がtrueならばSを実行して loop ··· end からぬけることを意味する。do Sがないときは ただちに脱出する。また、C_iは true, falseのほ

かに waiting という値もとるものとする。waiting は、C_iが含む入出力護衛行の相手プロセスが通信可能な状態にないときだけとるものとする*)。

```
loop
  when C1 do S1 exit;
       :
  when Cn do Sn exit;
end.
```

普通の if 文に比べて手間がかかるのは、C_nの評価が必要なことだけである。

debug mode のとき、すなわちすべての護衛列が偽のとき、その旨メッセージを出力したいときは、次に述べるくりかえし命令に準じればよい。

2.3. くりかえし命令

do $C_1 \to S_1 \square \cdots \square C_n \to S_n$ od

は次のような形に翻訳される（v_{do}は他では使われていない変数とする）。

---

* 3.1.5.3 参照

```
begin var vdo : set of 1..n ;  vdo := [] ;
loop
  when vdo = [1..n] exit ;
  case c1 of
   false :  vdo := vdo ∪ [1] ;
   true  :  begin  S1 ;  vdo := []  end ;
   waiting : ;
  end ;
    :
  when vdo = [1..n] exit ;
  case cn of
   false :  vdo := vdo ∪ [n] ;
   true  :  begin  Sn ;  vdo := []  end ;
   waiting : ;
  end ;
 end {loop}
end {begin}.
```

*　　**

ただし、被護命令をただひとつ含むくりかえし命令は、

もっと簡単にimplementできる。　すなわち、

```
do  C → S  od
```

は

---

*　[] は空集合をあらわす.

**　[1..n] は 集合 {1, 2, ..., n} をあらわす.

while not C do if C then S fi od

のように訳せばよい（while not C do ... od は、Cが waiting のときも ... をくりかえす）。プログラムにあらわれるくりかえし命令の多くは被護命令をただ1個含むものだと考えられるから、このようにすることにより、かなりの改善が期待できる。このやりかたをとった場合、くりかえし命令の効率は while 文とほとんど変わりなくすることができる。

3. 入出力命令

入出力命令に関しては、それを実際にどのようにしておこなうかという実行時の問題と、パラメタの対応をどのようにして検証するかというコンパイル時の問題とがある。

3.1. 入出力命令の実行

3.1.1. 通信機構のモデル

通信機構に関する2つのモデルをつくってみよう。いずれの場合も各処理装置は専用の Communication Buffer をもっているものとする。このCBはそれが属する処理装置だけが書きこむことができ、それ以外のすべての処理装置からなんらかの方法でよみだされる。

モデル① 他の処理装置のCBを常によめる場合：

書きこみをしている間を除けば常にCBの内容が他の処理装置からよめる。この場合非決定論的な通信がimplementしやすいが、ハードウエアが複雑になるか、通信によるoverheadが大きくなる。

モデル② 通信要求が認められたとき、その相手との間だけで CBの内容の授受ができる 場合：

入出力命令の機能をそのままハードウェアのレベルまでもちこんだ場合といえる。このような通信の排他性をみたすためには かしこい(intelligentな)通信制御装置の存在が不可欠であろう。

この場合、非決定論的な通信ができるためには、また通信がおこっていない間は通信要求をとりさげることができなければならない。通信制御装置が充分高速でないと、それがシステムのあい路になりうる。■

3.1.2. M^3の通信に関するハードウェア

M^3-system は、現在5台の処理装置*からなっているが、そのうち1台は host とよばれ、他は membersとよばれ、その機能は異なっている。members には次の命令がある。

* Texas Instruments TM-990/100M

・CKON ： データ転送の禁止をとく命令。hostを含むすべての処理装置がCKONを出すと、すべての処理装置の間でデータ交換がおこなわれる。その様子は図1のとおりである。

・CKOF ： データ転送が終わるまで待つ命令。CKONの実行後 またデータ転送がおこっていなければ、それが終わるまで次の命令へ進まない。すでにデータ転送がおこり、CKONが無効になっていれば効果はない。

・ISOL (=LREX) ； ISOLの実行は割出しをともなうので、割出し手続きを変えればISOLの意味を変えることができるが、ハードウエアに備わっている機能は、データ転送に参加しないことの宣言にある。すなわち、ISOLを実行した処理装置は他の処理装置間のデータ転送を妨げない(その処理装置のCBには0がかかれているようにふるまう)し、それから影響を受けない。(ただし、データ転送後 割込み信号が来るので、それを受けつけることは

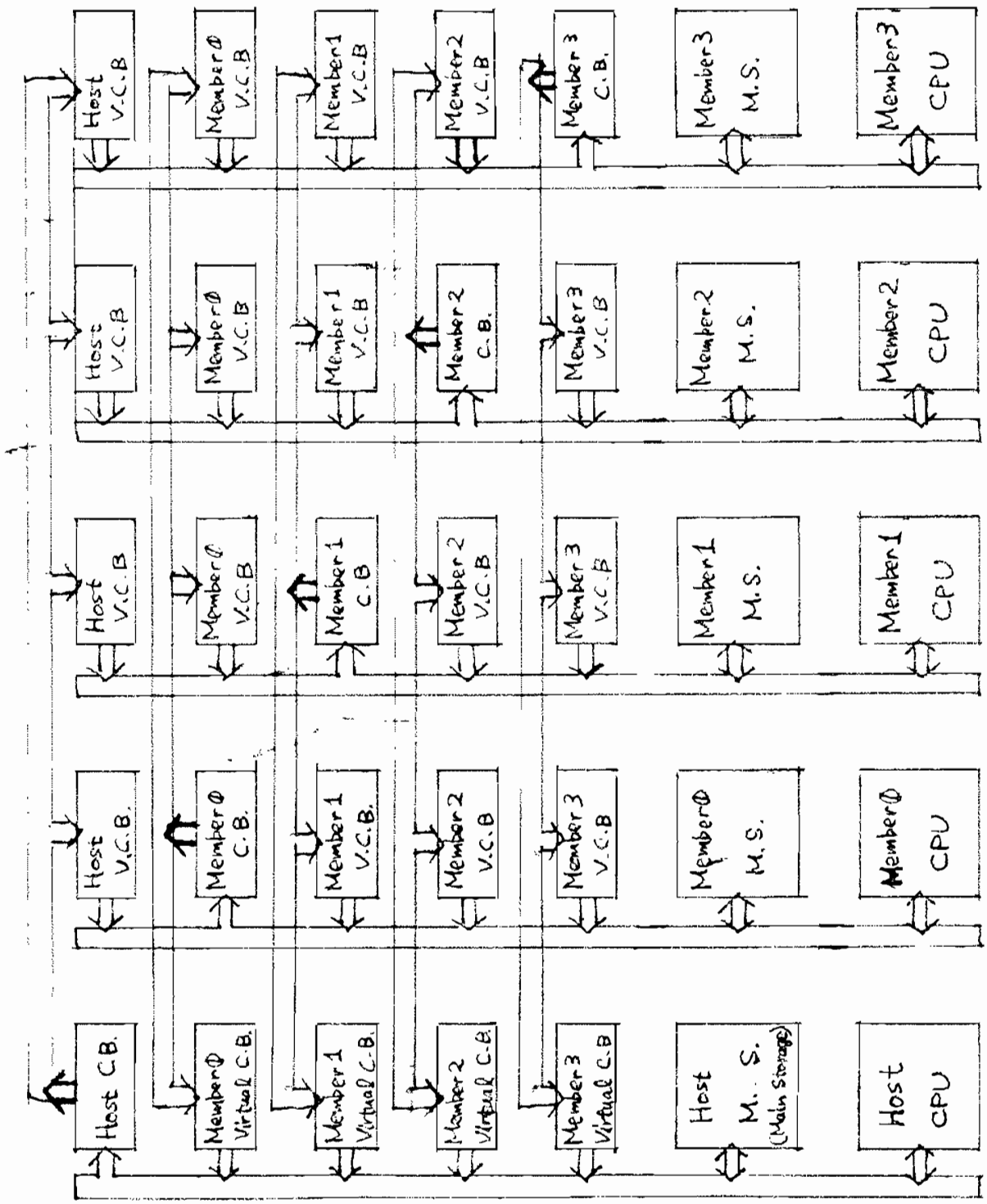


図1　データ転送の様子を ⟶ で示した.

できる。

また、hostには上の命令のうちCKONのみがある。(他の2つの命令は通信以外の目的に使っている)。M³のハードウェアに関するくわしい説明は[Wa78]、[Hi78]にかかれている。*(次頁)

M³はモデル②で述べたような意味の通信制御装置をもっていない。データ転送装置の機能はきわめて限定されている。しかも、それはモデル①に近い状態をつくりだすための手段として与えられているということができる。しかし、hostを通信制御装置として使うことは可能であり、そうすればモデル③の条件をみたすことはできるであろう。だが、そのほとんどをソフトウェアにたよることになるから効率はよくない。従って、①に近い形を採用するのが適当と考えられる。以下このことを仮定する。

M³の通信機構と①との相違点は、データ転送がある処理装置からみたおこるまで他の処理装置のCBがみかけ上更新され

ないことにある。従って、①と同じ条件を実現するためには相手のCBをしらべる前にデータ転送をおこなう必要がある。　決定論的な通信の場合は、これはとくに問題をひきおこさないであろうが、非決定論的な場合はデータ転送の回数をむやみに増やすことになる。M³では、データ転送の間、ISOLを実行していない処理装置を止めるので、特にhostはこれによって著しくその仕事を妨害されることになる。従って、データ転送の回数をおさえる工夫が必要である。

M³のハードウエアをふまえた上で再びやや抽象的なレベルにもどってimplementationの方策を考えてみよう。

＊　M³のハードウエアについては 3.1.7 で更に述べる。

3.1.3. 通信の条件

通信が正しくおこなわれるためには、次の条件がみたされていなければならない。

① 非通信状態、および異なる路のもとの通信状態を、求める路のもとの通信状態と錯誤することがない。

② あるプロセスPが路Cを用いて1回入出力をおこなう間に、Pのすべての相手プロセスも、Cを用いてちょうど1回入出力をおこなう。 ■

この条件をみたすように、処理装置のもつべき機能と通信の方法を決めよう。

3.1.4. 要求される機能と状態

各処理装置は次の(複合)命令をもっている必要がある。

① Require Communication

命令RCはパラメタを1つもつ。パラメタはそれぞれの路に固有な整数(路番号とよぶ)であって、それをCB

にかきこんだ後、すべての処理装置がデータ転送可能な状態(E状態とよぶ)にあれば"データ転送をおこさせる。もしデータ転送を禁じている(D状態)処理装置があれば、それが禁止をとくまで待ち、データ転送をおこなう。データ転送が終わるとその処理装置をD状態にして次の命令に進む。

② Wait Communication

命令WCは、データ転送の禁止をとき、データ転送がおこるまで処理装置をIdleにする。データ転送後はD状態にする。RCとのちがいは、データ転送を要求はしないことである。(データ転送が処理装置の仕事をさまたげる場合、それをむやみにおこさないためにWCが必要になる。そうでなければRCで代用できる)。

③ Enable Communication

命令ECは、通信の禁止をとくために用いられる。ECの実行後データ転送がおころうがおこるまいが、処

理装置は仕事をつづける。

RC、WCを実行してただちに通信がおこらなかったときの状態をそれぞれR状態、W状態とすると、各処理装置の状態遷移図は図2のようになる。

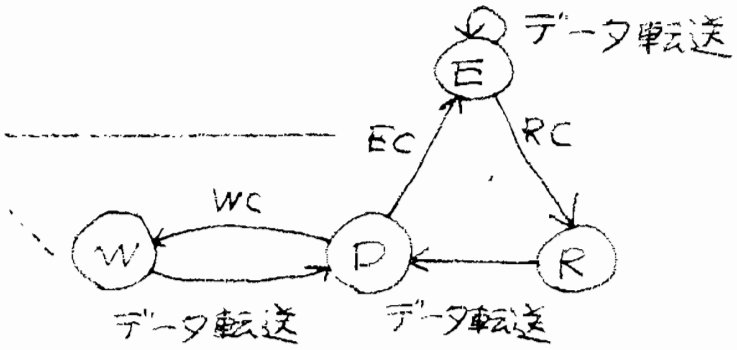


図2　1つの処理装置の状態遷移図。

状態Dはデータ転送が禁止された状態、状態Eは他の処理装置の通信を妨害しないためにデータ転送を許した状態、状態Rはデータ転送を要求してそれがおこるのを待っている状態であり、状態Wはデータ転送を要求せず、それを待っている状態である。

ある種の条件のもとではWCおよび状態Wはなくてもよい。そのかわり、Disable Communication すなわち E状態からD状態へ移るための命令が必要である（次節で更に述べる）。また、同様の条件のもとでは、WCをE状態からW状態への遷移命令としてもよい。

各処理装置のD状態を D_i、R状態を R_i とすると、E状態とW状態をあわせた状態は $\overline{D_i} \cdot \overline{R_i}$ であらわされる。このときデータ転送がおこる条件は $\bigwedge_i \overline{D_i} \wedge (\bigvee_i R_i)$ とあらわせる。すなわち、データ転送の条件がととのっているかどうかをデータ転送装置に知らせるのに、各処理装置とデータ転送装置の間には $\bigwedge_i \overline{D_i}$ または $\bigvee_i D_i$ と $\bigvee_i R_i$ または $\bigwedge_i \overline{R_i}$ の2本のバスを引けばよい。

3.1.5. 1対1の通信

まず1対1の通信にしぼって、その方法を考えてみよう。

3.1.5.1. 護衛以外の入出力命令の場合

ここでは護衛としてあらわれる入出力命令の場合は除いて考えることにする。

下に示すのがアルゴリズムだが、このうち { } で囲った部分は、それぞれ出力があるとき、入力があるときだけ実行する。入出力命令が使う路番号を CHANNEL-NR であらわす。また、その入出力命令が属するプロセスの CB（仮想CB（第1図参照））を MY_CB, 相手の を COMP_CB とする。プロセスは停止するときに CB に路番号*のかわりに terminated という値をかきこむものとする。また、非通信状態では、CB には路番号のかわりに not_communicating という値がかきこまれている。**

---

* 路番号は路名とは異なる。路配列に対して路名は1つだけついているが、路番号はその配列要素ごとにつく。

** あるいは、CB にかかれている値にかかわらず、データ転送のとき（E状態のとき） not-communicating が送出されるようになっていてもよい。その場合は次のアルゴリズムの 5. は不要である。

1. { MY_CB.data := value to send ; }
2. RC(CHANNEL_NR) ; *
3. while COMP_CB.channel_nr ≠ CHANNEL_NR do
   if COMP_CB.channel_nr = terminated then abort
   else WC
   fi
   od ;
4. { value to receive := COMP_CB.data ; }
5. MY_CB.channel_nr := not_communicating ;
6. EC **

---

* ここで" その処理装置 および その処理装置で" 実行されている プロセスは 通信準備状態に はいったという。

** EC を 実行すると、データ転送によって 仮想CBが更新されなくなる 場合は、5, 6. を 4. の前におこなうことが"できる。

Date

3.

WCのかわりにDCがあるときは、 のかわりに次の手続きをおく.

3' EC;

while COMP_CB.channel_nr ≠ CHANNEL_NR do

if COMP_CB.channel_nr = terminated then abort

od;

DC;

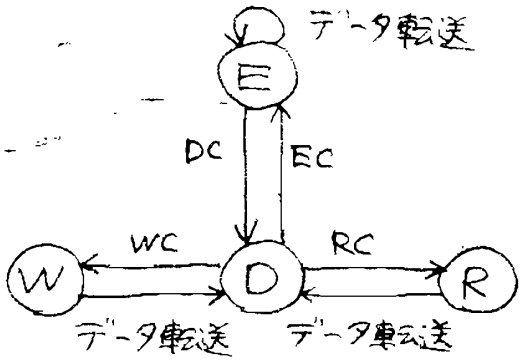
ただし、データ転送の結果 COMP_CB.channel_nr = CHANNEL_NR となってからDCが実行されるまでに再びデータ転送がおこなわれないことが保証されなければならない。

(3の別の変形について、3.1.7 で述べる)

出力の場合、出力すべき値を書いてからRCを実行すれば、不完全な出力を相手によまれてしまうことはおこらない。従って、RCの実行中、すなわち路番号の書きこみとデータ転送要求の間にデータ転送がおこらなければ、陽に臨界領域 [Br73] にはいる必要はない。あるいは、転送要求を出すまでその処理装置がデータ転送

に加わらないときも同様である。しかし、このような条件がみたされなければ"データ転送を陽に禁止する必要がある。* 入力の場合は、E状態のとき相手の仮想CBが更新される恐れのあるときは、4の操作をおこなう間ずっとデータ転送を禁止しておかなければならない。(そうでない場合は、脚注で述べたようにECを行なうタイミングを早めることができる。)

* この場合の状態遷移図は次のようになる。



3.1.5.2. 通信の様子

1対1の通信がどのようにおこなわれるかをみてみよう。

(1)プロセスpとqが通信するとき、両者が同時に通信を求めるか、またはD状態の処理装置があったために待たされると、1回の通信で両者がともに相手のCBに求める路番号を見出すことになる。従って、これで入出力は成立し、pとqは先へ進む(図3A)。(2)もしpが先にRCを実行し、データ転送がおこると、qからはpの路番号がよめるようになるが、qはまだそれをよまない。pは通信後qのCBをしらべるが、求める路番号を見出さないので待ち状態にはいる。qがRCを実行し、データ転送がおこなわれると、はじめて入出力は成立し、両プロセスは先へ進むことができる(図3B)■。

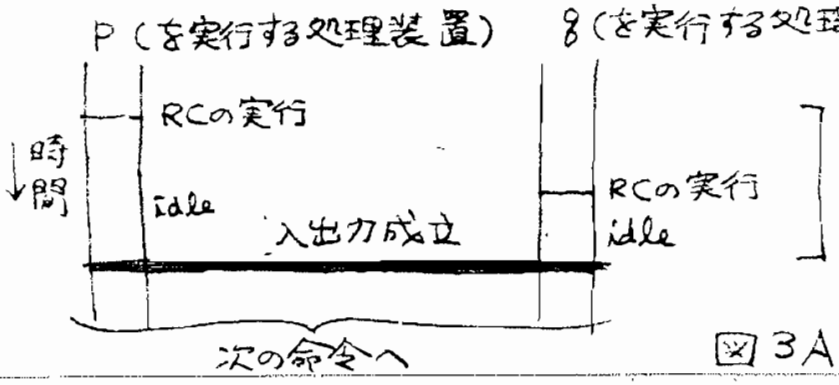


図3A p,qの通信の形の1

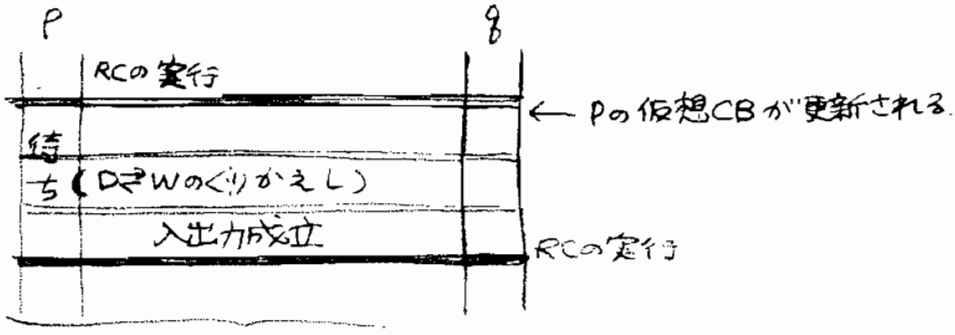


図3B　p,qの通信の形 その2

前節では、RC, WCの実行後の状態をDにした理由を充分に述べなかった。もしEにもどれば"よいとすると、遷移図は図4のようになって簡単だが、これではうまくいかない。

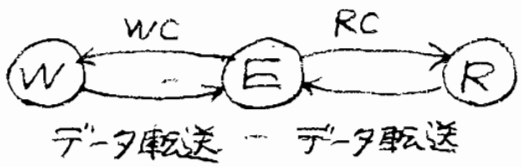


図4　不適当な状態遷移図

それは次のような場合である。RC, WCの実行後にE状態になるとすると、データ転送がおこりうるが、図3Bのような場合、pが待ち状態しかもE状態にあるときqがRCを実行すると通信は正しくおこなわれない。①もしこのときqがpのCBの内容をよむことができるとすると、qは入出力が成立したと考えて先へ進んでしまうが、pはまた"入出力が成

立したとは考えない。② もしgがE状態のpから値not communicatingを受けとったとすると、gは待ち状態にはいる。その後どのプロセスもRCを出さないとすると、pとgは通信可能な状態にありながら、デッドロックしてしまう(WCのかわりにRCを使うことができる場合には②の問題はおこらない)。■

3.1.5.3. 入出力護衛の場合

選択命令・くりかえし命令の入出力護衛を実行するときは、他の入出力命令とはちがって、相手プロセスが通信準備状態にないとき待ちつづけてはならない。従って、もし相手が通信準備状態にないときは、他の護衛列を評価しにいくが、この場合、入出力護衛を評価するたびにデータ転送の要求をすると、データ転送の頻度が極端に大きくなる(またはimplementationのしかたによっては、すでに値が真になりうる護衛列があるにもかかわらず、それを実行するまでに時間がかかる)。そのために、入出力護衛

の場合は、RCをおこなう前に相手のCBを調べることにした。すなわち、入出力護衛の実行手順はおよそ次のようになる。

```
1.  if COMP_CB.channel_nr = terminated then status := false
2.  elsif COMP_CB.channel_nr ≠ CHANNEL_NR then status := waiting
    else
3.      { MY_CB.data := value to send ; }
4.      RC ( CHANNEL_NR );
5.      { value to receive := COMP_CB.data ; }
6.      MY_CB.channel_nr := not_communicating ;
7.      EC ;
8.      status := true
    fi
```

ここで、statusは入出力護衛の値（2.2における意味）をあらわす。

ところが、1つの問題が生じる。それは、この入出力護衛」の実行の前に、同じ路のもとで通信がおこなわれ、その後データ転送がおこなわれていなかった場合、誤って通信が成立したと判断してしまうことである。これをさけるには2つの方法が考えられる。

方法①： 5.～8.を次のようにかきかえる。

```
If COMP_CB.channel_nr = terminated then status := false
elsif COMP_CB.channel_nr ≠ CHANNEL_NR then status := waiting
else
  { value to receive := COMP_CB.data;
    status := true
  fi;
MY_CB.channel_nr := not_communicating;
EC;
```

方法②： 入出力護衛」を含む被護命令の実行の前にはRCをおこなう。すなわち、選択命令の場合は、その実行(not_communicating)を開始するときに1度RCとECを続けて実行し、くりかえ

し命令の場合は、その上に1つの被護命令の実行がおわって再び護衛列の評価にさしかかると RC と EC を実行する（くりかえし命令からぬけるときはその必要はない）。(not-communicating)

①に従うと、上の手続きの2.で通信の成立を誤認した場合、道草を食うことがあるが、このようなことはまれにしか生じないであろう。これに対し、②を採用すると、問題の入出力護衛を含む選択命令またはくりかえし命令が、入出力護衛を含まない護衛列をもつ場合、RC/EC の実行はしばしばむだになるだろう。

3.1.5.4. 入出力護衛対入出力命令の通信の様子

通信をおこなう入出力命令の一方（これを実行するプロセスをpとし、相手プロセスをqとする）が護衛のときは、誤認をおこさない限り、護衛でない入出力命令どうしのときのように、データ転送が1回で通信が成立するということはない。qは必ず待たされる（すなわち、図3Bの Pと q を入れかえた形）。

方法①を用いていて、通信の成立を誤認した場合について考えてみよう。2つの場合に分けられる。第1は、誤認した直後の通信のとき、qが入出力命令以外の命令または別の路を用いる入出力命令を実行している場合である。この場合は、qにとってpの状態は無関係であるか、またはqとの通信は成立しない。pはデータ転送後、誤認したことを認識して通信要求をとりさげる。従って、なにも問題はおこらない。

第2は、誤認の直後の通信のとき、qが同じ路を用いる入出力命令を実行している(ちょうど"通信準備状態にはいった)場合である。このときは、データ転送後、pは正しく通信がおこなわれたことを"確認"し、qも通信が成立したと判定する。すなわち、pはたしかに誤認をしたにもかかわらず、正しく通信がおこなわれるのである。しかも、データ転送は1回ですむ。

3.1.6. 多対多の通信

3.1.6.1. 多対多の通信に関する強い解釈と弱い解釈

3.1.3で述べた通信の条件は「弱い解釈」に従ったものである。「強い解釈」では2番目の条件が次のようになる。

② あるプロセスpがcを用いて1回入出力をおこなう間に、cを用いるすべてのプロセスも、cを用いてちょうど1回入出力をおこなうこと。■

2つの解釈のちがいは、次のような場合に明白である。

```
co  channel c1, c2;
    g:: forward || r:: forward || s:: forward
||  p::  c1?g()
||  g::  c2?r(); c1!p()
||  r::  c1?s(); c2!g()
||  s::  c1!r()
oc.
```

弱い解釈のもとでは、まず r と s の通信がおこり、次に q と r の通信がおこり、最後に p と q の通信がおこって、p, q, r, s は停止するであろう（図5）*。ところが、強い

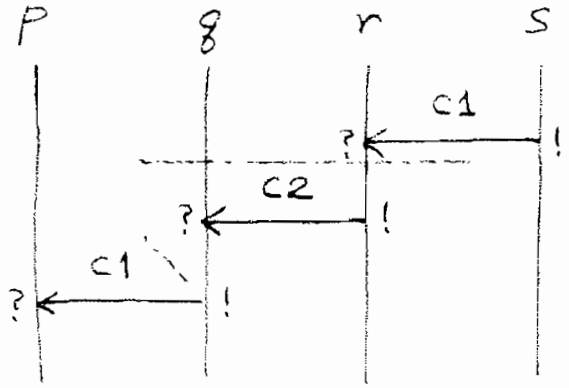


図5　弱い解釈のもとでの通信の様子

解釈のもとでは、r と s の通信と、p と q の通信とは独立におこりえないので、デッドロックが生ずる。

---

*　ただし、弱い解釈は強い解釈をも含むものと考えるべきであり、通信が図5のようにおこなわれることは保証されない。

Date

上のようなプログラムは もともと意味の定義されたものとみなすべきではないから、デッドロックがおこっても、それはプログラマの責任であるが、それにしても強い解釈を支持すべき理由はないであろう。強い解釈に従うと、上の例の場合、路1を用いて通信するとき各プロセスは他の3つのプロセスが路1に関して通信準備状態にあるかどうかをしらべなければならないが、弱い解釈に従うと、各プロセスは直接の相手プロセスの状態をしらべるだけでよい。弱い解釈の方が手間がかからないのである。

3.1.6.2. 通信の成立の判定法

多対多の通信と1対1のそれとの主なちがいは、通信が成立したかどうかの判定法にある。すなわち、3.1.5.1 のアルゴリズムの 3. の部分が複雑になる点である。アルゴリズムの 1., 2., 5., 6. は全く変わらないし、4. は入力すべき値をよみだすCBが複数になるということ

だけが異なる。

通信の成立を判定するには、すべての相手プロセスが通信準備状態になることをみれば十分であることは明らかだが、調べる相手プロセスの数を減らせないだろうか。この問題について考えてみよう。*

定義　路のもとのプロセスの連結性

路Cを用いて通信するプロセスの集合をPとするとき、Pの各要素を点であらわし、Pの任意の要素pとqについて、pの中の路Cを用いる入出力命令がqを相手プロセスとするパラメタ列をもつとき、pとqを結ぶ枝をかく。こうしてできたグラフにおいて、Pの任意の要素r, sが(非)連結のとき、路Cのもとでr, sは(非)連結であるという。■

* いまのところ、例外条件の判定に関してはあまり考察をしていない。ここでも、相手プロセスが停止していない場合に限って考える。

強い解釈に関しては次のことがいえる。

定理

強い解釈のもとでは、プロセスpが路Cのもとでの通信の成立を判定するための必要十分条件は、Cを用いる他のすべてのプロセスが通信準備状態にあることである。

[証明]

十分条件であることは明らかであろう。従って必要であることだけを示す。

路Cを用いて通信するプロセス p, q があるとする。
路Cを用いて通信するプロセスのうち、pを除いたすべてのプロセスの集合をPとする。

もしpが、Pの要素rが通信準備状態にあるかどうかをしらべないものとすると、rを除くPのすべての要素が通信準備状態になったとき、pは先へ進むことになる。ところが、rが通信準備状態になるまでPの要素は通信にはいれないくはいってはならないから、その状態

が続いている間に、pは次の、このもとの入出力命令の実行にはいることがありうる。この場合、pは再び通信をおこないうるから、qが1回入出力をする間にpがちょうど1回入出力をすることは保証されないことになる。従って、pはPのすべての要素の状態をしらべなければならない。■

弱い解釈に関しては次のことがいえる。

定理

弱い解釈のもとでは、路このもとでプロセスPが通信の成立を判定するための必要十分条件は、pと連結なすべてのプロセスが通信準備状態にあることである。

[証明]

十分性は明らかであろう。必要性は、前の定理の証明のPを、路このもとでpと連結なすべてのプロセスの集合（ただしP ∌ p）とすることで示される。*■

---

* rが通信準備状態になるまでPの要素が通信にはいれないことは、通信の条件②から、「プロセスrが（続く）

上の定理からわかることは、一度に多くのプロセスと通信することは、我々が決定したやりかたで"implement"する限りでは必ずしも効率は良くない、ということである。たとえば、次のようなプログラムを考えてみよう。

(脚注の続き) 路Cを用いて1回入出力をおこなう間に、Cと連結なすべてのプロセスも、Cを用いてちょうど"1回入出力をおこなう」という命題がえられるが、この命題からわかる。

A) sc channel c;
g::forward || r::forward || s::forward || t::forward
|| u::forward

|| p:: ... c!g(e)?u(v); ...
|| g:: ... c!r(e)?p(v); ...
|| r:: ... c!s(e)?g(v); ...
|| s:: ... c!t(e)?r(v); ...
|| t:: ... c!u(e)?s(v); ...
|| u:: ... c!p(e)?t(v); ...

cs

B) sc channel c1, c2, c3, c4, c5, c6;
g::forward || r::forward || s::forward || t::forward
|| u::forward

|| p:: ... c1!g(e); c2?u(v); ...
|| g:: ... c1?p(v); c4!r(e); ...
|| r:: ... c3!s(e); c4?g(v); ...
|| s:: ... c3?r(v); c6!t(e); ...
|| t:: ... c5!u(e); c6?s(v); ...
|| u:: ... c5?t(v); c2!p(e); ...

cs.

各プロセスは となりのプロセスとだけ入出力をする。

A)では通信は1回ですむが、各プロセスは通信の成立を判定するために他の5つのプロセスの状態をしらべなければならない。 B)では通信は2回おこなわれるが、通信の成立はそれぞれ1つのプロセスの状態を調べるだ

けで判定できる。さらにプロセスの数nが大きくなれば、判定にかかる手間はA)のやりかたではO(n)であるのに対し、B)のやりかたではO(1)である。

3.1.7. M³における各機能の実現

M³-system を構成する各 member processor の状態遷移図は、通信という観点からみると 図6 のようになる。

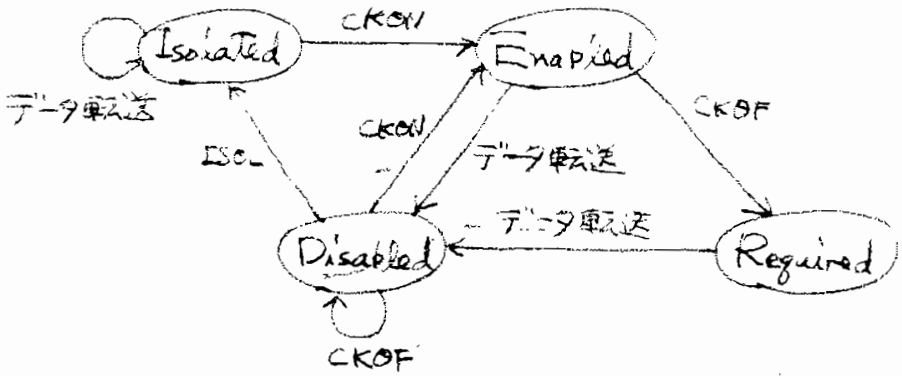


図6

M³の member processon の状態遷移図

Isolated を E状態のかわり、Enabled を W状態のかわりに使うとうまくいくようにみえる。しかし、すべての処理装置が Enabled になるとただちにデータ転送がおこってしまうなど、本来とはちがう点がある。また、host processon の

図7

状態遷移図は これとちがっていて、 おおそ のようで

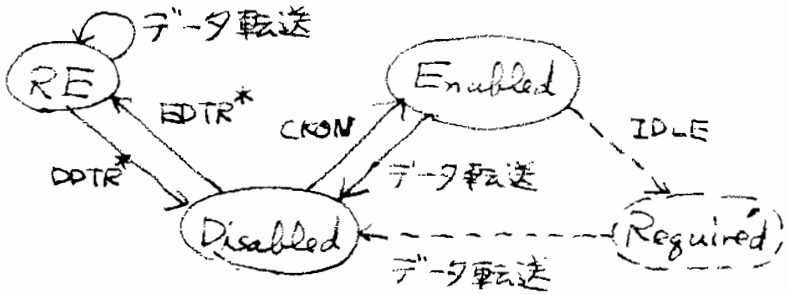


図7

M³の host processor の状態遷移図

ある。 hostには membersにおけるCKOFに対応する命令、はなく、特定の状況のもとで これの代用となるIDLEを用いることはできるが、満足すべき策ではない。また、Isolated がないかわりに RE がある。

これらの状態遷移図と、あるべき遷移図の完全な対応をつけることは不可能であるが、あるべき機能をsimulateする場合に、2とおりの解釈が可能である。いずれにしても D状態、R状態には Disabled, Required(*) を対応させる。

---

* これらの命令(マクロ)の名称は公式のものではない。

3.1.7.1. Enabledを E状態とみなす場合

ハードウェアを次のように変更する必要がある。まず、hostのCKOF命令を他の処理装置と同様に扱うようにすることが必要である。それから、データ転送がおこる条件を次のようにかえる。要求されている条件は $\bigwedge_i \overline{D_i} \wedge (\bigvee_i R_i)$ であったが、現在のMBでは $\bigvee_i R_i$ が定まっている（すなわち、$\bigwedge_i \overline{D_i}$*）。従って、これを加えればよい。現在と同じ使いかたもできるようにするためには、mode switchを作って、それでデータ転送の条件を変えればよい。

また、E状態でデータ転送がおこったとき状態は変化しないべきなのにD状態になるという問題があるが、これを解決するには、ハードウェアでE状態とN状態を区別して、E状態でデータ転送がおこっても状態変化がおこらないようにする方法と、わりこみを用いる方法とがある。わりこみを用いる方法とは、次のようなものである。

---

* Isolatedなどの状態を考慮に入れるともっと複雑になる。

M3ではデータ転送が終わったあとわりこみがかかるが、E状態のときはこれを受けつけてCKONを実行した後、わりこみ前の点から実行を再開し、R状態および準W状態（CPUがidleでないところが本来のW状態と異なる）のときはわりこみをおさえこんでしまえばよい。D状態のときはデータ転送はおこらないから、このわりこみもかからない。この場合、通信のアルゴリズム(3.1.5.1)のEは次のように変えられる。ただし、EC'は準W状態への遷移命令とする。

3.″ EC';

while COMP_CB.channel_nr ≠ CHANNEL_NR do od

3'とちがって、DCは不要であるしデータ転送がおこるとD状態になるから)。

Enabled状態ではCBは常に他の処理装置から(データ転送によって)よみだされるし、CKOFは本来のRCとちがって、路番号のかきこみをおこなわないから、路番号をかきこむ前にデータ転送を禁止しなければならない。

Date

従ってDC(Disable Communication)が必要である。

ハードウェアの変更をしなくても、効率の低下に目をつぶるならば、ほとんどの問題点は解決することができる。しかし、最後の点(DCが欠けていること)だけはなんともしがたいように思われる。

3.1.7.2. Isolated を巨状態とみなす場合

まず"member processors について考える。ECとしてはISOLを実行するだけでたりる(ISOLの実行後次の番地へ戻るようにしておくことが必要だが)。RCとしては、路番号をかきこんだ後、CKON;CKOFを実行すればよい。* しかし、WCはハードウエアを変えなければ正しくimplementすることはできない。Isolated状態では処理装置はデータ転送に加わらないが、これに加わるほかはIsolatedと

---

* CKONを出すまでデータ転送のときは0が送り出されるようになっているから DC は必要ない。 [3.1.2参照]

同じであるような状態をつくって、これをWに対応させることが必要である。ハードウェアを変更しないとすると、次のようにすることになるだろう。W状態は、Isolatedであって、かつデータ転送後のわりこみを受けつける状態、とする（E状態ではわりこみは禁止されている）。データ転送は必ず2回1組でおこなって、最初の転送後にわりこみがかかると、次のデータ転送には参加する。

しかし、host processonにはIsolated状態はなく、データ転送に加わらないということはハードウェアの上からできないようになっている。従って、hosには特別の方法を用いるしかない。第7図の状態REをE状態に対応させることになるだろう（3.1.5.1の脚注の図と図7を比べてみるとよい）。

3.1.7.3. 2つの方法の比較

あらゆる点で3.1.7.1の方法のほうがおおげさで、利点が少ない。ハードウェアを変更するにしろ、しないにしろ、3.1.7.2の方法のほうが良いであろう。

3.2. 入出力パラメタ列の対応を検証する方法

パラメタ列の対応則」とその意味に関しては「第1部付録2」で述べた。パラメタ列」の対応がついたあとは、ルーティンのパラメタと同じように処理すればよいから、ここではパラメタ列が過不足なくあることを検証するにはどうしたらよいか、という問題だけを考える。

パラメタ列の対応を検証するアルゴリズムは次のとおりである。

1. その入出力命令が使う路は、その命令を含むプロセスの中ではじめて使われるものかどうかをしらべる；

A) はじめて使われる路の場合は、登録されていないパラメタ列がわらわれても、次の条件をみたしていれば誤まりとせず、登録する。

条件: 相手プロセスがまだ定義されていないか、定義されていても、その中に、その路を用いる入出力命令があらわれていないこと。

B) すでに使われた路の場合は、登録されていないパラメタ列があらわれると誤まりとする。

2. 登録されたパラメタ列のうち、送信者または受信者がその命令を含むプロセスであるようなもののうち、その命令にあらわれないものがあれば誤まりとする。■

このアルゴリズムを用いると、パラメタ列の対応は次のようにしてたしかめられる。

① あるプロセスの中ではじめて路cが使われたとき、その入出力命令に誤まりがなければ 1.A) によってパラメタ列が登録され、路cがそのプロセス内で再び使われたときには、入出力命令の同型[第1部付録3]がたしかめられる。

② 以下の考察では、各プロセスの、路cを用いる最初の入出力命令だけについて考える。

プロセスq内の路Cを用いる最初の入出力命令Cのパラメタ列に過不足があるとき、相手プロセスをpとすると、4つの場合にわけることができる。

(1) pはqより前で定義されている(すなわちブロックをともなったpの宣言がqの宣言より前にある)。そして、p内の路Cを用いる入出力命令がもつ、相手プロセスをqとする(入力または出力)パラメタ列に対応するパラメタ列がCにはない。

(2) pはqよりあとで定義されている(すなわち、qの宣言より前にはpのforward宣言だけがある)。それ以外は(1)と同じ。

(3) pはqより前で定義されている。そして、Cがもつ、相手プロセスをpとする(入力または出力)パラメタ列に対応するパラメタ列が、p内の路Cを用いる入出力命令にはない。

(4) pはqよりあとで定義されている。それ以外は(3)と同じ。■

(1)および(3)の場合は、誤りはCをコンパイルするときに検出される。(1)の場合はアルゴリズムの2.で、(3)の場合は1.A)でわかる。(2)および(4)の場合は、誤りはCのコンパイル時には検出されない。あとでp内のeを用いる最初の入出力命令があらわれたところで、それぞれ(3)および(1)と同じようにして検出される。

4. 並列命令

4.1. 処理装置のわりあて

まず、各プロセスに どのように処理装置をわりあてるか を決めなければならない。 その際に、次のことを原則とする。

原則： 共時に走行しうるプロセスには 同じ処理装置を わりあてない。■

(この原則は、あるプロセスpからわかれてできたプロセスの資源は、pの資源(処理装置)の部分集合とするという資源保護規則[Br73]に合ったものである。)

次のようなプログラムに、この原則を適用してみよう。

```
co
  p :: ---
       co
         p1 :: ---
       || p2 :: ---
       oc
|| q :: ---
oc.
```

ここで、pとqは共時に走る。また、p1, p2はpと共時に走ることはないが、qとは共時に走る。従って、p1、p2のうちいずれかはpと同じ処理装置をわりあてるこ

とができるが、 p1, p2, q には、それぞれ別の処理装置をわりあてなければならない。

上のような原則をおいたのは、implementation を容易で効率的なものにするためである。たとえば上のプログラムで p と p1, q と p2 にそれぞれ同じ処理装置をわりあてたとしよう。 p と q とは、p が p1 にわかれる前に1回、再び p にまとまったあとに1回入出力をするものとする。この場合、q を実行する処理装置は、q の実行半ばで p2 を実行し、(図8参照) また q の実行を続けなければならない。このような context switching をさけるためには上の原則が必要になるのである。

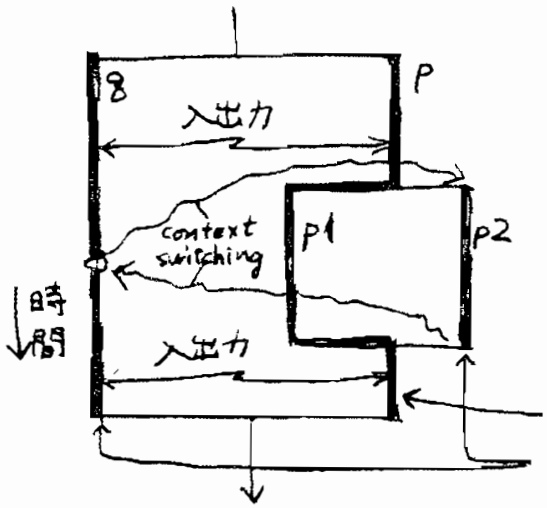


図8 原則に従わないわりつけ。context switching が不可欠である。

わりあてのアルゴリズムとしては、次のようなものが最もかんたんであろう。このアルゴリズムでは、処理装置に0からはじまる番号をつけ、0から順にわりつけていく。

```
module  processor scheduler;

var  lev : 0.. maxlev { 並列命令の入れ子の深さ };
     serial, parallel : array 0.. maxlev of 0.. maxprocessor;
     processor of new process : 0.. maxprocessor; { 求める番号 }

procedure enterpc;
 { 並列命令にはいったとき実行する. }
 begin
    lev := lev + 1;
    parallel [lev] := serial [lev - 1];
      { serial [lev]      はまだ定義されていない. }
 end;

procedure newprocess;
 { 新しいプロセスがあらわれたとき(すなわち. プロ }
 { セス名札があらわれたとき)実行する。          }
 { ただし、並列命令の最初のプロセスだけは       }
 { first process を実行する.                    }
  begin
     parallel [lev] := parallel [lev] + 1;
     processor of new process := parallel [lev];
     serial [lev] := parallel [lev]
  end;
```

```
procedure firstprocess;
  {並列命令の最初のプロセスがあらわれたとき実行する.}
  begin
    processor of new process := parallel[lev];
    serial[lev] := parallel[lev]
  end;

procedure exitpc;
  {並列命令から出るとき実行する.                    }
  begin
    lev := lev - 1;
    parallel[lev] := max(parallel[lev], parallel[lev+1])
  end;

begin {initialization}
  lev := 0;
  serial[lev] := 0; parallel[lev] := 0
end processor scheduler.
```

レベル i の並列命令を処理しているとき、parallel[i] は、処理中のプロセスおよびその中のすでにあらわれたプロセスと共時に走行しうるプロセスが次にあらわれるとき、それにわりあてるべき処理装置の番号－1を値とする。serial[i] は、処理中のプロセスおよびその中のすでにあらわれたプロセスと共時に存在しえないプロセスが次にあらわれるとき、それにわりあてるべき処理装置の番号－1を値とする。

もう少しくわしく説明しよう。

逐次的にあらわれる2つの並列命令に含まれるプロセスは共時に存在しえない。従って、ひとつの並列命令 P（そのレベルをiとする）があると、それに対し逐次的にあらわれる並列命令が使うことができる最小の処理装置の番号は、Pの前にあけるそれと等しい。従って、その値は、Pを含むプロセスpがあらわれたとき serial[i] に入れておき、pがでおわるまで保存しておけばよい。pよりあとであらわれる、pと並列なプロセスの中にあらわれるプロセスは、pおよびpの中のプロセスと共時に存在しうるから、pの中でプロセスにわりあてられた最大の処理装置の番号Mを求めて、それを parallel[i] に入れておき、parallel[i]+1 からわりあてればよい。Mを求めるには、pの中の1つの並列命令（Pとする）があるたびに、Pの中でプロセスにわりあてられた最大の処理装置の番号すなわち parallel[i+1] と、parallel[i] のうち小さくない方を parallel[i] に代入すればよい。

これらをアルゴリズムとして記述すると、上のようになるわけである。

このアルゴリズムは、次のようにかきかえることもできる。

```
        ⋮
{ ここまでもとのアルゴリズムと同じ }

procedure enterpc;

  begin
    lev := lev + 1;
    parallel[lev] := serial[lev-1] - 1;
  end;

procedure newprocess;
{ 並列命令の最初のプロセスも含めて、新しいプロ }
{ セスがあらわれたとき実行する。              }
  begin
    parallel[lev] := parallel[lev] + 1;
    processor of new process := parallel[lev];
    serial[lev] := parallel[lev]
  end;

procedure exitpc;

  { 以下は もとのアルゴリズムと同じ }
        ⋮
```

上記のアルゴリズムは、いずれも再帰を使わない場合のものだが、Pascalのような言語でコンパイラをかくときには、言語のそなえたスタック機構を利用しておく（serial, parallel

を配列としない) ことができる。

はじめのアルゴリズムを適用した例を注す。注釈 {i, j, k} は lev = i, serial[i] = j, parallel[i] = k であることをあらわす。 — は値が定義されていないことをもらわす。

```
— {0,0,0}
co {1,—,0}
  p:: {1,0,0}   { p's processor = 0 }
    co {2,—,0}
      p1:: {2,0,0}   { p1's processor = 0 }
    || p2:: {2,1,1}   { p2's processor = 1 }
    oc ; {1,0,1}

    co {2,—,0}
      p3:: {2,0,0}   { p3's processor = 0 }
      p4:: {2,1,1}   { p4's processor = 1 }
    oc ; {1,0,1}

 || q :: {1,2,2}   { q's processor = 2 }
oc {0,0,2}.
```

このアルゴリズムで処理装置のわりあてをおこなうことを前提として、次の用語を定義する。

定義　親処理装置と子処理装置

並列命令を含むプロセスに ありあてられた処理装置を、その並列命令の親（処理装置）、その並列命令を実行する他の処理装置を、その並列命令の子（処理装置）とよぶ。■

定義　主処理装置

プログラムの要素であるブロック（すなわち最も外側のブロック）を実行する処理装置（すなわち、最も外のレベルの並列命令の親）を主処理装置という。■

4.2. 並列命令の展開

プロセスの数は並列命令の実行により増減する。しかし、処理装置の数は一定であり、常に存在しつづける。従って、複数の並列命令をもつプログラムを、プログラム全体が(大域的宣言と最も外側のブロックの begin, end を除くと)1つの並列命令からなるような同一機能をもつプログラムでおきかえることができれば、並列命令の implementation のおおすじが決まったことになる。4.1で述べた処理装置のわりあてはその方法の一部を示したことになるが、おきかえを完成させるには、1つの処理装置にわりあてられた複数のプロセスをどうやってつなぎあわせるかという問題が残っている。

厳密にはこのようなおきかえはできないが、プロセスの停止の問題(停止していれば通信が失敗すること)を除けば、おきかえは可能である。ここでは、おきかえのしかたを一般的に述べることはしないが、例をあげて説明することにしよう。

Date . . . . . .

4.2.1. 輸入がない場合

```
begin   CS01;
  co   p1 :: CS1   ||   p2 :: CS2   oc;
  CS02;
  co   p3 :: CS3   ||   p4 :: CS4   oc;
  CS03
end.
```

上のプログラムは、4.1の方法によれば、主プロセス、p1, p3 を処理装置0（主処理装置）に、p2, p4 を処理装置1にわりあてることになる。このとき、このプログラムは、次のような Dihybrid 風のプログラムで近似される。

```
begin
  co channel  fork1, fork2, join1, join2;

    processor0 ::  CS01;
        fork1!processor1(); CS1; join?processor1();
        CS02;
        fork2!processor1(); CS3; join?processor1();
        CS03
  || processor1 ::
        do  fork1?processor0() → CS2; join1!processor0()
        □   fork2?processor0() → CS4; join2!processor0()
        od

  oc
end.
```

もう1つ例をあげよう。

```
begin  CS1;
  co
   p1 :: CS11;
      co  p11 :: CS111 || p12 :: CS121 oc;
      CS12
  || p2 :: CS21;
      co  p21 :: CS211 || p22 :: CS221 oc;
      CS22
  oc
end.
```

上のプログラムは次のようにかきかえられる。

```
begin
  co  channel  fork1, fork2, fork3, join1, join2, join3;

    processor0 :: CS1;
        fork1! processor2(); CS11;
          fork2! processor1(); CS111; join2?processor1();
          CS12;
        join1?processor2()
  || processor1 ::
    do fork2?processor0()→ CS121; join2! processor0() od
  || processor2 ::
    do fork1?processor0()→ CS21;
          fork3! processor3(); CS211; join3?processor3();
          CS22; join1! processor0()  od
  || processor3 ::
    do fork3?processor2()→ CS221; join3!processor2() od

  oc
end.
```

4.2.2. 輸入がある場合.

```
begin var x:T; CS01;
  co imports x;
    p1:: define x; CS1 || p2:: use x; CS2
  oc;
  CS02;
  co imports x;
    p3:: use x; CS3 || p4:: define x; CS4
  oc
  CS03;
  co imports x;
    p5:: use x; CS5 || p6:: use x; CS6
  oc;
  CS04
end.
```

上のプログラムは、輸入をすること以外は、4.2.1のはじめの例と似ている。このプログラムは次のようにかきなおされる。

```
begin
 co channel fork1, fork2, fork3, join1, join2, join3;

  processor0 :: var x:T; CS01;
    fork1! processor1(x); CS1; join1? processor1();
    CS02;
    fork2! processor1(x); CS3; join2? processor1(x);
    CS03;
    fork3! processor1(x);
    begin var xtemp:T; xtemp:=x;
      CS5'
    end;
    fork3? processor1();
 || processor1 ::
    do var x:T; fork1? processor0(x) →
          CS2; join1! processor0()
    □ var x:T; fork2? processor0(x) →
          CS4; join2! processor0(x)
    □ var x:T; fork3? processor0(x) →
          CS6; join3! processor0()
    od

 oc
end.
```

ただし、CS5' は、CS5 にあらわれる名前 x をすべて xtemp におきかえたものとする。CS1, CS3 は次のような命令で おきかえてもよい。

```
begin var xtemp:T; xtemp:=x;
  CS1'; x:=xtemp
end
```

begin var xtemp : T; xtemp := x;
  CS3'
end.

CS1', CS3'の意味は CS5'と同様である。

4.3. 疑似入出力命令のimplementation

並列命令を展開したときあらわれた「入出力命令」は、他の入出力命令が正しく機能するためには、普通の入出力命令とは違ったやりかたで"implement"しなければ"ならない。また、もしこれらの疑似入出力命令を、普通の入出力命令と同じようにimplementすると、N個のプロセスをもつ並列命令を実行するのに、その中の入出力命令による分を除いて、データ転送がN～2N回(これにはもっと少ない回数ですむこともあるが)、相手プロセスの状態判定が2N(N-1)回以上必要になる。* しかし、並列命令の持

* データ転送は、並列命令の開始時に1～N回かかる。各プロセスがそろって通信準備状態にはいれば"1回で"すむが、すべてずれるとN回かかる(開始時には各プロセスの足並みは比較的そろっているであろう)。また、終了時にはそろっていることは期待できないから、Nが充分大きくない限り、ほぼN回かかるであろう。相手プロセスの状態判定は、開始時、終了時に各プロセスともN-1回おこな(続く)

殊性を考えると、この回数をもっと減らせるのではないだろうか。その問題について考えてみよう。

4.3.1. 片道通信2回ですむか？

最低限次のことが必要である。並列命令の開始は親によって司令されるから、開始時に少なくとも1回、親から子への片道通信が必要である（入出力命令は"両道"通信——通信要求を双方がだす——であった）。それぞれの子は、並列命令の開始を知るため、少なくとも1回、親の状態を判定しなければならない。また、並列命令の終了は、親およびすべての子の停止によって判定されるから、すべての子から親への片道通信が必要である。ただし、子供の中にリーダーをつくって、それを経て親へ報告することは考えられる。ここでは、親がすべての子供の状態を自ら監視する場合に限って考えよう。その場合

（注釈の続き）うから、全部で2N(N-1)回以上となる。

には、並列命令の終了を判定するのに、親は少なくとも N−1 回の状態判定を必要とする。

以上をあわせると、少なくとも 約N回のデータ転送と、2(N−1)回の状態判定が必要である。

今示した最低限の転送と判定とで、並列命令を正しく implement することが可能であろうか。実は、これではうまくいかない。そのことを示そう。

さて、親は各並列命令に対応する論理型変数をCB上にもっているとしよう。問題にしている並列命令に対応するその変数をPCとする。その並列命令を実行するときは pc = true とし、その並列命令の外にあるときは常に pc = false であるようにする。これより、その並列命令の外を実行するときの状態を $\overline{PC}$ で、その並列命令を実行するときの状態を PC であらわす。

それぞれの子は、running（並列命令実行中）であるか、terminated（並列命令を実行していない）であるかのいずれかの状態にあり、その状態はCBの上に示されるとする。

並列命令の開始前は、(親は) PC であり、(子は) terminated である。親が PC になると、子供はそれをみて、(ややおくれて) running になる。子供は課せられた仕事が終わると再び terminated になり、親は自分の仕事がおわり、かつすべての子が terminated になると PC になる。すじがきは以上のとおりだが、はたして常にこのようにうまくいくであろうか。

親が PC になったときすべての子がただちに running になり子が terminated になったとき、親がただちに PC になることが保証されていれば、この方法でもうまくいく。しかし、特にあとの条件は、各プロセスに課せられた仕事が平等でないときにはみたすことができない。

条件がみたされないと、どのようなことがおこるか考えてみよう。簡単のため、子が1台の場合を考える。注目する並列命令に関する状態は、親子の状態の組であらわされる。並列命令が正しく実行されれば、すでに述べたように、状態は次のように遷移する(図 9A 参照)。

1. ($\overline{PC}$, terminated)
↓
2. (PC, terminated)
↓
3. (PC, running)
↓
4. (PC, terminated)
↓
5. ($\overline{PC}$, terminated)

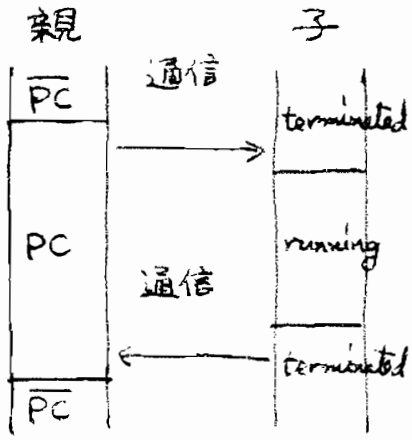


図9A 正しい実行

ここで問題になるのは、第2の状態と第4の状態が区別できないことである。そのため、第3の状態を経ずにもとの状態にもどってしまったり(図9B)、第4の状態と第3の状態を何度も行ったり来たりする(図9C)ことがおこりうる。後者の場合は子がむだな仕事をするだけで無害ともいえるが、前者の場合は致命的である。

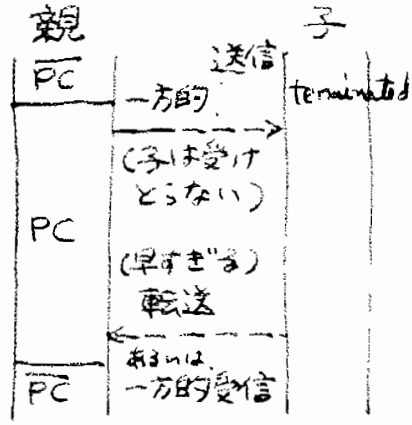


図9B 子が停止したままの場合

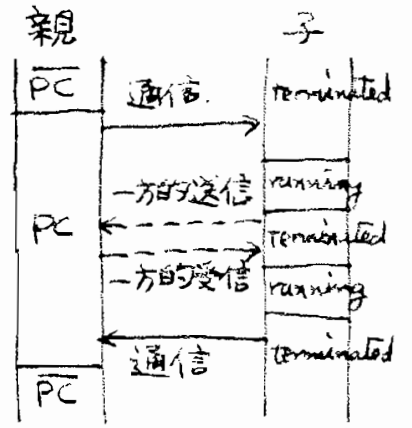


図9C 子が同じ仕事をくりかえし実行する場合

この事態がPCのかわりに入出力命令の路番号を代入する場所や、他の型の変数をもってくるなどしても改善されないことは明らかであろう。子についてもまた同様である。

4.3.2. 片道通信を3回おこなう方法.

片道通信が親から子へ1回、子供から親へ1回ではうまくいかないことがわかったから、更に1回おこなう方法を考えてみよう。最初の通信は、並列命令の開始を司令する、親から子へのものであること、また、子から親への通信が少なくとも1回あることを仮定すると、考えうる形は3通りある（図10）。しかし、このうち B), C) は図11のような誤りが生じうるので、可能なのは A) だけである。

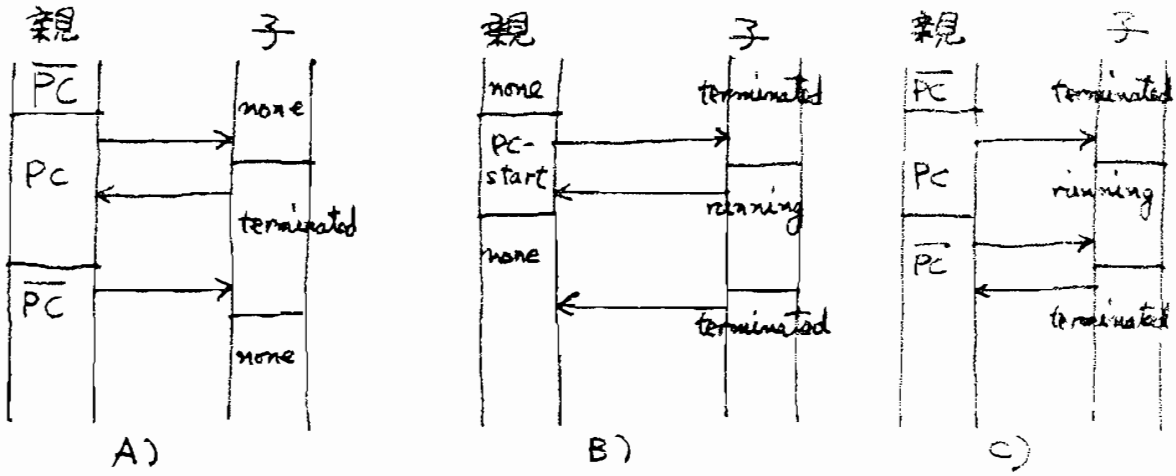


図10

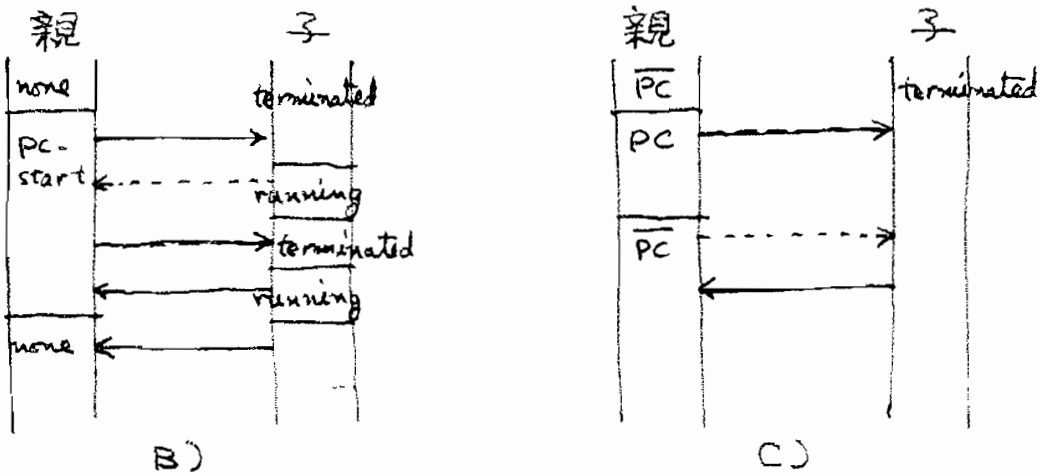


図11　図10 B), C) でおこりうる誤り

5. コンパイラについて.

コンパイラと連係子の機能

Dihybrid のコンパイラは次のような形につくるのが標準的であろう。

Pass 1 : マクロおよび準マクロ の展開

Pass 2 : 構文および意味の解析

Pass 3 : コード生成

Pass 1 の入力はいうまでもなく Dihybrid の原始プログラムであるが、Pass 1 の出力としてどのような形のものを出すかについては選択の余地がある。語彙解析をした結果を出力することもできるし、原始プログラムの形で出力することもできる。しかし、Pass 1 ではプログラムの誤りをほとんど検出できないので、原始プログラムの形で出力した方が、Pass 2 で検出された誤りをもとのプログラム上に位置づけるのに楽であろう。従って、Pass 1 の出力および Pass 2 の入力は、原始プログラムとしたい(ただし、Pass 1 の入力と出力との cross reference をなんらかの形で Pass 2 にわたすのが

望ましい)。しかし、展開をおこなうためには、ある程度の構文解析と定数宣言の処理、定数式の評価が必要である。*

Pass 2の出力は中間コードであり、このimplementationでは、それほどの最適化をおこなうつもりはないので、逆ポーランド型のコードを用いた。

Pass 3の出力はRelocatable Object Moduleであり、それも我々で独自の仕様を決めた(M⁴コンパイラ[Id79]は同じ仕様のROMを出力する)。

このため、Pass 3の出力を絶対コードに変換する連係子が必要であり、その出力はTI形式のLoad Moduleを拡張したものである。システムに依存する部分はなるべくこの連係子以降の段階におさめるようにした。また、この連係子は前方参照の解決をおこなう。

プログラムを実行するには、原始プログラムをコンパイラにかけ、その出力を連係子にかけたあと、その出力を、M³-systemのLoaderでloadしなければならない。

* コンパイラの負担を減らすには、(準)マクロ展開の (続く)

（注釈の続き）　ために評価することが必要な定式はすべて符号なし定数に限り、かつ定数名のときは大域的定数に限るという方法が考えられる。

6. 結び

implementation は現在進行中だが、これまでにPass2の大半とPass3の一部ができている。今、結論を出すことはできないが、これまでにわかったことをまとめておく。

implementation は当初の予想よりはるかに困難であることがわかった。わずかと思えるハードウェアの制約や奇妙さが、容易で効率的なimplementationを拒否する。* そのためにふりだしに戻ることがしばしばだった。これは（Hoareが述べているように）Dinybrid のような言語をあらかじめ与えられたハードウェアの上にimplementすることが困難だということを示すとともに、ハードウェアの設計の難しさをも示していると考えられる。

---

* たとえば、hostがmembersと同じ意味のCKOFやISOLをもたないこと、データ転送後のわりこみは、マスクされていると命令を1つ実行するだけでその要求が消されてしまうという事情があるにもかかわらず、優先度が低いことなど。

文献

[Br73] → 第2章の文献表

[Hi78] → ――――――

[Id79] 井出一郎： マルチコンピュータシステムの
ためのコンパイラの設計と試作，
東京大学工学部計数工学科卒業論文〔1979〕

[Wa78] → 第2章の文献表

([IH*76] 石畑清，正田輝雄： Bootstrapping PASCAL
using a Trunk, Technical Report 76-04,
東京大学理学部情報科学科)

付録. TM-990 における stack の構造

TM-990 への implementation の ひとつの特徴は、その stack の構造にある。従って、それについて述べることにしよう。**)

TM-990 というCPU*は、メモリの中に16個の汎用レジスタ（R0～R15）をもっていて、それをWP（work space pointer）とよばれるレジスタ（CPUの中にある）がさしている。命令BLWP（branch and link WP ≒ call）は WPの値をR13へ、PC（program counter）の値をR14へ、そしてST（status register）の値をR15へ格納した後、callすべき手続きへ飛び、RTWP（return WP）はその逆の操作をして戻る。このような TM-990の特徴を考えると、レジスタをstackの上にとるのが簡単で、効率も悪くないと考えられる。その考えに従って、stackの構造は図1のようにした。メモリ（RAM）の終わりをstackの底とし、そこからルーティンの外の変数をとる。

＊ チップの名称は TMS-9900 である。

（＊ PascalのHITAC8800/8700へのimplementation [IH*76] などと比べると、このimplementationの特徴がよくわかるであろう。）

その上にはルーティンの外を実行するときのレジスタをとる。
1つのルーティンを活性化することによってできる領域は、下からパラメタ、局所変数(これらはレジスタの外にある)、古いST, PC, WPの値(すなわちR15~R13)、stack pointer(R12)、expression stackというようになっている。
expression stackはR10(またはR9)からはじまり、複雑な式の場合にはR0をこえて上に伸びうる(R11とR10)は一時的な使用のためにとっておかれる。
ルーティンをよぶときは、expression stackの上端を新たなパラメタ領域の下端として、今述べたのと同じような領域をとる。ルーティンのパラメタは、他の式を評価するときと同じやりかたで扱うことができる(格納の必要がない)。また、パラメタ領域の下端に関数の値を入れるようにしておけば、関数から帰還するとき、その値を動かす必要はない(従って、手続からと関数からの帰還のしかたは同じでよい)。
この方法の欠点は、式が複雑になってレジスタをこ

えると、コードが急に高価につくようになるということである。

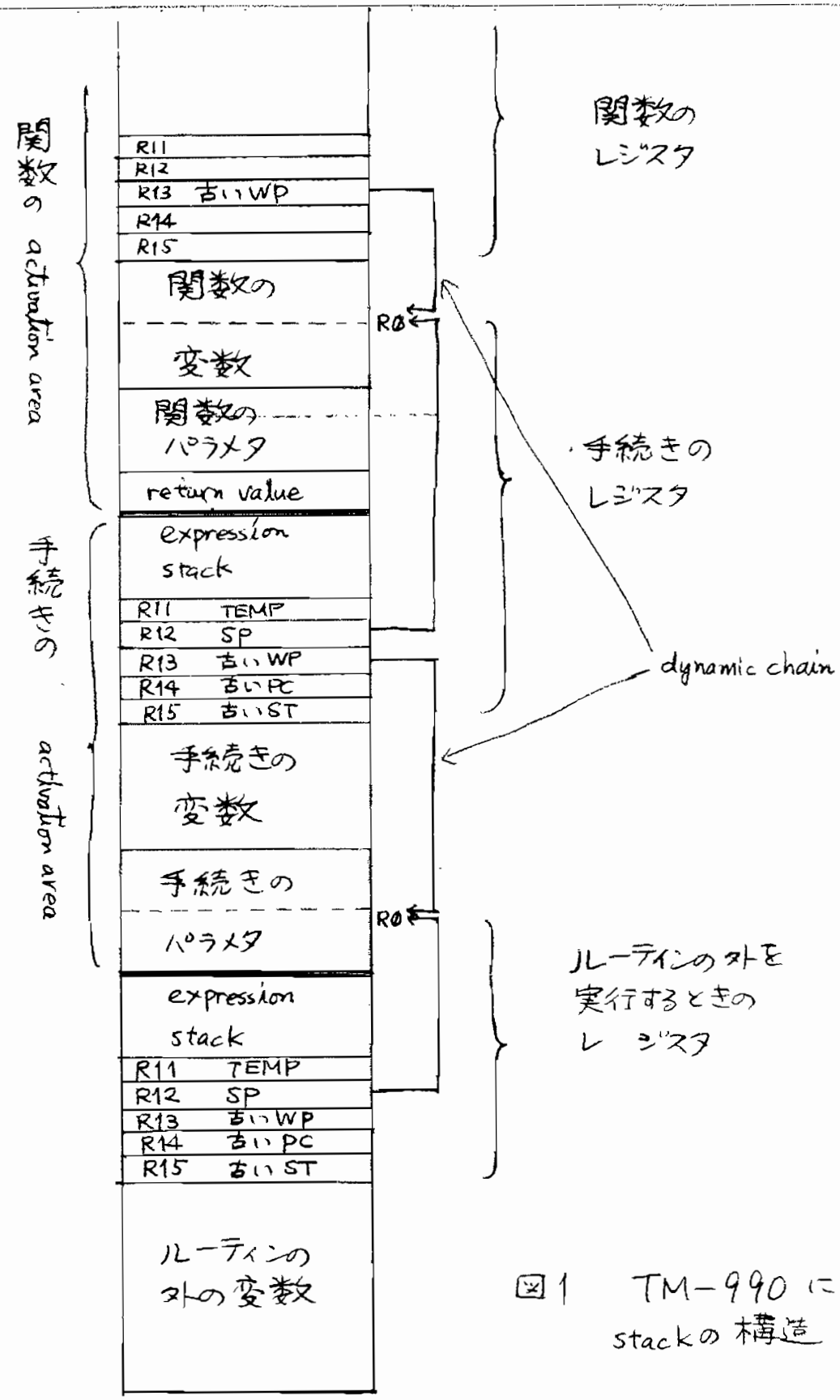


図1 TM-990 における stackの構造

最後に

multi computer system のための言語 および その処理系に関する研究は まだ はじまったばかりである。[Ho78] や [Br78] で 言語案が 示されたが、implementation はもちろん、完全な言語も発表されていない。 こういう情況のもとで 卒業研究として このテーマをとりあげたことは 時期尚早だったと思われる。もともとコンパイラの完成をめざして研究をはじめたが、それが 完成しないばかりでなく、implementation の基本にかかわる点に関しても 充分な根拠を示すことができず、 また おそらく誤まりを 防ぎえなかったと思われる。 Dihybrid じたい、洗練された言語とはいいかねる。 しかし、M³ のような システムの プログラム言語の設計や implementation の一つの方針を示すことはできたと思う。 もし、将来の研究のヒントになれば 幸いである。

謝辞

並列プログラミングについて考える機会と、言語設計の機会を与えてくださった森下巖助教授、Dihybridの設計にあたって貴重な御意見を述べてくださった和田英一教授、和田研究室の戸村哲さん、近山隆さんほかのみなさん、implementationにあたって親身に指導してくださった森下研究室の出口光一郎助手、全般にわたって協力してくれた井出一郎君、そして森下研究室の院生のみなさんに感謝いたします。