SYSPの基本計算機構　－新しい計算機構を求めて－

横井俊夫
（電子技術総合研究所）

1. はじめに

現在、研究開発中のシステムSysp（Structured Symbol System Processor）の基本計算機構を紹介する。Sysp[23),24),25)]の全体像は、Fig.1に鳥瞰するとうりである。本稿は拡張軸にそった紹介である。ここに述べるアイデアは、全てが新らしい着想というわけではない。むしろ、多くをActor（PLASMA[11),12)]）、LAMBDA[18),19),20),21),(注1)]等から受け継ぐ。特に、Actorモデルとその具現化であるPLASMA[(注2)]の着想を追うものである。しかし、PLASMAでの議論が、耳慣れない言葉と、あまりに盛沢山の構想とそれに比べあまりに少ない具体化と、時には細かすぎる形式化へのこだわりが、その基本となる計算の機構を理解するのを妨げていると思われる。勿論、筆者としては、それらの議論も高く評価するものではあるが、応々にしてその煩雑さのため、その有用な考え方の理解を妨げ、様々な誤解を生じているのが現状と思われる。そこで、Actorの良き（?）批判者でもあるLAMBDAの議論をからめつつ、具体的な例を用い、今までの作業の進展を追いながら、平易にSyspの基本計算機構を紹介するのが本稿の目的である。この紹介を通じて、そのメッセージの送受を基本とする計算の機構が、決して奇異なものではなく、現在Lispという形で、一応の完成を見た記号処理計算の体系（真の意味での汎用の計算の体系）を次に、どのように発展させるかの１つのしかし大きな方向づけを与えてくれることが、明らかになるものと思う。紹介の最後に、最近のトピックスである、Logic Programming, Functional ProgrammingとSyspとの関連を述べる。ただし、基本機構のレベルの話しであり、より厳密な議論は別稿[26)]にゆずる。

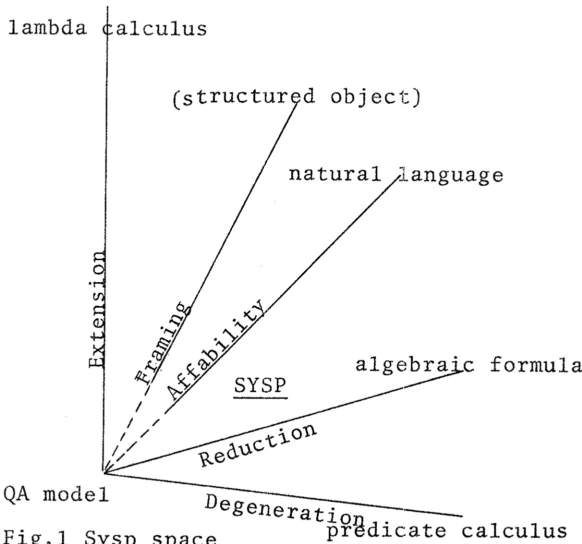


Fig.1 Sysp space

2. Syspの基本計算機構

2.1. λ－計算

関数の計算手順を自然に表現する計算モデルを与える。プログラムもデータもλ－式という高次言語のモデルとしても自然で適切である。その自然さゆえに、種々のプログラミング言語の基本機構となった。

2.2. Lisp

λ－式に名前をつけ、その名前を再帰的に参照で

（注1）メッセージの送受を基本機構として、実用レベルに達していると思われるものにSmall talk[9),13)]がある。この言語、メッセージの形式、送受の方式ともに一般的というわけではないが、うまい枠組に限定し、理解し易い、まとまりの良いシステムとなっている。

（注2）Actorモデルと PLASMAの間には、かなりの厳密な定義があるが、以後ではあまり気にせずに参照する。

きる機能、再帰的関数計算の機能が追加された。名前づけに伴なって、自由変数のscoping, bindingの問題が生ずる。方法は2つである。1つは、文書上の静的な方法と、もう1つは、実行上の動的な方法である。Lispは、伝統的に後者を採用してきた。いずれを用いようとも、計算の機構そのものに本質的な差異を生ずるものではない。具体的な場面では、それぞれ利害得失がある。動的な場合の長所、短所をあげる。

長所：① 関数評価のために作り出される情報は、返り先を示す制御情報と、引数の結びつけの情報から成る。この2つの情報の構造や操作が全く同一となるため、インタプリタが簡潔になる。特に、インタプリタを高速にするための作成技術である浅い結びつけ(注)(shallow binding)を利用できる。

② 自由変数を、その関数が使用される環境の差異を反映するものとして利用できる。

短所：① 自由変数の内容が動的に決定されることは、構造化プログラミングの精神に反する。実際のプログラムのほとんどが、自由変数は、全プログラムに共通に参照されるグローバル変数として扱われている。

② ①の結果として、実際には利用されない結びつけの情報を大量に持ち歩くことになる。

Fig.2 にLispの機能を用いた階乗の関数を示す。(a)が定義で、(b)が(FACT 3)のトレースである。トレース情報の形式は次の通り。

(entering <再帰呼びの深さ><関数名>) <実引数のリスト>
(leaving < 〃 >< 〃 >) <値>

2.3 続き関数[16]、J-操作子[14]

```
(DEFINE  FACT
    (LAMBDA  (N)  (IF  (= N 0)  1
                       (* N (FACT (- N 1))))))
```

(a) Definition

```
(entering  1  FACT)  (3)
(entering  2  FACT)  (2)
(entering  3  FACT)  (1)
(entering  4  FACT)  (0)
 (leaving  3  FACT)  1
(entering  1  TIMES) (1 1)
 (leaving  0  TIMES) 1
 (leaving  2  FACT)  1
(entering  1  TIMES) (2 1)
 (leaving  0  TIMES) 2
 (leaving  1  FACT)  2
(entering  1  TIMES) (3 2)
 (leaving  0  TIMES) 6
 (leaving  0  FACT)  6
```

(b) Trace of (FACT 3)

Fig.2 Factorial function in Lisp

λ計算を、種々のプログラミング言語の意味記述の手段に用いようという試みが行なわれた。λ計算の単純で自然な計算の機構は、少し自然な拡張を施すと、非常に表現能力が高まることに由来した。拡張の1つが、続き関数(continuation function)である。これは、評価手順をはっきり定める等のために導入された。もとの関数f oldは、1つ余分の続きを示す引数が付け加えられ、f newとなる。続きとは1変数の関数である。f newの値は、f oldの値に続き関数を適用したものである。

$$f_{new}(x_1, \cdots, x_n, C) = C(f_{old}(x_1, \cdots, x_n))$$

明きらかに、続き関数は書かれている場所とは、異なった所で実行される。続き関数の定義から文書上の結びつけ

(注) 浅い結びつけで静的結びつけを行なおうという提案[3]もある。

が必要である。そこで、静的な結びつけが採用される。一例をFig.3に示す。Fig.2の階乗の関数をf_{old}としてf_{new}である。(a)に定義、(b)に(FACT 3 (LAMBDA (A) A))のトレースを示す。C_1〜C_4は、それぞれの続き関数に仮につけた名前である。一番右側に、続き関数の自由変数に何が結びつけられているかを示す。静的な結びつけである[注]。トレース(b)を眺めてみると、無意味な"(leaving …)"の羅列があることがわかる。その性質上、続き関数からの返りは、即、その続き関数を含むFACT関数からの返りを意味する。そこで、関数の呼び出しに、別の関数の返りが重なるというような機能を設けた方が自然である。

このような拡張は、まず、λ計算に、名札と飛び越しなどのimperativeな要素を表現できるようにするために導入された。Landinの J-操作子[illegible]やReynoldsの離脱式[16]に代表される。その機能は、LispのERR, ERRSET (THROW, CATCH)の機能を一般化したものである。この離脱式の考え方は、SCHEME等によって、もう少しうまく表現される。まず、離脱用の関数の実行は、その性質上、常にtail(他の関数がその関数の値を引数として実行されることはない。)にある。Fig.3(a)のFACT関数では、続き関数Cの呼び出し、FACTの再帰呼び出し、ともにtailにある。SCHEMEでの考え方とは、tailでの関数呼び出しは、同時に返りの処理も行なわれるとするというものである。つまり、tailでの関数呼び出しは、引数の結びつけをともなった飛び越しとなる。この拡張は、tailにある関数の性質上、本来の計算の意味を少し

```
(DEFINE  FACT
    (LAMBDA  (N  C)
        (IF  (= N 0)  (C 1)
             (FACT (- N 1) (LAMBDA (A) (C (* N A)))))))
```

(a) Definition

```
(entering  1  FACT)  (3  C1)  C1=(LAMBDA (A) A)
(entering  2  FACT)  (2  C2)  C2=(LAMBDA (A) (C (* N A)))  (C=C1, N=3)
(entering  3  FACT)  (1  C3)  C3=(LAMBDA (A) (C (* N A)))  (C=C2, N=2)
(entering  4  FACT)  (0  C4)  C4=(LAMBDA (A) (C (* N A)))  (C=C3, N=1)
(entering  1  C4)    (1)
(entering  1  C3)    (1)
(entering  1  C2)    (2)
(entering  1  C1)    (6)
 (leaving  0  C1)    6
 (leaving  0  C2)    6
 (leaving  0  C3)    6
 (leaving  0  C4)    6
 (leaving  3  FACT)  6
 (leaving  2  FACT)  6
 (leaving  1  FACT)  6
 (leaving  0  FACT)  6
```

(b) Trace of (FACT 3 (LAMBDA (A) A))

```
(entering  1  FACT)  (3  C1)
(entering  2  FACT)  (2  C1)     (leaving  0  FACT)
(entering  3  FACT)  (1  C3)     (leaving  1  FACT)
(entering  4  FACT)  (0  C4)     (leaving  2  FACT)
(entering  1  C4)    (1)         (leaving  3  FACT)
(entering  1  C3)    (1)         (leaving  0  C4)
(entering  1  C2)    (2)         (leaving  0  C3)
(entering  1  C1)    (6)         (leaving  0  C2)
  (leaving  0  C1)
```

(c) Trace of (FACT 3 (LAMBDA (A) A)) using tail-transfer mechanism

Fig.3 Factorial function with continuation function

(注) 動的結びつけでも、自由変数が無ければ、スタックを用いる関数計算を、続き付きの関数計算に機械的に変換することができる。[28]

も変えることはない。このtail-transferの機構を前提にすると、Fig.3のFACTのトレースは(c)のようになる。さらに、この機構は、静的な結びつけの機構と相まって、内部的には全く繰返し計算となるように再帰呼び出しを用いることが可能になる。その例をFig.4に示す。Fig.3とFig.4からわかるように、この段階では、続き関数はあまり本質的な役割を果していない。ここで強調しておきたいのは、その性質上、続き関数も常にtailにあるということである。

2.4. Send, Receive (メッセージの送受)　さて、Fig.3(a)をその意味を変えずに少し書き替えてみる。λ式の仮引数を2つに分ける。もともとの関数の仮引数に対応する部分と続き関数を受け取る部分とにである。次に関数呼び出しをAPPLYを用いて書き直す。このAPPLY関数は通常のものと少々異なる。第3引数は続き関数である。このようにして書き替えたものがFig.5の(a)である。

```
(DEFINE  FACT
    (LAMBDA  (N)
        (LABELS  ((FACT1  (LAMBDA  (M ANS)
                                (IF  (= M 0)  ANS
                                      (FACT1 (- M 1) (* M ANS))))))
                 (FACT1 N 1))))
                                                        (a) Definition
(entering  1  FACT)  (3)
(entering  1  FACT1) (3  1)      (leaving  0  FACT)
(entering  2  FACT1) (2  3)      (leaving  0  FACT1)
(entering  3  FACT1) (1  6)      (leaving  1  FACT1)
(entering  4  FACT1) (0  6)      (leaving  2  FACT1)
  (leaving  3  FACT1)  6                          (b) Trace of (FACT  3)
```

Fig.4 Factorial function in iterative manner (SCHEME)

```
    (DEFINE  FACT
        (LAMBDA  (N)  (C)
            (IF  (= N 0)  (APPLY  C  (1)  NIL)
                (APPLY FACT ((- N 1))
                       (LAMBDA (A) NIL (APPLY C ((* N A)) NIL))))))
(a) Modified definition(Fig.2 (a))
    (LAMBDA  <argument>  <argument-for-continuation>  <body>)

    (RECEIVE  <message>  <sender>  <next-action>)
    (APPLY  <function>  <actual-argument>  <continuation>)

    (SEND  <message>  <receiver>  <continuation>)
(b) Interpretation Lambda and Apply into Send and Receive action
    (DEFINE  FACT
        (RECEIVE  (N)  (C)
            (IF  (= N 0)  (SEND  1  C  NIL)
                (SEND (- N 1) FACT
                      (RECEIVE (A) NIL (SEND (* N A) C NIL))))))
(c) Definition by Send and Receive action
    Fact : C=>n Then If (= n 0) then 1=>C
                                else (- n 1)=>Fact Then =>a Then (* n a)=>C
(d) Definition in Sysp external notation
    (entering  1  FACT)  (3) (C1)  C1=(RECEIVE (A) NIL A)
    (entering  2  FACT)  (2) (C2)  C2=(RECEIVE (A) NIL (SEND (* N A) C NIL))(N=3,C=C1)
    (entering  3  FACT)  (1) (C3)  C3=(RECEIVE (A) NIL (SEND (* N A) C NIL))(N=2,C=C2)
    (entering  4  FACT)  (0) (C4)  C4=(RECEIVE (A) NIL (SEND (* N A) C NIL))(N=1,C=C3)
    (entering  1  C4)    (1) NIL
        (leaving  3  FACT)
    (entering  1  C3)    (1) NIL
        (leaving  0  C4) (leaving  2  FACT)
    (entering  1  C2)    (2) NIL
        (leaving  0  C3) (leaving  1  FACT)
    (entering  1  C1)    (6) NIL
        (leaving  0  C2) (leaving  0  FACT)
     (leaving  0  C1)    6
(e) Trace of (SEND  3  FACT  (RECEIVE&REPLACE (A) NIL A))
```

Fig.5 Factorial function by Send and Receive actions

次に、2つの action を導入する。

。(SEND <message><receiver><continuation>(注))

<message> ⇒ <receiver> Then <continuation>

<message>を<receiver>に送り、返答は<continuation>に受け取る旨を知らせる。

。(RECEIVE <message><sender><next-action>)

<sender> ⇒ <message> Then <next-action>

<message>に照合するメッセージを<sender>より受けとると、<next-action>の実行に入る。<sender>についての議論は省略する。ここでは、送信側の<continuation>を受けとる仮引数となると考えておけば十分である。

この2つのactionの意味から、先程の書き替えたLAMBDA, APPLYとの対応がFig.5(b)のようにとれる。この対応づけにより、FACTを書き替えたのが、Fig.5の(c)と(d)である。このような書き替えを手順を追って示したのは、このSend, Receiveというaction, すなわちメッセージの送受という計算機構の導入が、さらに、それに基づいて得られる拡張が、今までに少しづつ行なわれてきたλ計算の自然な拡張の延長線上にあるということを示すためである。

このSend, Receive actionの導入により、すぐに導かれる拡張点をいくつか挙げて説明する。

(a) コルティン計算、並列計算　λ式は、Receiveで、関数の呼び出しはSendで表わされる。すなわち計算はReceiveとSendの2つのactionのみで表わされる。(Fig.5の(c)には、DEFINEとIFが、議論の本質にはかかわり無いので、そのまま残されている。) Sendは、呼び出した関数の値をReceive actionを続き関数とすることにより受け取る。このように計算を表わすと、SendもReceiveも常にtailで実行されることになる。Fig.5の(c)でも明らかであるが、もう少し直観的な例をFig.6に示す。ということは、前述のtail-transferの機構を用いると、Sendは、メッセージと続き関数を受信側のReceiveに送ったところでその役割を終え、以後の計算をその続き関数に託することになる。そして、実行環境をstackではなく、heap上に保存する機能を前提にすると、このSendとReceiveは、コルティン制御用のResumeや、並列処理制御用のActivateとWaitと同様の機能を持つと拡張できることになる。言うまでもなく、この拡張は、もともとのλ計算の機能をいささかも損うものではない。GEDANKEN(5)をはじめ、λ式のclosureを用いて、コルティンや並列計算を実現しようとした努力に比べ、驚くべき程の簡潔さで、それらが導入されたことになる。当然のことながら、続き関数となるReceive actionは、λ式のclosureに対応する。

```
(* (+ a b) (- a b))
```

(a) composition of function(call)

```
(SEND (a b) +
    (RECEIVE (x)
        (SEND (a b) -
            (RECEIVE (y)
                (SEND (x y) *
                    (RECEIVE (x) ))))))
```

(b) Send and Receive

Fig.6 A simple comparison between two notations of applicative computation

---

(注) 通常の実行モードでは、<receiver>と<continuation>は、実引数として closure となる。

以上のことが、LAMBDAとAPPLYをあえてReceiveとSendという言葉に書き替えた第一の理由である。

この拡張点をもう少し厳密に議論しよう。まずは、Actor理論ではどのようになっているか見てみることにする。この理論では、並列動作機能は、次の機構によって達成される。

"Actorは、自分の知り合い(acquaintance)であるActorにメッセージを送ることができる。送る相手は複数でもよい。この複数のActorのうち、メッセージを受けとることの出来たものすべてが起動される。"

このようにして、並列動作機能が組み込まれる。しかし，Actor理論では、並列"処理"に必要な同期化の機能等は組み込みとはしない。したがって、メッセージと同時に各Actorに渡される続き(Actor)も、このままでは、それぞれ別の計算として実行されるだけである。並列処理に必要な諸機能は、特殊な能力を持ったActorを用意することによって陽に定義される。

本来、Actorによる計算には、副作用は無い。しかし、並列に動作するActor同志が連絡を取り合うには副作用が必要である。その副作用を生じ得るActorがCellである。このCellに複数のActorが同時にメッセージを送るからには、交通整理が必要である。これを司るActorがSerializerである。さらに、資源配分を司るActor等々が用意される。このようにして、Actor理論は、色々な特殊機能を持ったActorを厳密に定義することによって、並列処理に必要な諸機能を形式化し、理論化する枠組を提供した。[1,27] 特に、最近の作業は、ほとんどこの周辺に向けられている。しかし、この結果、Actor理論が、そのもう1つの目的とした知識表現、問題解決過程の表現という課題からは遠く(?)離れることになる。
もちろん、この並列処理機能を定義する枠組を、問題解決過程の記述のための枠組に用いようとする試み[1]が行なわれたが、現時点では成功とは言いがたい。問題解決等の記述の枠組としては、具体的にActorはどのような姿になるかは、ここでは省略する。ここで指摘して居きたいのは、並列処理と問題解決過程の記述との間には、連続的な結びつけは得られていないということである。(注)

一方、Syspは、知識表現、問題解決過程の記述のための厳密で堅固な土台を作ることが、その目的である。そこで、計算の定義も、Actor理論より、もう少しマクロなレベルに定める。同期化の機能は組込みとする。Syspに於いて、このSend Receiveによって、各種の計算がどのように表現されるかをまとめたものが、Table 1(b)である。

Question	
Yes-no	[Name Mary, Age 21]?
Interrogative	[Name Mary, Age ?=(21 or 25), Hobby ?]
Answer	
Yes-no	Yes. (or No.)
Interrogative	(21 Swimming).
Order	[Name Mary, Hobby Cooking]!
Reply	Ok. (or I-can't)

(a) Types of message

action	continuation(next)
parallel	
Send	Receive, Send
Receive	Send, Receive
co-routine	
Send	Receive
Receive	Send
recursive	
Send-question	Receive-answer
Receive-question	Send
Send-answer	none
Receive-answer	Send

(b) Computations

Table.1 Message, action and computation in QA model

(注) 努力は続けられている。Minskyの社会理論や文献(17)はその一例である。

尚、並列計算の場合、Sendによって渡される続き関数は、そのままが続きを表わすわけではない。この続き関数のactionの系列の中に、自分の返答を受け取るReceive(本当の続き)がある事を示している。又、Receive actionには、同期化の機能も組込まれているとする。従って、並列計算の場合、相手の指定の仕方、つまりSendの<receiver>, Receiveの<sender>に若干巧妙な機構が必要である。

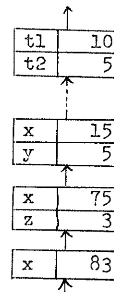
(b) 名前づけの機能　(a)の拡張によってReceive(LAMBDA)は、Send(APPLY)と同格に引上げられた。又、独立して関数を定義するReceive actionと、続き関数となるReceive actionも同格となった。2.3.節での続き関数は、その定義から一変数の関数に限定されていた。しかし、Receiveによって、続き関数は任意個の変数、任意の関数に拡張される。これは、単に多値の返り機能(Multiple-value Return Construct)が実現されただけではない。重要な事は、副作用無しに、計算結果に名前づけする機能を得るということである。一例をFig.7.に示す。(a)は、和と差、積と商を同時に求める関数を利用する計算の系列である。①,③,⑤のSendでは、変数によって最新の値を参照し、②,④,⑥のReceiveでは、得られた値にどういう名前をつけるかが示されている。従って、実行環境の入-変数の結びつけは(b)のように保存される。これによって、代入文の欠点である副作用を排し、利点である名前づけの機能だけを利用できるようになる。この機能は、コルティン計算や並列計算の場合に、変数に結びつけられるのが、交信相手の関数引数(functional argument)でも同様である。Fig.7(c)のMainのReceive actionの<sender>用の仮引数 f_1, f_2には、それぞれFooの@,⑥点の関数引数が結びつけられる。

このReceiveの名前付けの機能は、次の拡張である、パターン照合を介するデータ構造の合成と分解機能によって、その有用性は倍加される。

(c) 実行環境の完全保存機能と消去機能　(b)の名前付けの機能は、この計算機構が、本来、実行環境を完全に保存する能力を持つということを保証する。計算の全歴史が保存される。この機能は、次のような利点を生みだす。

```
......
 (t1  t2  ?) =>Sum&Diff                    .....(1)
    Then =>(x  y).                          .....(2)
       Then (x  y  ?  ?) =>Times&Divide     .....(3)
          Then =>(x  z).                    .....(4)
             Then (x  y  z  ?) =>Sum        .....(5)
                Then =>x.                   .....(6)
                 ......
```

(a) computation



(b) naming for result

```
Main: (t1  t2  ?) =>Foo
         Then f1=>s.
            Then ?=>f1
               Then f2=>d.
                 ...........

Foo: =>(x  y  ?)
          Then (x  y  ?  ?)=>Sum&Diff
             Then =>(x1  y1).
                Then x1.=>
                   Then =>? ----------@
                      Then y1.=>
                         Then =>?------⑥
                           .......
```

(c) naming for functional argument

Fig.7 Some example of denotation mechanism

非決定的プログラム(non-deterministic Program)は、人工知能向言語[22]で、具体的にその有用性を指摘されて以来、だんだんとより一般的に認められるようになった。勿論、その適用範囲など、その有用性の内容もより具体化してきた[4]。このプログラムを決定的化する方法は、深さ優先の後戻り制御と広がり優先のコルティン制御である。いずれにしろ、実行環境の保存機能が背後に必要である。そこで、人工知能向言語では、保存機能の是否を含めて大きな論点となった。しかし、もともと、実行環境の保存などを念頭に置いていない通常の言語上で議論される場合は、実りのないものとなった。

非決定的プログラムは、本来、環境保存機能を持つ計算機構の上に実現されて無理のないものとなる。

とはいえ、ほとんど再び参照されることない実行環境を次々累積していくのでは、あまりに不都合である。そこで、ゴミ集めが登場する。環境の保存を消去にheap(ゴミ集め)を用いるのは、コルティン計算や並列計算を考慮してのことである。多くの場合 プログラム上で、保存すべきか否かを読み取ることができる。

続き関数(Then部)を持たないSendは、送り返されてくる答を自分自身では受取らないこと、自分を呼出したより上位の関数にその受取りを委ねることを適応表示したことになる。ということは、そのSendが実行されるまで作り上げられた実行環境は、もはや、計算全体に何の影響も与えないことが明らかになったのであるから、消去してもよいことになる。この機能により、再帰呼出し計算に対しては、通常の場合と同様に、環境保存には、stackがあれば十分である。この続き関数のないSendは、SCHEMEのtail-transfer機能を、一段と整理し明確にした機構を与えてくれる。Sysp の結びつけ機構も文書上の静的なものである。この機構と相まって、本当に保存すべき実行環境は、極力少なくなるような機構となっている。

<u>2.5. パターン照合と非決定性</u>　　次の拡張は、実、仮引数の結びつけ機構の一般化と強化である。Plannerをはじめ人工知能向言語[22]で用いられその有用性が認められた、種々のデータ構造に対するパターン照合機能を導き入れることである。これにより、データ構造の分解と合成の手続きが、パターンとして、非手続化されて表現される。

データ構造をリストとして説明する。

(分解)　受信の際に受取ったメッセージの分解が行なわれる。Receiveの<message>パターンによって分解の仕方が表わされる。例えば、

⇒ (-- x y) Then ----

は、任意のリストの最後の2要素を取出し、xとyという名前をつける。

(合成)　送信の際に送出すメッセージの合成が行なわれる。Sendの<message>パターンによって合成の仕方が表わされる。例えば、

(`A x -y-) ⇒ ----

x=(1 2), y=(3 4) とすると、(A (1 2) 3 4)

というリストが作られ送出される。

この例でも明らかなように、SendとReceiveでは、その<message>の評価の仕方が異なる。Receiveでは、評価するものを明示するというQUOTEモードで、Sendでは、評価しないものを明示するというEVALモードで評価される。

これは、いわゆる左辺値(Left-hand value)，右辺値(Right hand value)の考え方をより一般化したものである。

Syapでは、基本データ構造として連想(association)，その特殊形として、対と集合(注1)が採用され、それらに対するパターン照合機能が用意される。この機能は、データ型のデータ化、変数間の照合機能により、より強力なものとなる。

パターン照合に、多重照合の可能性を含ませると、その意味付けとして、非決定性が導入される。Syapの基本計算機構には、もう一段拡張された非決定性が組込みとなる。すなわち、Sendの<receiver>と<continuation>, Receiveの<sender>と<next-action>，いずれも複数ものものが与えられてもよい。そして、これは非決定性(注2)を表現するものとして解釈される。

2.6. メッセージの類型化　次なる拡張は、SendとReceiveによってやり取りされるメッセージの類型化である。我々が日常使用する自然言語では、文は、発話者の基本的な意図を反映するように、いくつかに類別される。平叙文、疑問文、命令文、感歎文などにである。話し言葉にも、書き言葉にも、どれに属するかを容易に判定できるように、構造上の手掛りが設けられている。さらに疑問文には、どのような答を期待しているかを明示する機構がある。単なるYes-Noの答を求めているのか、あるいは、疑問詞を用いて、When～, Where～, How many～, What kinds of～ 等々である。文の系列として表わされる会話の流れにも、何をどのように伝えたいか、その意図を反映した構造を持つ。全体的な構造は、言語学上でも研究が始められたところであり、その成果を利用できる段階ではない。しかし、少なくとも会話として成立つ文の系列である以上、文と文との間には、局所的な制約ぐらいは、かなり一般的なものとして、認められるはずである。例えば、発した質問に対する相手の応答は、一番に期待されるのが、それへの答であり、次に期待されるのが、その質問に対する質問であり……等々である。

以上のような、自然言語の持つ、相手に自分の意図を理解し易くするための種々の機能を取入れ、しかも、プログラミング言語としての適切な機構に定着させるのが、ここでの拡張である。

現在設けられているメッセージの型をTable 1(a)に示す。命令(Order)とそれへの応答(Reply)は、基本計算機構には無いが、表現のために有効な副作用を生ずるメッセージの型である。

会話の流れに次の様な枠組をはめる。

(1) 質問をしたら(受取ったら)，必ず答を受取る(返す)。

(2) 会話の流れの上で、最も最近に受取った(送った)質問に対する答を、まず送る(受取る)。

このようにすると、質問と対応する答とは、互いに入れ子構造となる。これを良い構造を持つ(well-structured)と呼ぶ。

(注1) 連想は以前はstructured listと呼ばれていたものである。PLASMAでは、packageと呼ばれているものである。さらにPLASMAでは、対はsequenceと呼ばれ、集合に対しては、より基本的なcollectionが用意されている。collectionは、bagとかmulti-setと呼ばれるものと同じである。

(注2) この非決定性が、計算の最も基本的な表現要素としての条件式の機能を代行する。条件の判定と、それに基づく選択を明示する条件式は、基本要素としては不適切である。

質問を送ったのに、その応答として質問を受取るのは、コルティンとしてのやり取りを表わす。従って、再帰呼出し計算に限れば、質問に対し、返ってくるのは常に答である。更に、この良い構造を持ったQA-系列を単位に、問題解決の手法を用いて、より複雑な複合QA系列を表わす手段が用意される。

3. 他の計算機構(システム)との関連

3.1 Lisp　Syapの基本計算機構は、SCHEMEの手法を利用し、EPICS-Lisp[6]上に製作されている。Syapに於けるLispは、Lispに於けるLAPの役割を果す。Lisp関数(SUBR, EXPR, LSUBR)は、プログラム中に混在してよく、その評価はLispのインタプリタに委ねられる。その時、Syap関数とLisp関数の対応づけは、次の様に行なわれる。

(LABEL <id> (LAMBDA (a_1 ---- a_n) ----

はSyap上では、

(LABELS (<id> (RECEIVE (a_1 ---- a_n ?) ----

に対応づけられる。
そこで、Syap関数でもLisp関数でも次の3種類の呼出し方が利用できる。

① (<id> a_1 ---- a_n)
② (SEND (a_1 ---- a_n ?) <id> RECEIVE x REPLACE)
　　(a_1 ---- a_n ?) ⇐⇒ <id>
③ (SEND (a_1 --- a_n ?) <id> (RECEIVE x ----
　　(a_1 ---- a_n ?) ⇒ <id> Then ⇒ x ----

3.2. Logic Programming[5),8),29)]　述語論理式を特殊なもの(Horn Set)に限定すると、定理証明の機構が、帰納的関数を計算する機構になることを利用したもので、PROLOG[29)]に代表される。証明の仕方を指示する高階述語が、プログラミング言語の制御構造を記述する要素に対応する。
述語、

P (a b c)

は、a, b, cの間にPという関係が成立っていることを表わす。これを質問回答の行為として解釈すると、Pをメッセージの受け手とした場合、

(a b c)? ⇒ P Then ⇒ Yes. から
(? ? ?) ⇒ P Then ⇒ (a b c). までの

8通りが可能である。
Pをメッセージの送り手とした場合も、同様な8通りの解釈が成立つ。述語は、多数の質問応答の行為を凝縮したもの、縮退化したものと見ることができる。この例でみる限りは、述語(関係)表現はまとまりの良い表現法である。
述語が変数を含む場合は、いささか注意を要する。述語論理式は、全ての解釈の仕方に対応する知識を表現しているわけではない。PROLOGでも、入力データによって答が得られたり、得られなかったりする。さらに重要なことは、少々複雑な述語論理式になると、入力を何にし、出力を何にするかによって全体の記述の仕方が異なってくることである。述語論理式のままでは、正確に厳密に計算を表現しているとは言いがたい。そこで、述語論理式に、何が既知で、何が未知であるかを明示する機構が必要になる。一方、質問応答の行為では、質問文に明示される情報が既知であり、期待する回答が未知と、それぞれの情報の性質が明示さ

れる。この既知と未知の区別とそれから派生する質問応答の行為は、言語理解の最も基本となる機構であることが、自然言語研究からも指摘されている。質問応答行為を縮退化したものが述語であるという対応づけにより、Logic Programmingの機能もSyspの計算機構の上に組込まれる。詳細は文献(24,26)を参照されたい。この辺の議論を、より厳密に行なうと内包論理(Intensional Logic)の議論に重なる。

3.3. Functional Programming[2)] 高価な処理時間、高価な記憶領域、これらを無駄のないように使おうという精神のもとに組立てられてきたのが、現在、広く使用されている通常のプログラミング言語である。しかし、情勢は変った。急減に安くなりつつある処理時間、記憶領域、相対的にますます高価になるプログラミング。そこで、従来のプログラミング言語を根本から考え直し、さらに新らしい計算機の方式を考えようという動きが顕著になってきた。FLATSプロジェクト[19]の目指す方向であり、Logic programming等その目指す方向であり、----等々である。その中の1つとして、Functional Programmingというのがある。

述語(論理式)とは裏腹の関係になるが、いくつかの量の間の関係の表現法に関数というのがある

$$z = f(x, y)$$

x, yを独立変数、取る値の範囲を変域、一方、zを従属変数、範囲を値域と呼ぶように、より計算するという行為を意識した表わし方である。当然のことながら、この関数を用いての計算の表現法には、結果に名前付けし参照するという事はあるが、どこに結果をしまっておくか、記憶場所という考え方も、副作用という考え方もない。関数計算の手順は、データの変換の過程、データの流れに沿ったものとなる。本来、計算とは、そう表現すべきものであるというのが、Functional Programmingの出発点である。このFunctional ProgrammingとSyspの対応づけは、別稿(26)で論ずる。Backusが、applicativeな言語の欠点として挙げたhistory sensitiveではないという点は、Syspの基本計算機構の値の受取りを明示する機能により簡明に解決されている。

4. むすび Algol 68, PL/1へと、それなりに収斂した汎用プログラミング言語に対し、ここ数年、再検討し、新らしい言語を考案しようという気運が高まってきた。その気運の主な要素は次の3つである。これらは互いに相補的である。

① 記号処理化：あらゆる分野で、数値処理、ビット列、文字列処理が記号化され、高度な構造を持つ記号処理に置きかわる。

② プログラミングの純化：現在の計算機システムの制約から離れ、純粋にプログラム(イング)とはいかなるものかという見直しがなされる。

③ 諸機能の統合化：言語、支援機能、OSの機能、データベースの機能がuniformなシステムに統合される。各機能は、他の機能を有効に利用することにより、その能力を倍加する。

このような気運は、人工知能(特に、知識表現)研究、数式処理、ソフトウェア科学、ハードウェア技術等、少しづつニュアンスの差はあるが、情報処理研究の全分野に生じている。今後、沢山の新らしい応用分野が具体化され、沢山の新らしいプログラミング言語が開発され、それに基づいて沢山の新らしい計算機システムが考案されていくであろう。そのような現状認識のもとに、Syspの研究は進められている。

謝辞： Lisp上への製作は、元吉文男君の助力に負う。常に議論の手掛りを与えてくださった渕室長、研究の機会を与えてくださった西野部長に感謝する。

References

1) Atkinson, R. and Hewitt, C. : "Synchronization in Actor Systems," 4th SIGPLAN-SIGACT Symposium on Principles of Programming Langauges, Los Angels (Jan. 1977)
2) Backus, J. : "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," CACM, Vol.21 No.8, pp.613-641 (Aug. 1978)
3) Baker, H. G. Jr. : "Shallow Binding in Lisp 1.5," CACM, Vol.21 No.7, pp. 565-569 (Jul. 1978)
4) Dijkstra, E. W. : "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," CACM, Vol.18, No.8, pp.453-457 (Aug. 1975)
5) van Emden, M. H. : "Programming with Resolution Logic," Machine Intelligence 8 (ed. E. W. Elcock) Edinburgh University Press, pp.266-299 (1977)
6) ETL : "LISP User's Manual," EPICS-5-ON-4 (Mar. 1978) (in Japanese)
7) Foster, J. M. and Elcock E. W. : "Absys 1 : an Incremental Compiler for Assertions; an Introduction," Machine Intelligence 4, pp.423-429 (ed. Michie, D.), Edinburgh Univ. Press (1969)
8) Fuchi, K. : "Logic Programming - EPILOG," IPSJ WGSYM 1-2 (Jul. 1977) (in Japanese)
9) Goldberg, A. and Kay, A (eds.) : "Smalltalk-72 Instruction Manual," Xerox Palo Alto Research Center SSL 76-6 (Mar. 1976)
10) Goto, E et al. : Report of the FLATS Project, Vol.1, Information Science Lab., IPCR (Oct. 1978)
11) Greif, I. and Hewitt, C. : "Actor Semantics of PLANNER-73," Proc. ACM SIGPLAN-SIGACT Conf., Palo Alto (Jan. 1975)
12) Hewitt, C. : "Viewing Control Structures as Patterns of Passing Messages," Artificial Intelligence 8, pp.323-364 (1977)
13) Ingalls, D. H. H. : "The Smalltalk-76 Programming System Design and Implementation," Conf. Record of 5th ACM Symposium on Principles of Programming Languages, pp.9-16 (1978)
14) Landin, P. J. : "A Correspondence Between ALGOL 60 and Church's Lambda-Notation," CACM 8, pp.89-101 and 158-165 (Feb.-Mar. 1965)
15) Reynolds, J. C. : "GEDANKEN - A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept," CACM, Vol.13 No.5, pp. 308-319 (May 1970)
16) Reynolds, J. C. : "Definitional Interpreters for Higher-Order Programming Languages," 25th ACM, pp.717-740 (1972)
17) Smith, R. G. and Davis, R. : "Distributed Problem Solving: The Contract Net Approach," STAN-CS-78-667, Stanford Univ. (Sept. 1978)
18) Steele, G. J. Jr. and Sussman, G. J. : "LAMBDA : The Ultimate Imperative," AI Memo 353, MIT (Mar. 1976)
19) Steele, G. J. Jr. and Sussman, G. J. : "LAMBDA : The Ultimate Declarative," AI Memo 379, MIT (Nov. 1976)
20) Steele, G. J. Jr. and Sussman, G. J. : "The Revised Report on SCHEME A Dialect of Lisp," AI Memo 452, MIT (Jan. 1978)
21) Sussman, G. J. and G. L. Steele Jr. : "SCHEME An Interpreter for Extended Lambda Calculus," AI Memo 349, MIT (Dec. 1975)
22) Yokoi, T. : "Programming Languages for Artificial Intelligence (1)&(2)," Journal of Information Processing of Japan, Vol.17 No.7&8 (1976) (in Japanese)
23) Yokoi, T. : "SYSP(Structured Symbol System Processor) I. Basic Mechanism," Bul. Electrotech. Lab., Vol.42 No.4, pp.1-22 (Apr. 1978)
24) Yokoi, T. : "SYSP II. Various Structures for Representation," Bul. Electrotech. Lab., Vol.42 No.9, pp.1-15 (Sept. 1978) (in Japanese)
25) Yokoi, T. : "SYSP III. Combining Various Structures," Bul. Electrotech. Lab., Vol.42 No.6, pp.1-15 (Jun. 1978) (in Japanese)
26) Yokoi, T., Yokoyama, S., Sato, T., Motoyoshi, F. and Fuchi, K. : " Sysp : a new programming language for the next generation," draft for IJCAI-79 (1979)
27) Yonezawa, A. : "Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics," MIT LCS TR-191 (Dec. 1977)
28) Wand, M. and Friedman, D. P. : "Compiling Lambda-expressions Using Continuations and Factorizations," Computer Languages, Vol.3, pp.241-263 (1978)
29) Warren, D. H. D. and Pereira, L. M. : "PROLOG - The Language and its Implementation compared with LISP," SIGPLAN Notice, Vol.12 No.8 / SIGART Newsletter No.64, pp.109-115 (Aug. 1977)