北海道大学工学部研究報告
第130号（昭和61年）

Bulletin of the Faculty of Engineering,
Hokkaido University. No. 130 (1986)

L a p l a s
—マイクロ・コンピュータに適した新らしい言語—

原田　康徳*　北村　正直**

（昭和60年11月20日受理）

L a p l a s
—A new language suitable for micro-computers—

Yasunori HARADA and Masanao KITAMURA

(Received November 20, 1985)

Abstract

A new computer language, Laplas, is introduced. The name Laplas stands for Language Processor for Listing and Stacking. As its name suggests, Laplas makes use of list- and stack-processings together.

Laplas is a function-level programming language, in which new functions are defined in terms of the system functions and the functions defined previously. Local variables and recursive definition can also be used in defining a new function. By incorporating stack-processing with these, in defining a function in Laplas less local variables and recursive procedures are used than in LISP. Thus a Laplas program needs less memories and can be processed very quickly. Because of these features Laplas is considered as a language suitable for micro-computers.

Laplas is equipped with a three-dimensional and multidragonfly (not turtle) graphics. These fine features of Laplas make its graphics so powerful that even animations can be programmed quite easily. This three-dimensional graphics can be handled interactively even by a child with ease. Since stacking is used, the word order of Laplas is inverse-polish. A Japanese version of Laplas suitable for Japanese children can be created simply by renaming all the functions with suitable Japanese words.

1．序　　論

現在数多くのコンピュータ言語が計算機を用いる仕事に使用されている。それぞれの言語には得意とする分野があり，そこにおいてその言語は有効に使用されている。通常広く使用されている言語のFORTRAN, Cobol, PascalやBasicなどは逐次処理型の言語であるが，この論文において紹介する言語Laplasは，関数型プログラミング言語といわれるLISP, ForthやLogoなどと同じグループに属し，FORTRAN等の言語とは性格を異にする言語である。

今日マイクロ・コンピュータの機能は急速に進歩しつつあり，その上で使用できる言語の種類

＊　応用物理学科
＊＊　共通数物系

も増え，言語の機能も飛躍的に向上している。しかし，リスト処理を取入れた言語には未だ本当に小型機向きに改定されているものはほとんどない。Laplasはかなりのリスト処理能力を持ち，小型機にて使用できる言語を目指して開発されたものである。特にプログラムの構造をシンプルにすることに努力がはらわれた。プログラムの書式において括弧の数が少なくて済むのはこの努力の結果の一つである。この結果，この言語においてはプログラムが短く簡単に作れるようになった。さらにLaplasが初心者でも容易に習得できるのも，その言語の構造が単純化されたことに因る。

上にあげた言語の中でForthはスタック処理を用いた独特な性格の言語である。この言語においては，関数を定義し，それをスタック上に積まれたデータに作用させることにより処理作業は進められる。またスタック上に積まれたデータを細工するシステム関数も多く準備されていて，かなり複雑なプログラムをも組むことができる。一方LISPはリスト処理言語であり，LISPにおいては関数を定義し，それらを呼びだし，必要なデータを与えながら処理作作業を進める。LISPの関数の定義においては局所変数が複数用いられることがしばしばある。プログラムの実行中にこれらの変数を探して呼びだしたり，また新しい値を入れて収めたりすることが頻繁に行なわれる時には処理速度が遅くなることがある。またLISPの関数の定義において再帰定義の手法がしばしば用いられる。これは非常に便利な方法であるが，場合によってはプログラムの実行効率を悪くし，かなりのメモリー領域を必要とする。これがLISPをメモリー領域が比較的小さいマイクロ・コンピュータに移植する際の重要な障害となっている。

Laplasにおいてはリスト処理とスタッフ処理とを併用し，それぞれの特徴を組みあわせて使用している。例えば関数の定義において局所関数の幾つかを省略し，代わりにスタックを利用することにより処理速度を上げることができる。また，リスト処理をも併用してかなり複雑なプログラムをも組むことが可能である。またLaplasのプログラムはLISPのプログラムより括弧の数が少なく，その書式は簡潔で見やすい。このことがLaplasを初心者に親しみ易くしているのである。またこのことによりLaplasにおいてはプログラムの開発，デバッキングが容易になっている。さらに関数の定義においてもスタック処理を用いて再帰定義の幾つかを取除き，使用するメモリー領域を節約し，実行のステップ数を減らすことによりプログラムの実行効率を高めることができる。

第二章においてLaplasのスタック処理とリスト処理についてForthとLISPと比較しながらこの言語の概要を説明する。第三章においてはこれらの処理を組みあわせたLaplasの特徴を例を引きながら論ずる。第四章ではこの言語の得意とする三次元グラフィックの機能と，その応用例を紹介する。最終章はこの言語の将来の検討とまとめである。

2．Laplasの概要

2－1　スタック操作

Laplasのスタック操作はForthのそれとほぼ同じである。すなわちキーボードより数値を入力するとそれがスタック上に順次積まれていく。そのスタックに積まれたデータに対する“dup”，“swap”，“drop”等の基本的操作はシステム・ワードとしてLaplasにも備えられている。より複雑なスタック操作は最初のヴァージョンには組みこまれていない。勿論これらのスタック操作ワードは基本ワードの組合せで簡単に定義できる。しかし複雑なプログラムを実行するときには効率を良くする為，これらのワードはシステム・ワードとして組みこむ必要があるので，これらがシステム・ワードとして組込まれたヴァージョンも作成されている。Laplasのスタック操作がForthと異なる点はスタック上に数値データばかりでなく，リストをも積むことができることで

ある。

Forthは非常に興味深い言語ではあるが一部のファン以外にはあまり知られていないので，スタック操作も一般には知られていない。ここでLapalasのスタック操作を説明しよう。LaplasにおいてForthにおけると同様には関数をワードと呼び，スタック操作のワードも同じ名前を用いていることが多い。

スタックに数値，例えば3，を積むには3を入力し，リターンキーを押す。

```
：3          ：3  ?          ：3  d r o p
               3
```

“：”はプロンプトである。左側は何も表示されない。このときスタックの一番上には3が積まれている。中央の場合は一つの空白を置いて“？”が入力される。この行は「3を入力し，スタックの一番上のデータを（取出して）スクリーン上に表示しなさい」という意味である。3が表示された後ではスタックの状態は3を入力する前と同じになっている。右側の例は「3を入力し，スタックの一番上のデータを取り除きなさい」と読む。したがってこれは3をスタックに積みそれをただちに落すだけである。スタックの状態は3を入力する前と同じになっている。

スタック操作のワードとしてこの他に“dup”，“swap”，“rot”等がある。“dup”はスタックの一番上のデータを複製してそれを元のデータの上に乗せるワードである。“swap”はスタックの一番上のデータをその直ぐ下のデータと入れ替えるワードであり，“rot”はスタックの一番上から三番目のデータを取り出してそれを一番上に置くワードである。これを図解すると下のようになる。入力行の下の数列はそれぞれのワードが作用する直前のスタックの上の部分の状態，その下の行はワードが作用した後のスタックの状態である。

```
：1  2  d u p      ：1  2  s w a p      ：1  2  3  r o t
  ( 1  2 )           ( 1  2 )             ( 1  2  3 )
  ( 1  2  2 )        ( 2  1 )             ( 2  3  1 )
```

これらの数列で右側がスタックの上部にあたる。

Laplasにおいては常にスタックの一番上に何があるかを念頭に置いてプログラムすることが必要である。

Laplasではスタック上に数値の他にリストをも積むことが出来ると先に述べた。このことをある整数の階乗を求めるプログラムを例に取って解説しよう。そのプログラムは

```
(1)   :  1  ( c o u n t e r  * )  7  r e p e a t  ?
      5 0 4 0
```

となる。このときまず　1　がスタックに積まれ，その上に（ｃｏｕｎｔｅｒ　＊）というリスト，さらに上に　7　が積まれ，repeatはこのリストと直ぐ前の数，この場合は　7　を引数とする制御ワードで，このリストをこの数の回数だけ実行させる。“counter”というワードは“repeat”に付属する関数で一回目の時は　1　という値を取りリストを一回実行するごとに，順次1づつ増えていく。最後の“？”はスタックの一番上の値を取ってそれをスクリーン上に表示するワードである。このワードが実行されると表示されたデータはスタックから取り去られる。このように数値データのみならずリストをもスタック上に積むことができる点がLaplasのスタック処理の特徴である。

2－2 リスト処理

リスト処理の基本関数(ワード)としてはcons, head, tailがある。これらはそれぞれLISPのcons, car, cdrと同じである。勿論スタック操作を使用するので引数と関数の順序は逆ポーランドになっている。例えば

```
：1 nil cons ?
( 1 )
：( a b c ) head ?
a
：( a b c d ) tail ?
( b c d )
：( 1 2 3 4 ) ( tail ) 2 repeat head ?
3
```

というように語順が異なることと，スタックを使用する以外はLISPやLogoの対応する関数と全く同じである。リストを操作するその他の関数(ワード)はこれらのワードとその他の制御ワードを組みあわせて作ることができるのはスタック操作のワードの場合と同じである。しばしば使用するワードはシステム・ワードとして始めから組みこんで置くこともできるが，この言語システムをコンパクトにするためにシステム・ワードは現在は必要最小限のものにとどめている。これはスケールの違いはあるがLISPにおいても同じであろう。多くのLISPではREVERSE, APPEND等はシステム関数として組みこまれているが，これらをより基本的な関数で組みたてることができる。Laplasでも後でワードの定義の項で触れる様に同じことができる。

2－3 演算ワード

四則演算ワード＋, －, ＊, ／は逆ポーランド記法で入力する。しかし，演算記号（引数）という型式で書いてもよい。例えば1＋2という式は

```
：1 2 + ,    ：1 +( 2 ),    ：+( 1 2 )
```

という三通りの書き方がある。このとき演算記号と括弧の間に空白をいれてはならないことを注意して書く必要がある。これらの記法を用いて1＋2×5という式は次の様に書きあらわされる。

(2)
```
：1 2 5 * +
：1 +( 2 *( 5 ) )
：+( 1 *( 2 5 ) )
```

しかし，Laplasにおいてはこのどれもが入力された時に，すべて同じ文字列となる様に処理されて，その後で実行される。例えば，次の様にリストとした入力しその，リストを覗いてみると

```
：( +( 1 *( 2 5 ) ) ) ?
( 1 2 5 * + )
```

と逆ポーランド記法で表示されている。

Laplasはこのほか次の実数関数も備えている。

sin, cos, atn, sqrt, log, exp, int

2－4 比較演算ワード

Laplasは比較演算関数（ワード）として＝とminuspとを持っている。前者は二つの引数をとり，後者は一つの数値の引数をとる。＝の引数はどんなものであってもよく，それらは＝が実行さ

れるときにスタックの一番上に積まれている二つのデータである。それらが同じときtrueを，そうでないときnilをスタック上に積む。minuspはその引数（数値）が負であるか否かを判定するワードである。

```
:4 minusp ?
nil
:-5 minusp ?
true
```

このminuspを用いてその他の数値比較演算ワード<，>，≦，≧を次ぎのように定義することができる。

(3)
```
:'< (- minusp) .
:'> (swap <) .
:'≦ (> not) .
:'≧ (< not) .
```

これらのワードは皆二つの引数をとる。

このようにLaplasの現在のバージョンは基本的な関数のみをシステム・ワードとして備えていて，これらを使用して他のワードを組みたてることができる様になっている。しかし，実行効率を高める為には，この様な複合ワードの幾つかはシステム・ワードとして組みこむ必要があるだろう。どのようなワードをシステムに組み込むかは将来の課題である。

2－5 論理演算ワード

前節においてのべた比較演算ワードも論理演算ワードであるが，そのほかに基本的な演算ワードとしてLaplasはif, whileを持っている。これらは他の言語の相当する関数と同じ機能を持つが，逆ポーランド記法によるため引数の引渡し方が異なる，すなわちそれぞれ次ぎの書式で入力される。

(4) 論理値 （リスト1） （リスト2） if

(5) （リスト1） （リスト2） while

ifは論理値と実行可能な二つのリストを引数としている。最初論理演算ワードが来て，それが実行され論理値，真（true）か偽（nil）のどちらかの値がスタックに積まれている。真の時はリスト1を，偽の時はリスト2を実行させる。

whileは同様に実行可能なリストを二つ引数とする。論理型である最初のリスト1が真である間，次ぎのリスト2を実行し，リスト1が偽の時は終了する。

これらのワードは演算ワードと同様に拡張された記法で書きいれることができる，例えば

```
:'a (if(0 =
::        ("Zero" ?)
::        ("Nonzero" ?))) .
```

と定義されたワード a を3と0に作用させるとそれぞれ次ぎのようになる。

```
:3 a
"Nonzero"
:0 a
"Zero"
```

whileにも同様な記法がゆるされる。しかしLaplasは四則演算ワードの時と同様にこれを基本

的な逆ポーランド記法に書きなおしてから実行する。

not, and, or 等の論理ワードもスタック状態を考慮して定義することができる。not は一つの, and と or は二つの論理値を引数とする。例えば次の様に定義すればよい。

(6)
```
:'not (nil =) .
:'and (() (drop nil) if) .
:'or ((drop true) () if) .
```

その他の論理演算ワード，例えばあるデータがあるリストの要素であるかを判定する member 等も簡潔に定義できる。

2－6 ワードの定義

先に Laplas は関数型言語であると述べた。Laplas においては定数，変数，コマンド，関数等がすべて区別されずにワードとして同等に扱われる。また後からユーザーにより定義されて付加えられたワードも使用上はシステムワードと同等に扱われる。ワードの定義はすでに用いてきたがここで改めてのべよう。定義の例として階乗の定義を見てみよう。

```
:'! (1 swap (counter *)
                swap repeat) .
```

である。すなわち定義は

```
:'ワード名 (定義の内容) .
```

という形式でなされる。ここで空白に続く最後の"."が定義を締め括る機能を持つワードである。

ワードの定義の形式において，Laplas が LISP や Logo と異なる所は Forth と同様に受け渡す引数を変数を用いずに定義することができる点である。例えばある数の三乗を求めるワードの定義は

```
:'cube (dup dup * *) .
```

これを Logo では

```
TO CUBE :X
  RESULT :X * :X * :X
END
```

の様に定義する。この CUBE を使用して 3 の 3 乗を求めるには Logo では

```
PRINT CUBE 3
27
```

とする。一方 Laplas においては

```
:3 cube ?
27
```

と書く。Logo や LISP においては定義文における引数の入力の繁雑さからこのような単純なワードは実際には定義して用いることは希であるが，Laplas や Forth においては定義が短いのでこの様なワードでしかも数回しか使用しなくともこのような関数を定義して使用する方がメモリーの使用が少なく，プログラムが簡潔で見易く，またその保守にも便利である。

Laplas は LISP の様に再帰処理を定義の中に紙みこむことができる。

例えば整数の階乗を LISP の様に定義すると

```
(7)   :'!  ((n)  var
             'n  let
             n  0  =
             (1)
             (n  1  -  !  n  *)  if) .
```

となる。この定義においてvarはローカル変数を作るワードである。'n letは！の引数となる整数をnの値とすると言う意味である。

ローカル変数を使わずにスタック処理をもっと徹底して用いると，階乗は

```
(7')   :'!  (dup  0  =
            (drop  1)
            (dup  1  -  !  *)  if) .
```

と定義することもできる。これらの二つの定義を比較するとnの値が大きくなると，共にリターン・スタックのレベルが当然増加するが，データ・スタックのレベルは(7)の定義では増えないが(7')の定義では増える。しかしそれは単なるスタック操作で処理されるので実行速度にあまり影響はない。一方，(7)の定義ではnがローカル変数であるため，呼びだしの度に前のnの値をどこかに収納しなければならない，従ってその為のメモリー領域を消費し，かつ実行速度が遅くなる。この様な簡単な例でも最初に例としてあげたスタック操作と制御ワードrepeatを使用する最もForth的な階乗の定義から(1)のLISP的な定義まで幾つかの可能性がある。Laplasにおいてはプログラムの開発のし易さ，実行速度等をも含めて，最も効率的な定義を選ぶことができる。

効率的な定義について考察するために例としてリストを逆に並べかえるワードreverse，二つのリストを合わせて一つのリストにするワードappendの二つを取上げよう。これらもシステム・ワードの組合せで作ることができるが，Laplasの処理の特徴を示すために，これらの関数を再帰処理とローカル変数を使用してLISP的に書いた定義，またスタック処理を併用してこれらの処理を部分的に置換えた定義を検討しよう。

3．Laplasの特徴

3－1　Laplasの再帰処理とローカル変数

Laplasはスタック処理とリスト処理の特徴を巧みに組合せた言語であると先に述べた。このことをリストを逆に並べ換える関数(ワード)reverseと，二つのリストを合わせて一つのリストにするワードappendの二つを例として詳しく論じよう。

これらの関数をLISPの解説書に載っている例を参考にしてLaplasで定義すると次ぎのように書ける。

```
(8)   :'reverse  ((lis1)  var
      ::     'lis1  let  nil
      ::   (lis1  nil  =  not)
      ::   (lis1  tail  reverse
      ::     lis1  head  swap  cons
      ::     append)  while) .
```

```
(9)   :'append ((lis1 lis2) var
      ::     'lis2 let
      ::     'lis1 let
      ::     lis1 nil=
      ::   (lis2)
      ::   (lis1 head
      ::     lis1 tail lis2 append cons)
      ::     if) .
```

これらの定義において再帰定義の入れ子の中のローカル変数は元のローカル変数とは異なるメモリー番地に登録される。これらのローカル変数の呼びだし，値の収納等により効率は悪くなるのは当然である。またリストの長さが大きくなるにつれて，ステップ数と使用されるメモリー領域は飛躍的に増加する。実際1から200までの数を要素とするリストを我々のシステム(計算機：SORD-M68, Laplas-version 1.5 セル数：10,000)で試したところ使用メモリー・オーバーでシステムが落ちてしまった。これはLISPは大量のメモリーを必要とし，本質的に大型計算機向きの言語であることを示している。

ここで'lisl letというのは“スタックの一番上の値をlislとする”という意味である。これは引数の引渡しの操作でありLISPにおいては変数の宣言の中に含まれている。Laplasの新しい版では同様に変数の宣言中に引数の引渡しが含まれている。

次ぎに再帰処理を使用しないこれらの関数の定義を検討してみよう。

```
(10)  :'reverse ((lis1 lis2) var
      :: nil 'lis2 let 'lis1 let
      ::(lis1 nil = not)
      ::(lis1 head lis2 cons 'lis2 let
      :: lis1 tail 'lis1 let)
      ::while lis2) .
(11)  :'append ((lis1 lis2) var
      :: 'lis2 let
      :: 'reverse 'lis1 let
      ::(lis1 nil = not)
      ::(lis1 head lis2 cons 'lis2 let
      :: lis1 tail 'lis1 let)
      ::while lis2) .
```

ここでは再帰処理は全く含まれていない。しかしreverseにおいてローカル変数の数が増えている。従ってこれらもまだ効率の良い定義とは言えない。

さらに再帰処理を使わず，さらにローカル変数を減らすためにはスタック処理を利用して次の様に定義する。

```
(12)  :'reverse ((lis1) var
      :: 'lis1 let nil
      ::(lis1 nil = not)
      ::(lis1 head swap cons
      :: lis1 tail 'lis1 let)
```

```
    ::while) .
(13) :'append ((lis2 n) var
     :: 'lis2 let
     :: dup length 'n let
     ::(dup head swap tail) n 1 - repeat
     :: head lis2 (cons) n repeat) .
```

これらの定義は再帰処理によらず，しかもローカル変数の数も少ない。逆リストを作る関数においてはリストの要素の数に等しい回数だけローカル変数を呼び出したり，それに値を代入し，スタック上の三個だけ使用する。二つのリストを結合する関数においては，二つのローカル変数は単なる定数で一，二回使用されるだけである。しかし前のリストの要素の数だけスタックは使われる。このように再帰処理を使わずに，またローカル変数の数をできるだけ少なくすることにより，使用するメモリー領域を小さくできる。またこれは処理のステップ数を少なくすることにも繋がり，処理速度を高めているのである。(12)のreverseで100個の要素を持つリストを逆配列にするには約0.5秒かかる。(10)の定義によるワードを用いると同じ処理でも約0.7秒必要である。これよりローカル変数の数は予想通り処理速度に影響することが示された。我々の使用した計算機のＣＰＵは68000であるが，8086のＣＰＵを持つ計算機では同じ処理をするのに心持ち時間がかかる。

ついでにローカル変数を全く用いずに再帰処理を使ったこれらの関数の定義を示そう。

```
(14) :'reverse (dup nil =
     :: ()
     :: (dup tail reverse swap head
     ::  nil cons append)
     :: if) .
(15) :'append (swap dup nil =
     :: (drop)
     :: (dup head swap tail
     ::  rot append cons)
     :: if) .
```

このappendは100個の要素を持つ二つのリストに作用させても処理速度は(8)ほど遅くないが，reverseはかなり遅く，100個の要素のリストに作用させてもメモリー・オーバーを引起こす。これはappendという再帰処理をするワードを使った二重の再帰処理をしているからである。このappendの定義の4行目のrotはスタックの上から3番目のものを取り出して，スタックの1番上に積むスタック操作関数である。

これは一例であるが，ほとんどすべての再帰処理やローカル変数，またはその両方を用いる関数について同じ議論が適用できる。従ってワードの定義においては，

1；できるだけスタック操作を利用する

2；それが複雑になるときは再帰処理やローカル変数を用いる

3；ローカル変数の数は最小限にとどめる

4；一つのワードのなかに二重，三重の再帰処理は避ける，やむを得ない時はメモリー・オーバーフローが起きないことを確かめる等，慎重にワードを定義するべきであるという点を心得えておく必要がある。

LaplasはForthとLISPの両方の特徴を生かした言語であり，関数の定義もこの章で示したようにLISP的なものから，Forth的なものまで何種類もの定義を組むことができる。我々の経験によればこれらの中で最も効率の良いのは中間の定義であるが，これはシステム・ワードの選び方とその組み方にある程度依存するであろう。

3－2 その他の特徴

Laplasはインタープリタ型の言語である。この言語は幾つかのワードを覚えれば，それらを使ってプログラムを作り，実行させることができる。またLaplasはほとんどすべての記号，文字をワードの名前に使用することができる。従って，日本語ですべてのワードを，例えば

```
：′くりかえす （ｒｅｐｅａｔ） .
```

と“repeat”を“くりかえす”と言い直して（renaming），Laplasの日本語版を簡単に，しかも短時間に組むことができる。しかもこうして作られた日本語Laplasの実行効率はほとんど変らない。またLaplasの語順が逆ポーランドなので，すなわちその語順は日本語の語順と非常に近いので，欧米の子供たちがLogoに直ぐ慣れるように，日本の子供たちはこの言語に直ぐに慣れるであろう。

インタープリタ型の言語の特徴を充分に生かすには良いスクリーン・エディタが必要である。LaplasはＡＰＬや幾つかのBasicに備わっているエディティング機能ににた機能を持っている。スクリーン上にあるプログラムやデータ，リスト等を編集しようとするときには訂正する箇所にカーソルを移動し，そこで編集，修正ができる。またその場で実行することも可能である。この機能はグラフィック・プログラムを開発するときに特に有効である。またLaplasのプログラムが非常に短くて済むので，スクリーンに出ているプログラム文を修正することで充分である。もしもスクリーン上にない関数，例えばreverseというワードを修正したいときは，

```
：′ｒｅｖｅｒｓｅ ｔｙｐｅ
```

と入力すれば関数領域に現在収められているこのワードの内容がスクリーン上に表示される。そこにカーソルを移して修正し，カーソルをこのワードの定義文の最初の行に移動してからカーソルが定義文の最後の行を過ぎるまでリターン・キイを繰返し押してワードの修正を完了させる。このようなスクリーン・エディタによりLaplasのプログラム開発は非常に容易になった。

4．三次元グラフイックス

著者の一人（原田）が開発した三次元トンボ・グラフイックスがLaplasの中に組みこまれている。Laplas言語の特徴はトンボ・グラフイックスの機能を非常に使い易いものとしたばかりでなく，この三次元グラフイックを強力なものにした。

4－1 基本操作ワード

Laplasのグラフイック用のワードはあまり多くない。それはマイコンのグラフイック機能が機種によって大きく異なり，あまり沢山のワードを用意するとこの言語を移植する際に大きく作り直さなければならないからである。従って，ほんの数個のワードのみを用意し，それだけを機種によって手直しすれば他の機種に移植できるようにした。しかし，この言語の能力と組みあわせれば，ほとんどマイコンのグラフイック言語に匹敵するか，もしくはそれ以上の機能をLaplasに持たせることができる。

グラフイック用の基本的なワードは

home　トンボを画面の中央（座標の原点）に，右向きに，背中を上に向けて位置させる
cls　グラフイック画面上に描かれているものをすべて消去する
turn　トンボの向きを現在の向きから引数の角度だけ時計と逆回りに回転させる
up　トンボの向きを現在の向きから引数の角度だけ上に向かせる
spin　トンボの姿勢を現在の向きに変えずに引数の角度だけ右に捩じる
move　トンボを現在向いている方向に引数の距離だけ進ませる
draw　現在向いている方向に引数の距離だけトンボを進ませながら線を引く
color　引数の番号（1～15）の色を選ぶ
tomdup　トンボ・スタック上の一番上のトンボ（位置，向き，姿勢）を複製する
tomdrop　トンボ・スタック上の一番上のトンボを取り去る
tomswap　トンボ・スタックにswap操作をほどこす
sphere　トンボの現在位置を中心として引数の大きさを半径とする球を描く

これらの機能を使って正6面体，正12面体，正20面体の展開図や立体図および面間の角度を任意に選んだ展開図等を描くプログラム（ワードの集合）や，水素，酸素，炭素および窒素からなる化合物の立体構造図，化学構造式，分子式等を示すことができ，また再帰処理を用いて192個の炭素原子を含むダイヤモンドの結晶を描くこともできるプログラム，点対称群の操作をワードとして種々の結晶構造を立体的に描く結晶グラフイック・プログラム等がパッケージとして完成している。複数のグラフイック画面を持ち，しかも処理速度の早い16ビットのCPUのマイコンでは，これらのプログラムをそのパレット機能を使用するワードを用いたアニメーション・プログラムと組合せて，得られた図形を回転させたり，振らせたりすることができる。またロボットを宇宙遊歩させるプログラムもLaplasにて組まれている。

これまで関数（ワード）を定義するとか，プログラムを組むとかいう表現を使ってきたが，関数型の言語においては実はこれらは同じことなのである。一つの大きなプログラムも名前をつけてワードとして扱うことができるのである。しかし，多くの場合そのワードの定義中にはまた幾つもの他のワードが使われているので，それらのワードはこのプログラムが実行させるときにはLaplasのワードの定義域に登録されていなければならない。それにはこのプログラム自身がこれらのワードの定義文を持つか，または予めこれらのワードがプログラム・ワードと一緒に纏めてLaplasの中に呼込まれていることが必要である。先に触れた幾つかのプログラムはこの様な意味でワードの集合である。

これらのプログラムのワードの一部を同じ名前でしかも異なる定義を持つもので置換えるとそれらのワードを用いたプログラムは異なる仕事をする。これを我々は“プログラムの実行環境を設定する”と呼んでいる。例えば先の化学構造の立体図のプログラムを用いてメチルアルコールを描くワードを“methyl”という名前で次ぎのように定義する。

```
:'methyl (cst h c h h o h) .
```

そうしてmethylと入力するとこの分子立体構造モデルが三次元的に表示される。ここでh，c，oはそれぞれ水素，炭素，酸素原子を描かせるワードである，またcstは化学構造の図を書くため，複数のトンボを準備するワードである。そこで

```
:chemi2
```

と入力する。このワードはこれらの原子の別な定義，すなわち化学構造式を描く為の定義の集まりである。ここでmethylと入力するとこの分子の構造式が画面に表示される。この化学構造図を描くプログラムが呼びこまれると実はchemi1というワードが実行されて化学構造図が描かれる環境が設定される。chemi3というワードは分子式を表示する為の環境を設定する。chemi2の環境にあるときchemi1と入力すれば初期の環境が回復される。

この“プログラムの実行環境の設定”という概念はプログラミングにおいて非常に有用なものである。例えばＣＡＩの実行プログラムにおいて，学習の進行を学習者の反応によってプログラムによって分岐させる代りに，あるワードの内容を取り替えることにより同じ分岐を実現させることができる。

5．まとめと将来の展望

Laplasはリスト処理とスタック処理とを合わせて使用する小型計算機にも適した関数型言語である。この二つの処理の組合せにより，使用するメモリー領域が少なくて済み，また処理速度が高められた。さらにこれはこの言語に強力な能力を与えている。

この言語の今後の課題は，実行効率を高めるため，いかに組み込み（システム）ワードを多くし，しかも現状の使い易さを保持するかである。

現在グラフイックとＣＡＩへの応用プログラムはあるが，人工知能的な問題への応用を含めて他の分野における利用を考えることが必要である。

Laplasの言語としての評価もさらに徹底してなされることがこの言語の将来にとって必要であろう。

参考文献

Lispについて

1．McCarthy, J., et al., LISP 1.5 Programmer's manual, MIT Press, 1962.

2．黒川利明，LISP入門，培風館，1982.

Forthについて

3．Katzan, Harry Jr., Invitation to FORTH, Petrocelli Books, Inc., 1981.

トンボ・グラフイックスについて

4．原田康徳，鏡 慎，三次元版タートルグラフィクス，Information, vol. 3 No. 6 & 7, 1984.