ＶＡＬＩＤ言語システム：　遅延評価機構とその実現

長谷川　隆三　　雨宮　真人
（日本電信電話公社 武蔵野電気通信研究所）

1．まえがき

関数型言語〔1,2,3〕は記述の簡潔さ、読み易さ、検証の容易さなどの点で優れた性質を持っているが、現行のマシン上での実現効率の悪さから、従来型言語程の普及をみていない。データフローマシンは問題の持つ並列性をそのまま抽出でき、且つ関数型プログラムを効率良く実行しうるマシンとして期待されている。筆者らは、データフローマシンのリスト処理への適用を目指し、構造メモリを用いたリスト処理向きデータフローマシンのアーキテクチャ〔4〕を提案するとともに、Lenient cons，Lazy cons 等の先行・遅延評価機構の実現法並びにその有効性を示した〔5,6〕。

データフローマシンは関数型言語の基礎になる call-by-value 型の計算を効率良く実現する手段であるが、これをより高い計算能力を持つ call-by-name 型計算に適用するには、ハードウェア技術・コスト等の面で多くの問題がある。

以上の観点から、我々はデータフローマシンの計算能力を最大限活用するような関数型言語 Valid〔7〕の開発を進めてきた。本稿では、Valid の言語設計について述べ、言語の評価を目的としたノイマン型マシン上での言語処理システム（Valisp：Valid to Lisp translator）のインプリメンテーションを中心に議論する。特に本稿では、Valid における遅延評価・パタン照合機構の効率的な実現法について論じ、その応用例を示す。

2．Valid の特徴

2．1　Valid の基本思想

Valid は 記号処理と数値処理を再帰概念という統一した枠組みでとらえるとともに、データフローマシンの持つ能力を最大限に活用するように設計された言語である。
プログラムとデータの分離による直接実行というデータフローマシンの特徴を生かすため、現 version の Valid では高階関数やパタン照合などの機能のサポートは直接実行可能な範囲に限定している。以下に Valid の設計方針を示す。

- 関数性を遵守する。
- 直観的にわかり易いシンタクスを与える。
- 値呼び（call-by-value）による引数授受を原則とする。
- 関数定義には自由変数を許さない。
- 自由変数に対しては静的束縛規則を用いる。
- 取扱うデータ構造はスカラ，配列及びリストとする。
- ユニフィケーションや高階関数等の一般的扱いはインタプリタで行う。

2．2　言語構造

Valid シンタクスは Algol 系言語を基調にしている。これは数学などで使われる記法との隔たりが少なく、また従来型言語に親しんだユーザに違和感を与えない点を考慮してのことであるが、そのセマンティクスはあくまで関数性を遵守したものになっている。Valid 言語構造の主な特徴を以下に述べる。

Valid の言語構造は定義と式からなり、プログラム自身１つの式である。定義には値定義・タイプ定義・関数及びマクロ定義がある。　また式には、tuple 式・block 式・条件式・再帰式・並列式の他、return 式・recur 式・値の参照・関数適用及びマクロ展開等が含まれる。

A．1　定義

プログラムは数学的な関数概念に基づいて構成されており、変数は値や関数本体などの実体を指す名前として用いられる。Valid における名前（変数）の定義は一意的でなければならない。

（1）値定義（value definition）

x＝ expression 又は〔x1,..,xn〕＝ expression

値定義ではこのように、expression が返す複数の値にそれぞれ x1,..,xn の値名を付与することができる。但し、x ＝x＋1 や x=y ； y=z ； z=x のような循環定義は許さない。

（2）タイプ定義（type definition）

typename：type ＝ type-specification

この定義により、typename は type-specification で指定された構造のデータタイプを示すことになる。Valid では以下のように任意の変数 var に対し、必要に応じてデータタイプを指定することができる。

var ：typename　又は var： type-specification

（3）関数定義（function definition）

f ：function (x1,..,xn) return (t1,..,tm)
＝ expression

関数定義は関数名とその本体を定義する。関数本体は複数の値を返すことができ、それ自身１つの式である。上記の定義により、x1,..,xn を仮引数（必要ならそのタイプを指定）とし、t1,..,tm のタイプのm個の値を生成する関数 f の本体が expression で与えられることが定義される。関数定義では再帰的定義を行うことができるが、関数名以外の自由変数を持つことは許されない。プログラムテキスト中に関数名の参照があれば実行時にその関数本体で置き換えられる。

(4) マクロ定義 (macro definition)

f : macro (x1,..,xn) return (t1,..,tm) = expression

関数定義と同様。但し、マクロ名の参照はコンパイル時にマクロ本体に展開される点が関数定義と異なっている。

A. 2 式

(1) tuple 式 (tuple expression)

(e1, ... ,en)

tuple 式とは式の組であり、1つ以上の式を", "で区切って並べ"("と")"で囲ったものである。一般に式は複数個の値を生成する。例えば、式 ei が ki 個の値を生成する場合、(e1,..,en) は $\sum_{i}^{n} ki$ 個の値を生成することになる。

(2) block 式 (block expression)

clause
  definition 1 ; ・・・; definition m ;
  return-expression ;
  definition m+1 ; ・・・; definition n ;
end

Valid はブロック概念を持ち、各種定義をブロック内に局所化することができる。ブロックを示す clause , end は { , } と略記してもよい。ブロック自身式であり、ブロックの返す値は次の return 式により示される (return は省略可能)。 retun expression

ブロック内における値定義の評価順序は指定されず、データ依存関係のみに基づき並列に評価される。

(3) 条件式 (conditional expression)

i. if 式 (if expression)

if predicate 1 then expression 1
elsif predicate 2 then expression 2
  ・・・
else expression 0

if 式では述部は並列に評価される。複数の述部が真になった場合は最も若番の述部に対応する式が評価され、その値が if 式の値となる。述部の評価を逐次的に行いたい場合は elsif を else if で置き換えた構文を用いる。

ii. case 式

case
  predicate 1 → expression 1 ;
  ・・・
  predicate n → expression n ;
〔;others → expression 0〕
end

〔 〕内は省略可能。case 式と同様に述部は並列に評価され、全ての述部が偽であれば expression 0 が評価されることになる。但し、複数の述部が真になるとそのうちの1つが非決定的に選択される。

(4) 再帰式 (recurrence expression)

for (recur-vars) : (init-value-exprs)
do recur-body

Valid では、末尾再帰構造も非末尾再帰構造も再帰概念に基づき、上記再帰式又は再帰関数によって記述される。

再帰式は無名の再帰関数を定義するものである。ここで recur-vars は再帰本体 (recur-body) の各インスタンスへの入力値を示す変数であり、再帰本体内で局所的に使用される。再帰式への初期値は init-value-exprs により指定される。再帰本体は1つのブロックであり、通常、再帰の停止条件を判定する条件式と次の recur 式を含んでいる。

recur expression

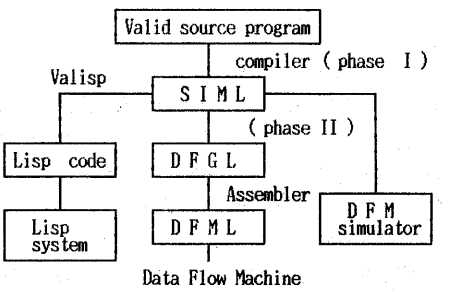
(5) 並列式 (parallel expression)

for each elem-vars in set-exprs
do parallel-body

並列式はベクトル演算のような並列実行可能な処理単位を fork-join 概念に基づき陽に記述する便利な手段である。ここで set-exprs は1..5のような range または list タイプの式 (の並び) であり、elem-vars は set-exprs により生成される集合の要素値を示す変数 (の並び) である。(但し、elem-vars は parallel-body 内の局所変数。) 集合の各要素は各 parallel-body に引渡され並列に計算される。各 parallel-body で生成された値は、set-exprs のタイプに応じて配列またはリスト構造に作り上げられる。

3. Valid 言語のインプリメンテーション

Valid 言語処理系の構成を以下に示す。



Valid ソースプログラムはコンパイラ (phase I) により s式表現された中間言語 (SIML) に落とされ、SIML からそれぞれデータフローマシンの機械語, ソフトウェアシミュレータ及び Lisp のコードに変換される。

以下では Valid プログラミングシステム (Valisp) のインプリメンテーションを中心に述べる。

3．1　Valisp　作成の目的

Valisp　は Valid　ソースプログラムから Lisp のコードへ変換するトランスレータで、以下の目的で作成された。

・Valid 言語によるプログラミング環境を提供する。
・Valid 言語の記述能力の検証を行う。
・データフローマシンによる処理方式を検討する際の、思考実験ツールとする。

使い易さ，編集・デバグ機能の面を考慮し、ここでは Dec 上の Maclisp　を用いて作成した。 Valisp では Maclisp で提供されている関数が一部を除いて殆どそのまま使用可能である。現在、Emacs を一部修正した Valid　用エディタ (Valedit と呼ぶ) 及び、Rosetta Smalltalk と類似の機能をもつ簡易マルチウインドウなどのプログラム環境が用意されている。

3．2　Lisp　コードへの変換規則

Valid ソースプログラムから Lisp コードへの変換規則を appendix に示す。変換においては、Valid 言語のセマンティクスを忠実に反映し、関数性を遵守するようにした。
ここでは変換規則の詳細説明は省き、主な特徴だけを述べる。

・ tuple　式はリスト表現し、値定義においてリストの各要素を分解する。

・block 式は逐次評価の let＊　に変換する。但し、block 式中の値定義についてはデータ依存関係を解析し、依存関係に応じて並べ換えを行う。

・case　式は現時点では非決定性を実現しておらず、if　式と同様の変換を行っている。

・再帰式については、末尾／非末尾構造を検出し、それぞれ do　ループ及び label　式（再帰式は無名の再帰関数を定義するので実現上は関数名を内部で生成することになる）に変換する。

・並列式は mapcar を用いて実現する。

・高階関数の実現においても、　関数と環境の組である閉包 (closure) を作ることはしていない。閉包がつくられるのは、4．章で述べるように、データフローマシン上での効率的実現が可能である遅延評価に対してのみである。従って現時点では、引数とし受取り結果として返せる関数は自由変数のないものに限られる。

・自由変数をもつ式・関数の評価では、i．静的束縛規則により自由変数に値を代入し自由変数をなくしてしまう，ii．閉包をつくる，の２つ方法がある。 Valid　ではデータフローマシンでの実現性を考慮し、i．の方法を採っている。この方法は　block 式中に自由変数が現れると、次のように束縛変数の名前の変更を行い、全て束縛変数に変換することに対応する。

```
{x=1 ; {x=2 ;a *3} ; a=3}
==> {x=1 ;xx=2 ;a *3 ;a =3}
```

Valid　では関数定義内の自由変数を許していないが、コンパイル時にこれ対する値を引数として受け取るようなプログラムに変換することにより、この制約を取り除くことが可能になる。

3．3　パタン照合

論理型言語の特徴の１つであるパタン照合によるデータ構造操作の機能は関数型言語でも取入れることが可能である。ここでは、Valid　のシンタクスを拡張しパタン式及びパタン定義を導入する。

i．パタン式

パタン式は次のようなリスト表記法を用いて表現する。

```
(e1 . e2) == cons (x, y)
( ) == nil
(e1 , e2 . e3) == (e1 . (e2 . e3 ) )
```

ここで ei は任意の式であり、パタン式はリスト構造で実現される。また、パタンの略記法として次の表記を許す。

```
(n .. m) == (n . (n+1 .. m) )
(n ..) == (n . (n+1 ..) )
```

ii．パタン定義

パタンに対する値定義（パタン定義）は　block 式，関数の仮引数など値定義が行われる任意の場所に書くことができる。但しパタンの受け側では、変数又はアトムしか許されない。パタン照合のルールは次の通りである。

```
(x . y ) = (e1 , e2 , e3)
    ==> x=e1, y = (e2 , e3 )
( ('a. y ) . z ) = ( (e1 . e2 ) . e3)
    ==> y=e2, z =e3 if eq ('a, e1)
```

パタンの受側が変数であれば任意の式と照合する。パタン定義中にアトムが現れるか，または同じ変数が出現した場合は同一性の検査を行う。照合に失敗すれば error　として例外処理に入る。Lisp　による実現では、appendix - 11．に示すようにパタン定義は個々の値定義に分解され照合検査のコードが生成される。　例えば ('a , x . y) = e　は、x =cadr (e) , y =cddr (e) に分解され、eq ('a, car　(e)) に対するコードが生成されることになる。

Valid　におけるパタン照合は単方向のユニフィケーションに限定されているが、car , cdr 等のデータ構造の選択オペレータが省略できるので、直観的に分かり易いプログラムをかくことができる。現時点では照合に失敗した場合、例外処理に制御を渡すようにしているが、非決定性処理機構を導入しパタン照合を関数起動や case 式にも適用する方式を検討中である。

4. 遅延評価

関数評価の方法には最内側評価 (innermost evaluation), 最外側評価 (outermost evaluation) などの一般的評価法の他、先行評価・遅延評価などの名称で呼ばれる評価法がある。

最内側評価では、すべての実引数を評価した後に関数の評価に入る。 call-by-value による引数授受を基本とする pure data driven 型計算がこれに相当する。 先行評価は call-by-value による引数授受に non-strictness 概念を導入し、引数の値が定まらなくとも先に関数の結果を返すようにしたものである。関数の部分実行や Lenient cons 〔4〕機構はこの有効な実現法であり、最大限の並列性抽出が可能になる。 最内側評価は一般に実現が容易な反面、停止性・資源の浪費などの点に問題がある。

一方、最外側評価では外側の関数適用に対する評価を先に行い、必要に応じて引数の評価を行う。この評価法は call-by-name や call-by-need による引数授受機構を実現することにより達成される。最外側評価では、最内側評価における停止性の問題が解決されるなど計算能力の点で優れているが実現効率の点では問題が多い。

遅延評価は call-by-value による引数授受を原則とする関数評価において、実引数の評価を遅延させることにより、call-by-name や call-by-need のような引数伝達機構を実現しようとするものである。Lazy cons 〔8〕はこの一例である。

先行評価. 遅延評価は並列処理において重要な評価法である。これらを組合せることにより、与えられた資源のもとで最大限の並列処理性を引出す、効率のよい処理方式の実現が可能となる。次節では、 Valisp における実現法及び制御法について述べる。

4. 1 遅延評価機構の実現

遅延評価は関数評価時に、実引数に対する式と環境の組 (閉包) を recipe として引渡し、引数値が必要になった時点でそれを評価することによりなされる 〔2, 3〕。データフローマシン上では、これを引数トークンの gating 及び demand の送出によって実現している 〔8, 9〕

自由変数に対して静的束縛規則を用い、式或いは関数の評価の際に必ず閉包をつくる場合には、遅延評価制御オペレータ delay, force は次のように定義される 〔3〕。

```
delay == (lambda (e) (lambda () e))
force == (lambda (e) (e))
```

Valid では call-by-value を基本とし、式・関数の評価において閉包をつくる方式をとっていない。したがってここでは、〔3〕のような方式のインタプリタを作成せず、Lisp の評価機構を利用してデータフローマシン上での実現法に近い形で インプリメントした (appendix - 9. 参照)。

・ delay e は次の4項組のs式 (recipe と呼ぶ) に変換する。

```
(list 'recipe' (lambda (x1 .. xn) e')
              (list x1 .. xn))
```

ここで x1,..,xn はe内の自由変数, e'はeのs式表現であり、 (list x1 .. xn) により定義時の自由変数の値を環境として組込んでいる。

・ force e は (force e') に変換する。 force は Lisp の関数として定義され、 e' が recipe であれば apply により定義時の環境を使って e' を評価する。

4. 2 遅延評価の制御

・delay, force による明示的制御

式又は関数適用に対する評価の遅延・開始の指定は次のように delay, force を用いて行う。

```
f (delay g (x, y))   又は

{u = delay g (x, y) ;  f (u)}
```

これにより、 g (x, y) は未評価のまま関数 f に引渡される。 f 内でこの値が必要であれば force u と指定する。

〔x,y〕= delay e のように複数の値を返す式または関数 e が遅延される場合は、x または y のいずれかの値が必要になった時点で e の評価が開始される。

しかし複数の値定義式は appendix - 6. のように変換されるため、x == 'recipe , y == ' (lambda .. e') となりこのままではうまく動作しない。そこで上記値定義は appendix - 9. 3に示すように、

```
u=delay e ; x =delay car (force u) ;
          y =delay cadr (force u) ;
```

と等価なコードを生成し、x, y の値が必要とされる (x, y が参照される) 場所に force x, force y に対応するコードを埋込むようにしている。

・cons に対する遅延制御

delay, force を用いた場合、一般に制御が複雑になりプログラムの意味も不透明になる。 Valid ではこれをユーザに開放せず、代わりに cons に対する遅延評価指定法として次に定義される strcons, 及び lazycons を提供している。

```
strcons (x, y) == cons (x, delay y)
lazycons (x, y) == cons (delay x, delay y)
```

これを用いることより、無限リストで表される stream を記述することが可能になる。

stream に対する操作は、次の関数を用いる。

first (x) == car (x)

rest (x) ==force cdr (x) (single forcing)

head (x) ==force car (x) (recursive forcing)

tail (x) ==force cdr (x) (recursive forcing)

これらの関数の Lisp による定義は appendix-9. 2 B を参照。ここではセルの car 部又は cdr 部が recipe セルであれば直接、rplaca, rplacd によりrecipe の評価値に書替えを行うことにより、call-by-need を実現している。例えば、ints (n) == strcons (n , ints (n+1))

に対し、

{x =ints (1) ; u=rest (x) ;v =rest (x) ・・}

を評価する場合、u =・・, 又は v=・・のいずれか一方により1回だけ recipe が計算され、他方はその値を参照するだけでよいことになる。

5. 遅延評価機構の応用

遅延評価機構を用いることにより、stream 処理やコルーチンなどの制御が容易に実現される。Program 1に Valid による stream 処理の記述例を示す。

rstream は端末から系列要素を1つずつ受取り、 それを stream として送出する。 stream は、strcons によって無限リストの形で形成されている。 compact は端末からの stream を受取ると重複する要素を remove で除去し短縮化された系列を生成する。この短縮化された系列は wstream によって端末に出力される。

図1は Program 2に示したコルーチン制御プログラムを実際に走行させた例である。 ここでは fibonacci 数列, パスカルの三角形, 素数列の3種の無限系列を生成するプロセス (Program 1と同様に strcons を用いて記述される) が起動され、端末からのメッセージコマンド (各数列の出力要求) に応じウインドウに指定された個数の数列を出力する様子が示されている。 例えば、fibonacci プロセスは ['fib,5] (fibonacci 数列を5個出力せよ) というメッセージを受取ると、1,1,2,3,5 の数列を出力し次のメッセージが送られるまで中断する。 再び ['fib,3] が送られるとその後の数列 8,13,21を出力する。プロセスの中断・再開は delay,forceの機能による。データフローマシン上での遅延評価機構の実現及び応用については文献 [6, 9] を参照されたい。

6. あとがき

本稿では、データフローマシン用の関数型言語 Valid の設計並びに、関数型プログラミングの実験ツールとして Dec 2060 上に作成した Valid 言語システム (Valisp) について述べ、 Valid における遅延評価・パタン照合機能の効率的な実現法を示した。Valisp は Valid プログラムをLisp のコードに変換して実行する形式をとっている。

現在、関数型言語と論理型言語の融合を目指し、パタン照合機構, 非決定性処理機構等を導入した Valid 第二版の言語仕様を固めているが、これについては別の機会に報告する。

最後に御討論頂いた第二研究室の諸氏に感謝致します。

参考文献

[1] J.Bacus , Can Programming be liberated from the Von Neumann style ? A functional style and its algebra of programs, CACM vol 21, No.8, 1978.

[2] P.Henderson, Functional Programming Application and Implementation, Prentice-Hall , 1980.

[3] D.S.Wise, Interpreters for functional programming J.Darlington, P.Henderson and D.A.Turner edts, Functional Programming and its Application, Cambridge University Press, 1982.

[4] M.Amamiya,R.Hasegawa,O.Nakamura and H.Mikami, A list-processing-oriented data flow machine architecture", AFIPS NCC, pp.143-151 (1982) .

[5] 長谷川, 雨宮, データフローマシンによる並列リスト処理, 信学論 (D) ,J66-D,12,pp.1400-1407, (昭58-12)

[6] M.Amamiya and R.Hasegawa, Data Flow Computing and Eager ／Lazy Evaluation, New Generation Computing , vol 2, No.2,1984. To appear.

[7] 雨宮, 尾内, データフローマシン用高級言語 VALID について, 信学技報, EC82-9 (1982-05) .

[8] 長谷川, 雨宮 データフローマシン上での Lazy Evaluation の実現について, 第24回情処全大5D-8.

[9] 雨宮, 長谷川, 清木 データフロー・アーキテクチャに於ける先行評価・遅延評価機構とその並列推論制御への応用, 信学技報 EC83-36,1983.

Appendix Valid to Maclisp Translation Rules

Note.
Here, a s-expression for a valid expression e is denoted by e'.

1. simple expression

1.1 operator application (prefix form)
```
op(x,y) ==> (op x' y')
```
1.2 operator application (infix form)
```
x op y ==> (op x' y')
```
1.3 function application
```
f(x,y,z) ==> (f x' y' z')
```

2. tuple expression
```
(e1,..,en) ==> (list e1' .. en')
```

3. block expression
```
clause
  x1 = e1;
    ..     ==> (let*
  xn = en;       ((xi1 ei1')
  return e         ..
                  (xin ein'))
                    e)
end

 -- Here, (xij eij') is one of
 --          {(x1 e1'),..,(xn en')}.
 -- each (xij eij') pair is arranged
 -- according to data dependency.
 -- clause .. end construct may be
 --                written as { .. }.
ex.1
clause            (let* ((z 3)
   x = f(y,z);           (y (g z))
   y = g(z);  ==>        (x (f y z)))
   z = 3;              (+ x 1))
   return x+1
end
```

4. conditional expression
4.1 if-then-elsif
```
if p1 then e1          (cond (p1' e1')
elsif p2 then e2 ==>         (p2' e2')
  ...                          ..
elsif pn then en             (pn' en')
else e                       (t e'))
```
4.2 case (translated same as above)
```
case
  p1 -> e1;            (cond (p1' e1')
    ..           ==>          ..
  pn -> en;                 (pn' en')
  others -> e               (t e'))
end
```

5. function definition
```
f:function(x1,..,xn) return(typ1,..,typn)
=e;
==> (defun f (x1 .. xn) e')
```

6. multiple value definition
```
{[x1,..,xn] = e; ... }
==> (let* ((res e')
           (x1 (car res))
              ..
           (xn (cad..dr res))
            ... )
```

7. recurrence expression
```
for (x1,..,xn) : (e1,..,en) do e
-- Here, e includes recur expression
```
7.1 tail recursion case
```
==> (let (G01 .. G0n)
      (do ((x1 e1' G01)
              ..
           (xn en' G0n))
          (nil)
          (return e')
        loop))
-- Here, e' includes a form of (go loop)
```

7.2 non-tail recursion case
A. pure non-tail recursion
```
==> ((label F00 (lambda (x1.. xn)
                   e')) e1' .. en')
```
B. mixture of tail and non-tail recursion
```
==> ((label F00 (lambda (x1 .. xn)
        (let (G01 .. G0n)
           (do ((x1 x1 G01)
                   ...
                (xn xn G0n))
               (nil)
               (return e')
             loop)))) e1' .. en')
 -- Here, e' includes both (go loop)
 -- and (funcall F00 ..)
```
ex.1 (tail recuresion)
```
for (x) : (e) do
  if p then x
  else recur(x+1)

==> (let (G01)
      (do ((x e G01)) (nil)
        (return
         (cond (p x)
               (t (setq G01 (+ x 1))
                  (go loop))))
        loop))
```
ex. 2 (non-tail recuresion)
```
for (x) : (e) do
  if p then x
  else {z=recur(f(x)); g(z)}

==> ((label F00 (lambda (x)
       (cond
        (p q)
        (t
          (let* ((z (funcall F00 (f x))))
              (g z))))))) e)
```
ex.3 (mixture of non-tail and tail recursion)
```
for (x) : (e) do
 if p1 then x
   elsif p2 then recur(x+1)
   else 2*recur(x-1)

==> ((label F00 (lambda (x)
      (let (G01)
       (do ((x x G01)) (nil)
         (return
          (cond
           (p1 x)
           (p2 (setq G01 (+ x 1))
               (go loop))
           (t
            (* 2 (funcall F00 (- x 1))))))
         loop)))) e)
```

```
8. parallel expression
   for each u in x do e
   ==> (mapcar '(lambda (u) e') x)
  ex.1
  for each u in '(1 2 3) do u+1
   ==> (mapcar
        '(lambda (u) (+ u 1))
        '(1 2 3))

9. lazy evaluation
 9.1 delay
  A. function application
   delay f(x1,..,xn)==>(delay (f x1 .. xn))
  delay of which definition is omitted
  here yields
  ==> (list 'recipe
            '(lambda (x1 .. xn) (f x1 .. xn))
            (list x1 .. xn))
  B. any expression
   delay e
   ==> same as above.
       (list 'recipe
             '(lambda (x1 .. xn) e')
             (list x1 .. xn))
   -- Here, x1,..,xn are free variables in e'
  ex.1
   delay x+1
   ==>(list 'recipe '(lambda (x) (+ x 1))
  ex.2
   delay {a=f(x); return a}
   ==>(list 'recipe
         '(lambda (x) (let* ((a (f x))) a))
                                  (list x))
 9.2 force
  A. explicit force
  force x  ==> (force x)
   -- Here, force is defined as follows.

  (defun force (x)
   (cond ((closurep x)
          (force (apply (lmdbody x)
                        (lmdargs x))))
         (t x)))
  (defun closurep (x)
   (cond ((not (atom x))
          (eq (car x) 'recipe))
         (t nil)))
  (defun lmdbody (x) (cadr x))
  (defun lmdargs (x) (caddr x))
  B. implicit force
  use the following functions
  defined below.
  B.1 single forcing for
      strcons (stream cons)
   rest(x) ==> (rest x)

  (defun rest (x)
   (cond
    ((closurep (cdr x))
     (apply (lmdbody (cdr x))
            (lmdargs (cdr x))))
    (t (cdr x))))
   or
  (defun rest (x)       ; efficient version based
   (cond                ; on graph reduction
    ((closurep (cdr x)); concept
     (cdr
      (rplacd x
              (apply (lmdbody (cdr x))
                     (lmdargs (cdr x))))))
    (t (cdr x))))

  B.2 recursive forcing for suspending cons
   head(x) ==> (head x)
   tail(x) ==> (tail x)

  (defun head (x)
   (cond ((closurep (car x))
          (car (rplaca x (force (car x)))))
         (t (car x))))
  (defun tail (x)
   (cond ((closurep (cdr x))
          (cdr (rplacd x (force (cdr x)))))
         (t (cdr x))))


 9.3 delayed multiple value defnition
    {[x1,x2,..,xn] = delay e;
        ...
     g(x1,..,xn)}

  A.
  ==>(let* ((res (delay e'))
            (x1 (delay (car (force res))))
                       ...
            (xn (delay (cadd...dr (force res))))
                       ... )
       (g (force x1) .. (force xn)))
  B.
  ==>(let* ((res (delay e'))
            (x1 (delay (car* res)))
                       ...
            (xn (delay (car* (cdd..r* res))))
                       ... )
       (g x1 .. xn))

    (defun car* (x) (car (force x)))
    (defun cdr* (x) (cdr (force x)))
   --Note that in case of B, each argument
   --of an application should be forced.


 10. higher order function

  g:function (f:function,x) return (type)
  =  ... f(..) ...
  ==> (defun g (f x) ... (funcall f ..) ...)

  ex.1
   g:function (f:function,x) return (list)
   = if null(x) then nil
     else cons(f(car(x)),g(f,cdr(x)));

   ==> (defun g (f X)
        (cond ((null x) nil)
              (t (cons (funcall f (car x))
                       (g f (cdr x))))))

 11. Pattern matching

  11.1 patten expression
    [e1.en] ==> (cons e1' en')
    [e1,..en] ==>(cons e1'
                       (cons .. (cons en' nil)))

  11.2 pattern definition
    [x1.x2] = e
    is equivalent to
       x1 = car(e); x2 = cdr(e)
    [x1,..xn] = e
    is equivalent to
       x1 = car(e); .. ; xn = cad..dr(e)
```

```
   -- Note that if xi is a quoted atom
   -- or already
   -- defined variable in the
   -- pattern definition
   -- then a code to check equality
   -- is generated.

 ex.1
  {['tom.x]=e1; [y.y]=e2; f(x,y)}
  ==> (let* ((x (cdr e1))
             (y (car e2)))
        (cond ((and (eq (car e1) 'tom)
                    (eq y (cdr e2)))
               (f x))
              (t (exception))))
```

```
-- Program 1  compact stream --

rstream: function(sig:signal)
         return(stream)
= strcons(read(sig),rstream(sig));

wstream: function(s:stream) return(stream)
= for (x):(s) do
  if null(car(x)) then nil
  else {s0=princ("   ");
        s1=princ(car(x));
        s2=terpri();
        recur(rest(x))};

compact: function(s:stream) return(stream)
= if null(s) then nil
  else strcons(car(s),
               compact(remove(car(s),
                              rest(s))));

remove: function(x,s:stream) return(stream)
= if null(car(s)) then nil
  elsif eq(x,car(s)) then remove(x,rest(s))
  else strcons(car(s),remove(x,rest(s))) ;

 wstream(compact(rstream(nil)));
```

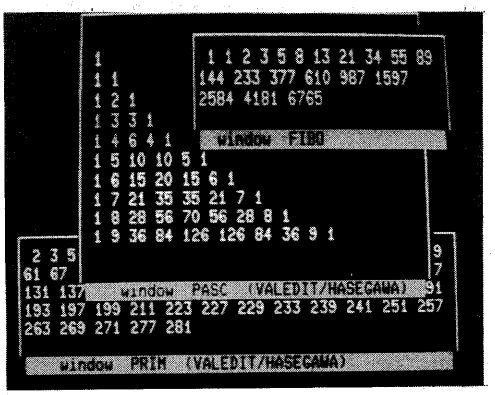


図1　コルーチンの実行例

```
-- Program 2  coroutine --

coroutine: function(sig:signal) return(stream)
= { sg=wcreate(sig);
    m=inp(sg);
    u=fibo(sg);
    v=pasc(sg);
    w=prim(sg);
    cor(m,u,v,w) };

inp: function(sig) return(stream)
     = [read(sig).inp(sig)];

cor: function([[cmd,cnt].nmsg],u,v,w) return(stream)
= case
   cmd == 'fib -> cor(nmsg,outfib(u,cnt),v,w);
   cmd == 'pas -> cor(nmsg,u,outpas(v,cnt),w);
   cmd == 'pri -> cor(nmsg,u,v,outpri(w,cnt));
   others -> cor(nmsg,u,v,w)
  end;

outfib: function(u,cnt) return(stream);
        = outs(u,cnt,'fibo);
outpas: function(u,cnt) return(stream);
        = outs(u,cnt,'pasc);
outpri: function(u,cnt) return(stream);
        = outs(u,cnt,'prim);

outs: function([e.x],cnt,wnam) return(stream)
      = if cnt == 0 then [e.x]
        else { s0=wprinc(" ",wnam);
               s1=wprinc(e,wnam);
               outs(x,cnt-1,wnam) };

fibo: function(sig) return(stream)
      = [1,1.addlist(fibo(sig))];
addlist: function([e,x1.x]) return(stream)
         = [e+x1.addlist([x1.x])];

pasc: function(sig) return(stream)
      = pastri([1.zeros(sig)]);
pastri: function([v1.v]) return(stream)
        = [[v1.v].pastri([1.vsum([v1.v],v)])];
zeros: function(sig) return(stream)
       = [0.zeros(sig)];
vsum: function([u.x],[v.y]) return(stream)
      = [u+v.vsum(x,y)];

prim: function(sig) return(stream)
      = sieve(2,ints(2));
ints: function(m) return(stream)
      = [m.ints(m+1)];
sieve: function(n,[e.m]) return(stream)
       = [e.sieve(n,del(e,m))];
del: function(m,[e.n]) return(stream)
     = if remainder(e,m) == 0 then del(m,n)
       else [e.del(m,n)];

-- Note 1.
--  [x.y] is the abbreviation of strcons(x,y).
--  pattern definition [x.y]=exp means
--  that x=car(exp) and y=rest(exp)
--  Note that cdr part of exp is forced.

-- Note 2.
--  wcreate is a function which creates
--  windows for fibo, pasc and prim.
--  wprinc outputs an object to the
--  specified window.
```