シンタクティック・シュガーによる手続き定義と内部手続き・高階手続き機能を持つ拡張 Logo 言語†

山本順人†† 中山和彦††

近年，教育の場において使用されるようになってきている Logo は，簡素な構文と高いアルゴリズム記述能力を備えた言語である．しかし，計算機言語として Logo を見たとき，上に述べた特徴がよりよく発揮され，大型のプログラムの作成に使用されるためには，情報隠蔽を巧みに行うためのプログラム構造を持つ必要がある．また，高階性を利用したプログラミングが行いうれば，より複雑高度な処理を記述することができよう．本稿では，従来の Logo 言語に考察を加え，これらの機能を備えた新しい Logo の試みにつき述べる．まず，処理の基本であるワードの解釈と手続き定義の方法につき，シンタクティック・シュガーを導入した形の実現を行った．すなわち，処理系内部における「リーダ」と「評価器」の役割の配分を整理し，評価器の解釈動作を手続き呼び出しとして統一した．さらに，従来不明確であった手続き本体の構造を，「手続きオブジェクト」として定義することにより，手続きを定義することの意味および動作を明確化しえた．このことがもたらす効果として，内部手続きを容易に定義することが可能となり，プログラム構造上のモジュラリティ向上に有効な手段となりえた．また，クロージャの形で目的とする手続きオブジェクトを作成する高階手続きも，同様な方法で定義することが可能となった．これらの機能は，実際の処理系を作成し試用することにより，その有効性を確かめえた．

1. はじめに

近年，教育の分野において，「情報」に対する関心が集まっており，学校教育へのコンピュータの導入が盛んに行われつつある．その中で，コンピュータ・リテラシー教育の一環として，Logo 言語が使用されるようになってきた．

この Logo 言語は，1968 年，米国科学財団の後援により，BBN 社が実施した研究プロジェクトの成果として誕生した．それ以来，MIT の人工知能研究所が中心となり研究を進めてきている．その中で特に，MIT メディア研究所のシーモア・パパート (Seymour Papert) は，ピアジェ理論を背景とし，教育の分野における Logo の適用に，数々のプロジェクトを推進しており[12]，MIT 電気工学計算機科学部 (Department of Electrical Engineering and Computer Science) のハロルド・エーベルソン (Harold Abelson) は，言語自体の開発およびタートル・グラフィックスとして知られているグラフィカル・インタフェースについての理論付けを行っている[1]．Logo に関する研究は，その後英国エディンバラ大学やロンドン大学[11]等でも盛んに行われている．興味深い例として，エディンバラ大学では，AI の講義に Logo が利用されている[6]．

Logo は，小さな手続きを組み合わせることにより，より大型のプログラム・モジュールを構成するスタイルの言語で，手続きの再帰構造を取り扱えることと合わせ，潜在的にはアルゴリズムの記述能力に優れており，その構文は非常に簡素で，セマンティックスが明解であるという特徴を有している．しかし，一方，高速演算，大容量記憶が容易に利用しうる計算環境となりつつある現在，プログラムの複雑化，大型化に対応する記述能力は，近代的なプログラミング言語にとっては必須の要件と考えられるが，従来からの Logo 言語においてもこれらの点を整備する必要が生じている．特に，情報隠蔽を行うための内部手続きやそれを含むオブジェクトの生成等の概念は，プログラム構造の設計上重要であるにもかかわらず，残念ながら現在含まれてはいない．しかし，Logo 言語の利用される場面を考えたとき，上述の要件が実現されたならば，幼時に初めて触れたこの計算機言語が，その後，自然な延長線上で実用の言語として使用することが可能となり，その教育的効果は大と言えるであろう．

そこで本稿では，従来からの Logo 言語に考察を加え，上に述べたごとく，新しい機能とアルゴリズム記述能力を持つ言語として構築することを試みた．その主たるものは，

(1) シンタクティック・シュガーを可能な限り利用し，処理系，特に評価器 (evaluator) の動作の簡約

† Defining Procedures in an Extended Logo: Utilization of Syntactic Sugar and Support for Internal Procedures and Higher-Order Procedures by NOBUHITO YAMAMOTO and KAZUHIKO NAKAYAMA (Institute of Information Sciences and Electronics, University of Tsukuba).

†† 筑波大学電子・情報工学系

化をはかる.

(2) 内部手続きの定義を可能にすることにより,プログラムのモジュラリティを向上させる.

(3) 手続きそのものを処理の対象とする高階プログラミングを実現する.

(4) オブジェクトの生成・取り扱い機能を装備する.

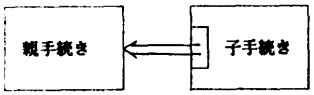
等である.

2. Lisp 族の一員としての Logo

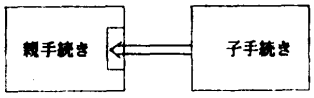
Logo はその生い立ちから見ても, Lisp の影響を深く受けていることは容易にうかがい知れる. 類似している点としては, 前置記法に基づいた簡単な構文であること, 手続き呼び出しを動作の基本としていること等がまず挙げられる. この見かけ上の類似が, Logo を Lisp の「子供版」とみなす傾向を形づくる主要因になっていると思われる. ここでは Logo と Lisp の差異に焦点をあてることにより, Logo の持つ計算機言語としての特徴と, それを基にした新しい姿を考察するものである.

Lisp においては,関数型パラダイムに示されるごとく, すべての手続きは返り値を持っている. この返された値は, 呼び出し側で利用するか棄却するかが決められる構造となっている. 一方, 従来の Logo においては, プリミティヴは値を返さない「コマンド」と値を返す「リポータ」に区別され, 各手続き中の文は命令すなわちコマンドで始まる列で構成されるという事実から, 手続きは基本的には返り値を返さない. つまり呼び出された側がそれを結果として出力するかどうかを決めなければならない. このため, Logo は厳密な意味での関数性を有しているとは言えない (**図1**).

また, プログラム構成上の基本要素としてのシンボル (Logo ではワード, Lisp ではアトム) の取り扱い



送り出し型 (Logo)



受け取り型 (Lisp)

図 1 返り値の受渡し
Fig. 1 Return value handling.

	Logo	Lisp
(I) シンボル名	"x	'x
(II) シンボルの値	:x	x
(III) 手続き呼び出し	x	(x ···)

図 2 シンボルの三形態
Fig. 2 Three forms of symbol.

が評価器の動作という点において根本的に異なっている.

Logo のワードには三形態が存在する. **図2**に, Lisp の場合と対照させた三形態を示す.

(I)は, シンボル名そのものを表したいときの形である. Logo においても Lisp においてもクオーティングすることにより, その印字名を得ることができる.

(II)は, そのシンボルに結びつけられた値を取り出したいときの形である. Logo においては「:」がシンボル名につく. もしくは, 手続き thing の助けを借りることを前提とした形である.

(III)は, その名前を持つ手続きを呼び出すときの形である.

この図から明らかなように, Logo においては, ただ単にワードを評価すると, 手続き呼び出しが生じる(III). この形は, Lisp の場合には, 結びつけられた値を取り出すもの(II)であり, 大きな相違となっている.

さらに, Lisp の場合では, この状態(II)から手続き quote の助けで印字名そのものを(I), また,「(」,「)」で区切られたコンビネーションの先頭要素に置くことで, 手続き呼び出し(III)を表現することがなされている.

一方, Logo においては, 手続き呼び出し(III)を基底状態とし, (I)の「"」の意味付けは Lisp と同様に, 手続き quote の助けをかりて印字名そのものを表していると解釈されうる. また,「:」は, その印字名から他の手続き, 具体的には手続き thing の助けで, そのワードに結びつけられた値を取り出すこと(II)を行うと解釈できる.

3. ワードとシンタクティック・シュガー

Logo の文は, 形態的には, 上に述べたワードを並べることにより構成される. このことは, 文の解釈時において処理系が, 三つの異なる形態のワードを次々と読み込み, 処理せねばならないことを意味する. つまり, その中には, 手続き呼び出しの場合もあれば, 値の取り出しの場合もあり, これをそのまま直接的に

実現しようとすれば，処理系，特に評価器の構成を複雑とする要因となろう．

一方，評価器側からみた場合，ワードの最も基本的な解釈動作は，手続き呼び出しであるとみなすのが形態上自然である．すなわち，Logo の文は前章(Ⅲ)の形態であるワードの列として構成されることが望ましいと言えよう．

そこで，従来からのワードの表記法と，評価器の入力としての要求との間隙を埋める方法として，シンタクティック・シュガーを導入する．シンタクティック・シュガーは，表されている意味を変えることなく，表現のみを容易にする技法である．Logo は，「読み込み-評価-印刷」ループを動作の基本構造としており，文は通常，キーボードやファイルからリーダによって読み込まれ，評価器に渡される．この構造を利用し，文を読み込んだ後，リーダにおいて従来型の表現を実現しているシンタクティック・シュガーをはずせば，評価器の入力としての要求を満たすことが可能になる．すなわち，読み込んだ文に対し，

```
"x → quote x
:x → thing quote x
```

の変換を施せば，シンタクティック・シュガーが取り除かれる．ここに，thing は結びつけられた値を取り出す手続きである．

例えば，リスト b の最初の要素を a と命名する文，

```
make "a first :b
```

は，この対応付けに従えば，等価な文，

```
make quote a first thing quote b
```

と変換される．後者は，文がすべて形態(Ⅲ)のワードの列として構成されているので，これを入力とする評価器は，その基本動作としては，ワードに対する手続き呼び出し実行のみが行えれば，Logo の処理すべてを行いうることとなる．

よって，伝統的な Logo の構文である「"」や「:」を，評価器の動作としての意味付けをせず，リーダの処理するシンタクティック・シュガーとして取り扱うならば，それを受ける評価器の動作，意味付けが簡素化され，処理系作成を容易にすることができよう．

4. 手続きの定義

従来の Logo においては，複合手続き（以下単に手続き）は，[ ] で表されたリストで記述するようになっている．タートルの動きとして，「前に 100 歩進み，90 度右に曲がり，前に 200 歩進む」手続きは，

```
[fd 100 rt 90 fd 200]
```

と記される．この手続きを実行させるには，run 手続きを用いて，

```
run [fd 100 rt 90 fd 200]
```

とすれば実行される．しかし，引数の概念が入っていないため，この手続きを表しているリストを他の手続きにデータとして受け渡す目的には適せず，原理的な困難がつきまとう．そこで，より強力な表現を目的とし，Lisp と同様に，手続き本体を記述するために，チャーチ (A. Church) の lambda 式の記法を用いる．

組み込み手続き lambda は，

```
lambda [引数並び] [手続き本体]
```

の形式で，手続き本体に記された動作をする手続きオブジェクトを作り出す，と定義する．

手続きを定義するということは，一群をなす処理(動作) に対し，それを識別するための名前を与えることを意味している[2] と考えられる．ゆえに，本稿の Logo における手続き定義とは，

「lambda 手続きにより作り出された手続きオブジェクトに対し，名前（手続き名）を与えること，つまり，手続き名となる変数（ワード）と手続きオブジェクトを結びつけること．」

であるとする．

従来の Logo においては，変数名が保持される空間と，手続き名が保持される空間は，必ずしも同一ではない．つまり，変数の中に，手続き名を表すものと値を表すものの二種類の区別が存在している．もしこの二空間を併合し，同一命名空間内に置くことができれば，値の関係付けと手続き本体の関係付けとを，見かけ上，同じ記述で行うことが可能となる．

手続き名と手続きオブジェクトを結びつけるには，下記のように手続き make を用いる．

```
make 手続き名 手続きオブジェクト
```

make は結びつけをする手続きである．例えば，foo は 2 倍の計算を行う手続きとして定義したいとすると，

```
make "foo lambda [x] [2 * :x]
     手続き名  手続きオブジェクト
```

のように表現することができる．ここに，lambda は目的手続きオブジェクトを作り出し，make を用いて，手続き名 foo に関係づけられる．

一方，伝統的な Logo の構文では，

```
to 手続き名 <引数並び>
```

＜手続き本体＞
end

の形式で手続きが定義される．しかし，この形式はLogo の手続き呼び出しの一般形である，

演算子 ＜被演算子並び＞

とは異なり，スペシャル・フォームである．つまり，固有の処理を必要とする形式である．

この二者を比較したとき，to～end は make～lambda～ によるプリミティヴな表現と等価であり，かつ，単純な対応関係がつき，機械的に書き換え可能な表現であることが理解される．プログラミングという視点に立てば，to～end 表現は可読性において優れていると考えられるが，評価器の動作においては，make～lambda～ 表現は他の手続きの呼び出し実行と同じ形をしており，to～end を解釈する特別な（手続き定義）動作を必要としない点において好ましいものである．

そこで，ワードの三形態の場合と同様に，この to～end をシンタクティック・シュガーとして実現し，リーダで変換する方式とすれば，通常の手続きと同様に，手続きの定義は単に make 手続きの呼び出し実行として評価器に行わせることが可能となる．

5. 内部手続き

make により手続きを定義することができると，その定義方法の自然な拡張により，内部手続き，すなわち手続き内部で用いる手続きを定義することも容易に行えるようになる．このことは従来の Logo では行うことが困難である．しかし，内部手続きを始めとする情報の隠蔽は大きなプログラムを構造的に組み上げる上での主要な技法の一つであり，それを容易に行いうる能力は，本稿の Logo を特徴付けるものである．

以下の例では，入力を2倍し，さらに2乗する手続き f を考える．ただし，本稿の Logo の構文では，「！」の前置されたワードは仮引数を表す．またすべての手続きは最後に実行された結果を返り値として常に返す．

まず，従来の Logo のプログラミング・スタイルにおいては，手続き square と手続き double をそれぞれ（外部で）定義しておき，それを用いて手続き f を定義する(**図3**)．この場合，定義された各手続きは全域的であり，他の手続きから自由に参照されうる．

さて，make は手続きなので，f の中に書くことは許される．そこでこの make を使って，**図4**に示される形に f を記述することができる．この手続き中では，文①は手続き square を定義し，文②は手続き double を定義しており，文③によりそれら両手続きを呼び出す構造となっている．

```
手続き square:   to square !y
                   :y * :y
                 end
手続き double:   to double !z
                  2 * :z
                 end
手続き f:        to f !x
                  square double :x
                 end
```

図3 手続きの定義（外部）
Fig. 3 Defining procedures conventionally.

```
to f !x
  make "square lambda [y] [:y * :y] …①
  make "double lambda [z] [2 * :z]  …②
  square double :x                  …③
end
```

図4 内部手続きの定義（1）
Fig. 4 Internal procedure (1).

```
make "f lambda [x]
  [ make "square lambda [y] [:y * :y]
    make "double lambda [z] [2 * :z]
    square double :x ]
```

図5 内部手続きの定義（2）
Fig. 5 Internal procedure (2).

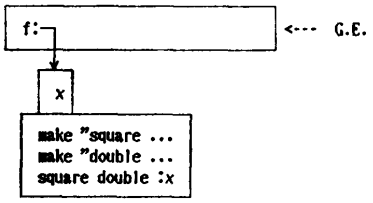
```
to f !x
  to square !y
    :y * :y
  end
  to double !z
    2 * :z
  end
  square double :x
end
```

図6 内部手続きの定義（3）
Fig. 6 Internal procedure (3).

make により作り出された手続きは，手続き f の内部でのみ有効な変数 square と double に結びつけられた手続きオブジェクトであるため，f の外部からは見えない手続きである．

シンタクティック・シュガーをはずした表現では，**図5**となる．つまり，手続き f の定義段階においては，f に関する手続き lambda と手続き make の実行のみが行われるだけである．内部手続きとしての square と double の定義動作は，この段階ではまだ行われていない．

実行前および実行後



実行中

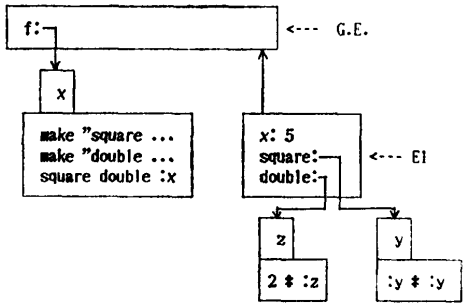


図 7 内部手続きの状態 ($x=5$ の例)
Fig. 7 Definition of internal procedure.

再び手続き f をシンタクティック・シュガーを用いて表現すると，**図 6** となり，通常の表現で，かつ内部で手続きが定義されている．

図 7，G.E. で指し示されたグローバル環境において，この定義された手続き f が呼び出され実行されると，新しく追加されたフレームを持つ環境 E1 において，はじめて，手続き square と手続き double の定義が行われる．そして，これらを使用して $(2*x)^2$ の値が算出され，手続き f の終了とともに消滅する．手続き f が再び呼び出されるときには，上述の動作が繰り返される．

6. 高階手続きの定義

手続きの基本的な振舞いは，入力されたデータを，所定のアルゴリズムに従って処理し，結果を出力データとして返すものである．この入力データとしては，数値や文字列，リスト等が選ばれ，出力データも同種のものが出力されるのが通常の場合であろう．しかし，解析学における微分操作を例にとれば，入力として関数（例えば sin 関数）が与えられ，処理の結果としては導関数（この場合 cos 関数）が得られている．つまりこの場合，入力データ，出力データは共に関数である．

高階手続きは，この考えと同じ基盤に立ち，入力として Logo の手続きそのものをとり，処理の結果として，やはり Logo の手続きを出力する，「手続きを取り扱う手続き」のことである．この高階概念を取り扱う能力は，従来の Logo の計算対象を大きく拡張するもので，今後のプログラミングにおいても重要となろう．

具体的な実現方法としては，Lisp におけるレキシカル・クロージャの考え方を継承しており，5 章で述べた内部手続き定義と同様な構造を作り出している．4 章で変数（ワード）には，値と手続きオブジェクトが同じ形式で結びつけられることを述べた．このことは，手続きの仮引数に数値（例えば 5）を与えることと，手続きオブジェクト（例えば square）を与えることが同じ動作原理で行いうることを示唆するものである．この機能を利用することにより，手続きの受渡しを行う高階手続きを定義する．

図 8 は，入力されたリストの各要素に対し，それぞれ同じ手続きを作用させ，結果をリストとして出力する手続きを作り出す高階手続き mapcar の定義例である．この手続き mapcar は，作用させたい手続きを引数 f として受け取る(①)．②において，実際の作業をする手続き map1 を内部手続きとして定義する．この手続きの定義にあたっては，引数として渡された手続きを，f という名前で使用し，実行時に引数 x に結びつけられる入力リストの各要素に次々と作用させることが記述されている．定義された手続きオブジェクトは，ワード map1 に結びつけられているので，③において，この map1 に結びつけられた本体を，手続き mapcar の返り値として出力する，という構造となっている．この新しく作り出された手続きは，所定の目的を実行する手続きである．

例として，リストに含まれる各要素を 2 乗する手続き squared-list は，この手続き mapcar と手続き square を用いて，

```
make "squared-list mapcar :square

to mapcar !f                              ...①
  to map1 !x
    if empty? :x [output [ ]]             ...②
    fput f first :x map1 butfirst :x
  end
  :map1                                   ...③
end
to square !x
  :x * :x
end
```

図 8 高階手続き定義の例 (1)
Fig. 8 Higher-order procedure (1).

と表現することにより定義することができる．手続き mapcar は手続き square の定義を使用し，目的の動作をする手続きを作り出し，make がそれを手続き名 squared-list に結びつける．このようにして作り出された手続き squared-list は，引数としてリストを要求し，

```
squared-list [1 2 3 4 5]
```

の形で実行させることにより，[1 4 9 16 25] の結果を出力する．

図 9 は，同一の母体から，リスト中の要素の和を求める手続きと，積を求める手続きを作り出す例を示している．高階手続き make-f は，入力として演算手続きとその単位元を取る．和を求める手続き sum は，手続き add と単位元 0 を用い，

```
make "sum make-f :add 0
```

とすることにより，また，積を求める手続き product は同様に手続き mul と単位元 1 を用い，

```
make "product make-f :mul 1
```

と作り出される．

図 10 は，数値的な微分により導関数を作り出す例である．高階手続き m-deriv は，手続き f と微分したい次数 n をもとに手続き deriv を用いて n 次の導関数を表す手続きを出力する．手続き deriv は，入力手続き f と微小区間 dx (この例では 0.00001) をもとに，微分の定義の実行を行うオブジェクトを作り出す．m-deriv に手続き sin と次数 1 をあたえ，

```
make "my-cos m-deriv :sin 1
```

とすると，cos 関数の値を計算する手続き my-cos が得られる．

上の例に見たように，引数として手続きオブジェクトを受け取り，実行結果として目的の動作をする手続きオブジェクトを返す手続きは，このような方法で容易に定義することができる．この高階手続きは，Logo のプログラミングに新しいスタイルをもたらすと期待される．

```
to make-f !f !e-unit
  to calculate !list
    if empty? :list [output :e-unit]
    f first :list calculate butfirst :list
  end
  :calculate
end
to add !x !y
  :x + :y
end
to mul !x !y
  :x * :y
end
```

図 9 高階手続き定義の例 (2)
Fig. 9 Higher-order procedure (2).

```
to m-deriv !f !n
  if :n=0 [output :f]
  deriv m-deriv :f :n - 1 0.00001
end
to deriv !f !dx
  lambda [x] [((f :x + :dx) - (f :x)) / :dx]
end
```

図 10 高階手続き定義の例 (3)
Fig. 10 Higher-order procedure (3).

7. Logo の処理系

前章までに述べた機能は，実際の処理系を作成することにより，その上で実現している．**図 11** は，この処理系の概念的な基本構造を表している．

Logo 言語の特徴の一つに汎用手続き (generic procedure) の機能が挙げられる．汎用手続きは，与えられた引数に対して，データ・タイプの異なりから生ずる取り扱いの区別を意識することなく，その手続きを適用し，意味的に近い概念の処理を行うものである．例えば手続き first は，文字列 'Logo' に対しては，先頭の文字 L を返し，また，リスト [Logo is a powerful language] に対しては，リストの先頭のワード Logo を返す．表層 Logo は，本稿の Logo の言語仕様を作り出している部分であり，この汎用手続きの概念を含んでいる．表層 Logo はそのまま直に実現されているのではなく，核 Logo で記述することに

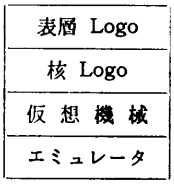


図 11 処理系の基本構造
Fig. 11 Basic structure of the system.

```
to first !x
  if string? :x [output string-first :x]
  list-first :x
end
to fput !x !y
  if string? :y [output string-fput :x :y]
  list-fput :x :y
end
```

図 12 汎用手続きの実現例
Fig. 12 Generic procedure.

よりその動作として実現されている．**図 12** は，表層 Logo の仕様に含まれる手続き first および fput を核 Logo により記述した例である．この例では両手続きとも，引数として与えられたデータの種別を判断し，それぞれ文字列用手続きもしくはリスト用手続きを作用させている．

核 Logo は仮想機械の命令列で書かれ，インタプリタとして動作する．仮想機械はリスト処理に適したスタックを基本構造とするアーキテクチャを持った機械である．この機械の動作は，それを支えるエミュレータにより，実際のハードウェアで実行される．

8. おわりに

以上，手続き定義を主にプログラム記述能力の向上を目指した Logo につき述べた．シンタクティック・シュガーによるリーダ，評価器の機能分担は，評価器の構成をより容易にしえた．また，内部手続き定義，高階手続き定義の能力は，Logo プログラミングに新しいスタイルを導入することができ，特に高階プログラミングは，Logo の新しい適用を期待させるものであった．その他，詳細には述べなかったが，関数性を推し進めたこと，静的スコープを採用したこと，値の取り出しと区別するために，引数の表記法を定めたこと等も，本稿の Logo を特徴づけている．

処理系の開発にはワークステーション (Sun 4/280) を使用したが，プログラミングの段階においては，可能な限り移植性を考慮した．このため，処理系内のリスト領域を調整することで，パーソナル・コンピュータ上にも移植することができ，Logo 利用環境の多面化をはかりえた．

現時点での問題点としては，文の構造的解釈が実行時まで待たされることに起因している実行速度の遅さが挙げられる．このことは，Logo の言語仕様として，文に明白な構造を示す括弧等のマーカが不要である，ということの裏返しで生ずるものである．しかし，インクリメンタル・コンパイリングを行うことを考えたとき等には，ぜひ乗り越えたい問題の一つである．

今後は，この言語の持つ構文上の簡素さを，より一層有用とさせるべく，引続き考察を重ねていきたい．

最後に，処理系の作成については，本学研究生，河村政雄氏に負うところ大である．ここに記し謝意を表する．

参考文献

1) Abelson, H. and diSessa, A.: *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*, MIT Press (1981).
2) Abelson, H. et al.: *Structure and Interpretation of Computer Programs*, MIT Press (1985).
3) Abelson, H. et al.: Intelligence in Scientific Computing, *Comm. ACM*, Vol. 32, No. 5, pp. 546-562 (1989).
4) Allen, J.: *Anatomy of Lisp*, McGraw-Hill (1978).
5) Bird, R. S.: Algebraic Identities for Program Calculation, *Comput. J.*, Vol. 32, No. 2, pp. 122-126 (1989).
6) Bundy, A. (ed.): *Artificial Intelligence: an Introductory Course*, Edinburgh University Press (1980).
7) Church, A.: *The Calculi of Lambda-Conversion*, Princeton University Press (1941).
8) Clinger, W. (ed.): The Revised Revised Report on Scheme or An UnCommon Lisp, MIT (1985).
9) Henderson, P.: *Functional Programming: Application and Implementation*, Prentice-Hall (1980).
10) Howe, J. A. M. et al.: Model Building, Mathematics and Logo, Yazdani, M. (ed.), *New Horizons in Educational Computing*, pp. 54-71, Ellis Horwood (1987)
11) Noss, R.: *Creating a Mathematical Environment through Programming: A Study of Young Children Learning Logo*, University of London Institute of Education (1985).
12) Papert, S.: *Mindstorms: Children, Computers, and Powerful Ideas*, Basic Books (1980).
13) Steele Jr., G. L.: *Common LISP: The Language*, Digital Press (1984).
14) 山本ほか：関数型プログラミング・パラダイムによる Logo 言語の拡張，昭和 63 年度人工知能学会全国大会，5-10 (1988).
15) 山本ほか：抽象データ型を拡張した Logo における高階手続きの記述，教育工学関連学協会連合全国大会，1 D 01 (1988).

(平成元年 6 月 28 日受付)
(平成 2 年 1 月 16 日採録)

山本 順人（正会員）

昭和24年生．昭和46年大阪大学基礎工学部電気工学科卒業．昭和51年同大学院基礎工学研究科博士課程修了．工学博士．同年国立民族学博物館第5研究部助手．民族音楽の情報処理，特にその計量方法に関する研究に従事．昭和57年筑波大学電子情報工学系転任，現在にいたる．関数型プログラミング，大型電子計算機システム等に興味を持っている．電子情報通信学会，人工知能学会各会員．

中山 和彦

昭和9年生．昭和32年国際基督教大学教養学部卒業．昭和39年同大学院教育研究科修了．教育学修士．同大学助教授，文部省学術調査官を経て，現在，筑波大学電子情報工学系教授．コンピュータの教育利用の研究に10年以上にわたり従事．また，大量データベースに関する研究にも興味を持っている．著書「未来の教室」，「コンピュータ支援の教育システム—CAI」など．日本科学教育学会会員．