Vol. 39 No. SIG 1 (PRO 1) 情報処理学会論文誌 Dec. 1998

Visibility に従ったプログラミング言語

原田康徳† 宮本健司†

新しいプログラミングパラダイム Visibility Programming (VP) を提案する．VP に従ったシステムは「機能の開放性」(機能拡張が自由に行なえる) を持つ．VP によるシステムは 2 つの部分，1) システムの全ての状態を含む共有データ空間，2) その共有データ空間を操作するプロセスの集合，から構成される．本稿は，新しいスタイルの対話型言語として，VP に従ったプログラミング言語 moji-lisp と moji-forth を紹介する．これらは，エディタ上で動作する言語で，テキストをヒープとして使用するものである．

Programming Languages based on Visibility

HARADA YASUNORI† and MIYAMOTO KENJI†

We propose a new programming paradigm, Visibility Programming(VP), to construct user-extendable systems. A system based on VP consists of two parts: 1) a shared data space that includes all information of the system, and 2) functions that manipulate the shared data space. In this paper, we describe two VP-based programming languages: moji-lisp and moji-forth.

1. はじめに

GUI の普及などで，コンピュータはより対話的に使用されるようになってきた．コンピュータを人間の作業を支援する道具として考えるときに，どのようなプログラミング言語が必要なのであろうか．それは，単に対話的なシステムを記述するためだけでなく，もっと人間の作業そのものに深く関係するようなものであるべきだろう．

Lisp, Prolog, Logo のような対話型言語は，コマンドラインからインタプリタを呼び出して，その実行結果を得るようなものである．しかし，このようなコマンド形式は，ユーザインタフェースとしては古いタイプのものである．現在の GUI によるアプリケーションに見られるような，イベント駆動をベースにした対話的なシステムにマッチするような，新しいスタイルの対話型言語をデザインする必要がある．

GUI を支えるプログラミング技術として，オブジェクト指向プログラミングは重要である．アプリケーションはオブジェクトとして定義され，ユーザの動作に反応して計算を行う「魔法の紙」として作られる．

しかし，この「魔法の紙」方式の欠点は機能の開放性がないことである．機能の開放性とは，機能の拡張が自由にできること，拡張された機能は他の機能に影響を与えないこと，機能の組合せにより新しい機能を作ることができること，などの性質である．紙が機能を持っている「魔法の紙」では，新しい機能を紙に追加することは (プラグイン技術は後で述べる) 困難であり，機能ごとに複数の紙を用意しなければならない．そして，紙の間を copy/paste のプロトコルによってデータを移動させることで，複数の機能を適用させるのである．

それを解消する新しい対話的なシステムの新しい考え方として我々は Scott Kim によって提案された，Visibility[1] に注目している．Visibility を持つシステムとはコンピュータの全ての状態は画面で表現されており，ユーザの操作と画面とで，次の画面が一意に定まるようなシステムである．このシステムには，ユーザに隠された状態が存在しない，ユーザとコンピュータとの間に大きなギャップを感じない，などの特徴がある．

この Visibility を単なるユーザのコンピュータとの対話のモデルとして捉えるのではなく，コンピュータ内の小さな機能間の通信のモデルとして拡張する．これを Visibility Programming(VP) と呼ぶことにする．

我々は VP に従ったプログラミング言語を開発した．テキストを対象とした moji-lisp/forth[7] とベクトルグラフィックを対象とした VISPATCH[9),10] である．これらの開発の動機は，VP の性質を明らかにし，より柔

† NTT 基礎研究所
NTT Basic Research Laboratories

軟な対話型システムの構築技術として高めたいからである．

moji-lispとmoji-forthは，プログラムも含む全ての情報がテキストで表現されたプログラミング言語である．これらの言語で用いる関数は，引数と戻り値としてテキスト上の位置をとり，テキストに副作用を残す．これはちょうど，Lispでの関数で，引数と戻り値としてセル領域へのポインタをとり，セル領域に副作用を残す，と解釈できること同じである．moji-lispは構文がLispに似ているため，アルゴリズムなどを記述しやすいという利点がある．一方，moji-forthは逆ポーランド記法でプログラムを記述するが，キーボードマクロに近い構文のため，連続した動作の列を記述するのに適している．

VISPATCHはユーザのイベントにより図形の書き換えを起動する言語で，書き換えルールも図形で表現されていることから，自己拡張可能なプログラムが実現できる．

すでに，VISPATCHはいくつかの報告があるので，本稿では，moji-lisp/forthの説明に重点を置く．次の節ではVPが解決する問題について述べる．3節では，moji-lisp/forthのデザインを説明し，残りの節で考察，まとめをする．

2. Visibility Programming

機能の開放性を持っているシステムには，小さな機能の集合と，それらが操作する共通のデータ空間とで構成されているものがある．たとえば，Emacsによるアプリケーションは操作対象としてのテキストバッファと，それらを操作する機能としてのe-lispの関数群から構成されている．バッファ上の文字列サーチの機能(関数)は，すべてのバッファ上で動作する．電子メールやWWWのブラウザのようなアプリケーションもバッファを介して作られているので，文字列サーチの機能はここでも利用できる．ユーザがバッファ上で動作する機能(たとえば，英単語の辞書検索など)を新しく作製したとしても，やはり全てのアプリケーションで利用できるようになる．それ以外にも，UNIXのファイルシステムの名前空間や，UNIX上のテキストによるデータベース(`/etc/passwd` など)とフィルタコマンドの組合せなど，小さな機能の集合と共通のデータ空間によるシステムは，機能に対して開放性を持っている．

このような小さな機能の集合と共通のデータ空間によるプログラミングをVPと呼ぶ．アプリケーションは複数の小さな機能を持ったプロセスを組み合わせることで作られる．データ空間にはプロセスの内部状態やプログラム自身も表現されているところが，単純な黒板モデルとは異なっている．特に対話型システムに限定すると，このデータ空間はユーザから自由に見えて，編集できることが必要である．

一方，オブジェクト指向プログラミングによるアプリケーションでは，データとそれを操作する機能を，一つのオブジェクトとして隠蔽している．オブジェクトの内部のデータはそれぞれのオブジェクトが用意しているメソッドによって行われる．一度オブジェクトとして作られてしまうと，それに新しい機能を追加することは難しい．新しい機能は，オブジェクトが公開していないデータを参照する必要があり，また，それぞれのオブジェクトで内部構造が異なっているからである．様々なユーザの要求に答えるようにアプリケーションが拡張されて来た結果，一つのオブジェクトが沢山の機能を持った巨大なものとなってしまうのである．

プラグイン技術は，あらかじめ統一されたインタフェースによって，アプリケーションの機能を拡張させるための技術である．しかし，プラグインのプログラムを記述するためには，必要なアプリケーション内部のデータ構造にアクセスできる必要がある．また，プログインのプログラムが様々なアプリケーションで再利用できるためには，各アプリケーションでのデータ構造が統一されている必要がある．これはまさにVPそのものである．

このようにVPは機能の開放性を持つシステムの枠組として重要である．

3. moji-lisp/forth

moji-lispとmoji-forthはテキストを対象としたVPのためのプログラミング言語で，動作環境としてEmacsを利用している．moji-lispはLispに似た構文，moji-forthはFORTHに似た構文でプログラムを記述する．構文と若干の計算機構の違い以外は，moji-lispとmoji-forthは同じ性質をもっている．ここでは，最初に両者に共通している性質を述べ，次にそれぞれの言語に固有の性質を述べる．

3.1 計算対象

moji-lisp/forthはプログラムと計算対象のデータはすべてテキストで表現され，計算もテキスト上で行われる．関数はテキスト上の位置を入出力して，テキストに副作用を残すが，それによって，テキスト上に書かれたデータを計算しているように見せている．

ユーザも同様に任意のテキストの任意の位置に書かれた文字を見て編集することができる．このことは，これらの言語にはデータの入出力に関する機能は必要ない

ことを意味する．ユーザが編集したデータで計算を行ない，その結果はユーザが指定した位置に書き込まれる．このユーザとの対話は表計算ソフトのそれと似ているものである．

3.2 データアクセス

基本的なデータをアクセスするには，テキストを簡単に構文解析する必要がある．たとえば，リストに相当するデータは

```
(a (1 2 3) b c)
```

という文字列の先頭の位置によって表現される．しかし，car，cdr のようなメモリ構造があるわけではないので，括弧や空白などの文字を調べながらデータの中身をアクセスする必要がある．たとえば moji-lisp の構文を用いて説明すると，リストの先頭を X が指しているとし，その4つの要素 a，(1 2 3)，b，c を順にアクセスするには，

```
(setq item (forward-char X))
(while <itemが文字 ")" を指していない>
  ...
  (setq item (forward-sexp item)))
```

のようになる．ここで forward-char は一文字分先(右)へ進む，forward-sexp は(構文を解析して)一つの Lisp の式だけ先(右)へ進む，という Emacs の関数に対応している．

一方，Lisp での関数 cons に相当する機能は，式の挿入によって行われる．たとえば，X がリストを指しており，それに Lisp での (cons 'a X) に相当した動作をさせる場合は，

```
(insert (forward-char X) 'a) X
```

となる．(insert P D) は位置 D から参照している一つの式を，位置 P に複製して挿入する，という働きをする．cons との違いは，元のデータが破壊されしまうことである．cons が一つのセルの割り当てと，ポインタの操作という，極めて単純な操作しか行わないのに対して，moji-lisp の例では，複製や挿入という重い操作を行っている．

3.3 メモリモデル

テキストの位置の実装は，テキストの挿入 / 削除といった操作に付随して移動させるために，Emacs の marker オブジェクトにより表現されている．marker オブジェクトは，挿入 / 削除などの操作でも，テキスト上の文字と一緒に移動するという性質を持っている．

これらの例で理解されるように，moji-lisp/forth では部分的に共有されたデータを表現できない．たとえば，Lisp の append が返すリストの後ろの部分は，第2引数のリストを共有している．このようなことは，moij-lisp/forth では不可能である．その代わり moji-lisp/forth ではデータを直接コピーしたり，書き換えたりする．一方で，テキストは挿入 / 削除といった高度な操作が可能である．この問題はテキストという表現形式の持つ性質をそのままひきずっている．表現形式に関する議論は後で述べる．

moji-lisp/forth では，fixnum のようなポインタに埋め込まれたという軽いデータ構造は持っていない．小さな数値データであっても，テキストを用いて表現される．それは，実装上の効率よりも，データのインタフェースの統一を重視したからである．

そのために，数値計算でも戻り値を格納する場所を意識する必要がある．例えば，add という関数は2つの位置を引数にとり，一方にある数値データを，もう一方の数値データに加え(破壊的に書き換える)，その位置を返すという動作をする．

```
(add 2 1)
```

というプログラムは，2に1を加えることになるが，2が置かれている位置(プログラムの中)にその答を格納するので，このプログラムが

```
(add 3 1)
```

のように書き換えられる．

このような副作用を持つ関数は使いにくいので，答を格納する領域を適当に割り当てて，そこへの位置を返すような関数も用意されている．

```
(+ 2 1)
```

これは，新たに答を格納する領域(*moji-temp* というバッファ内)を割り当て，そこに答3を書き込み，そこへの場所を返す関数である．この方式では，答が離れたバッファ内に書き込まれるために，答がどこに返されたはユーザに分かりにくい．まとまりの計算が終了した段階で，ユーザが指定した領域に，結果をコピーして見せる方がよい．

moji-temp に書かれたデータの大半はゴミになる．しかし，moji-lisp/forth ではゴミ集めは行わない．なぜなら，全てのプログラムから参照されていないデータでも，ユーザが必要としているという可能性があるからである．そこで，*moji-temp* に書かれたデータはユーザの責任で消去する必要がある．計算が進むと，*moji-temp* バッファは大きくなるが，プログラムの作法上ここには中間状態しか存在しないので，適当なタイミングで，この領域を消去してもかまわない．

3.4 moji-lisp と moji-forth の関係

moji-lisp と moji-forth の関数は同じ名前空間を持ち，互いに呼び合うことができる．引数の順序は，スタックトップが最後の引数に対応し，つまり引数の順に

スタックに積まれている．たとえば，(add A B) とA B add は同じ関数を同じ引数の順序で呼び出している．

moji-lispでは引数の個数が固定であり，値の個数が1と決められている．その反面，プログラム中で引数を変数として受けることができるので，複雑な引数の呼びだしでも分かりやすいプログラムを記述することができる．

一方，moji-forthでは入出力の値の個数は動的に変化させることができる反面，引数の個数が多い場合，プログラムはスタックの入れ換えや複製などの本質的ではない操作が多くなってしまい，プログラムは書きにくくなってしまう．

経験的にはスタックの深さが3を超えるようなプログラムはmoji-lispを，パイプラインのようにスタックを浅くしか使わないようなプログラムはmoji-forthで記述するのがよい．

3.5 関数定義

Lispでは，関数の定義はdefunという命令を実行することで行われる．この実行によって，関数名とその関数の実体とを関係付けるようなメモリ構造を作り出すのである．このメモリ構造は処理系によって異なっているが，もしその構造を作り出すことができるのなら，defunという命令を用いなくても，関数の定義ができる．

moji-lisp/forthでは，命令によってではなく，このようなメモリ構造を直接作り出すことで，関数の定義を行う．具体的には，

```
inc: (lambda (x) (add x 1))
dec: [ 1 sub ]
```

という記述(構文は行の先頭から名前，コロン':'，0個以上の空白，開き括弧)があると，それらは関数の定義と見なされる．開き括弧が'(' の場合moji-lisp，'[' の場合moji-forthの関数である．

このようにメモリ構造がはっきりと見えている利点としては，関数定義を操作するプログラム(たとえば，wrapperを関数の定義にかぶせるなど)を簡単に作りやすい，ことがあげられる．

3.6 大域変数

moji-lisp/forthでは大域変数が用意されている．大域変数はテキスト上で，

```
X: = 10
Y: = (1 2 3 4 5)
```

のように表現することで，作り出すことができる．ここで変数X の値は10，Yの値は(1 2 3 4 5) である．たとえば，moji-lispで

```
(setq X Y) (setq Y 0)
```

を評価すると，大域変数が

```
X: = (1 2 3 4 5)
Y: = 0
```

のように書き換えられる．さらに，

```
(add Y 1)
```

によって，

```
X: = (1 2 3 4 5)
Y: = 1
```

のように書き換えられる．

3.7 名前の探索

moji-lisp/forthでは関数と大域変数は同じ空間に割り当てられ，空間はテキストバッファの列により定義される．組み込み関数やmoji-lispの局所変数でない名前(仮にNameとする)が現れると，その名前を，1) キー入力が行なわれた現在のバッファ，2) 変数func-buffersに登録されているバッファ列，3) 環境設定ファイル"conf.mj"，を順に検索する．一つのバッファ内では，バッファの先頭から，"\nName:"というパターンを検索する．そのパターンが見つかると，コロン(:)の次の空白でない文字Cによって，その名前の定義の型が決められる．Cが丸括弧"("のときはmoji-lispの関数，角括弧"["のときはmoji-forthの関数，等号"="のときは大域変数である．

func-buffers 自身は環境設定ファイルに，

```
func-buffers:= ("func.mj" "primitive.mj")
```

のように定義されている．この定義を(setq ..)などで書き換えることで，簡単に関数や大域変数の探索を変更することができる．

3.8 キーマップと実行環境

moji-lisp/forth ではキーマップ(キー列とそれが入力されたときに起動される関数との対応表)の定義をキー列の名前を持つ関数を明示的に定義をすることで行なっている．たとえば，^F は関数名であり，それはmoji-lisp/forthの環境でコントロールFを押したときに起動される．これにより，キーマップは上で述べたようなバッファ列に従った階層を持つことになる．

Emacsではポイント(一般にはカーソルと呼ばれる)とマークという2つの位置を用いて編集用の関数を呼び出すことで，編集作業が行われる．それを拡張してmoji-lisp/forthでは，位置(バッファ名+バッファ内の位置)の大域的なスタックを用いて関数を呼び出す．このスタックトップはEmacsでのポイントに，それ以降に積まれた位置はマークに相当する．たとえば，

```
^F: (lambda (x) (forward-char x))
```

というmoji-lispの関数は^Fを押すことで起動され，スタックトップ(すなわちポイント)が引数として渡され，

その位置を一文字先に移動させた位置を返し，スタックに積む．この結果，ポイントは一文字先に移動する．

3.9 moji-lisp

moji-lisp では特殊形式 setq，let，cond，quote などが用意されている．lambda や let によって，局所変数を定義することができる．それ以外の機能は，moji-lisp/forth 共通の組み込み関数として提供されている．

3.10 moji-forth

moji-forth はプログラムの構文を forth のように逆ポーランド記法で記述する．moji-forth の最大の特徴は，パイプラインのような1入力1出力の関数を順に呼び出す場合，極めてシンプルに表現できることである．たとえば，

```
^O: [ ^F ^F ^P ]
```

のような moji-forth のプログラムは，ポイントを順に移動させるキーボードマクロのように見える．

moji-forth では全ての語に実行規則が定義されている．語の先頭の文字によって，

(1) 数字：先頭の位置をスタックに積む．
(2) '(', '[', '"'：先頭の位置をスタックに積む．
(3) それ以外：その定義を探して実行する．

と，なっている．

moji-forth は，構造を直接扱うことができるため，整数しか扱えない FORTH よりも強力になっている．制御構造は，ブロックを用いることができる．

```
a10: [ 10 [ "a" insstr ] repeat ]
```

この関数 repeat は，繰り返し回数10とブロック ["a" insstr] が渡され，ブロックの中身を10回繰り返して実行する．その結果，文字 a が10個挿入されることになる．

次の例は，ポイントの下にある文字に応じた処理を行う．

```
^C: [ dup "[0-9]" looking-at
        [ "N" insstr ] [ "-" insstr ] if ]
```

ここで，A B looking-at は B で示された文字列を正規表現として解釈し，それが A の位置でマッチするかを調べる関数である．また，C T E if は C の条件が真なら T を偽なら E をそれぞれ実行するものである．これによって，ポイントの下に 0 から 9 までの文字がある場合文字 N を挿入し，それ以外は文字 - を挿入する．なお，偽を表す位置は特別なバッファの原点を使用している．

3.11 プログラム例

以下に moji-lisp/forth によるプログラム例として，8-Queen パズルを述べる．VP の特徴がよく現れるように，ここでのプログラムは，パズルの解を全て求めるのではなく，人間がパズルを主体的に解き，その手助けをコンピュータがする，というものである．このような問題設定は，解き方が分からない(プログラム化できない)ような難しいパズルを解く，パズルを解くことを楽しむ，人間が美しいと感ずる解を求める，などの場合に有用である．

図1は関数の定義とそれを実行させた例を示している．関数 Board は moji-forth で記述されており，チェスのゲーム盤(罫線で囲まれた 8×8 の空白)を生成する．次の関数 Put はゲーム盤上の任意の位置を引数 x として呼び出され，x に空白が書かれているなら，x に文字 Q を重ね書き (replstr) し，x から Queen が進む8つの方向に関数 Move を呼び出す．ここで，^F, ^B, ^P, ^N は，それぞれ右，左，上，下方向への1文字移動の関数であり，これらを組み合わせて，方向を moji-forth の構文として作り出している．関数 Move は位置 x と方向 dir を与えられ，(setq x (exec x dir)) によって x を dir の方向に一つ進めて，x が空白か点(他の Queen が移動可能の場所を示す)を指しているある間，x を dir の方向へ一つ進めて，点を置いて行くというものである．ここで，exec は，moji-lisp から moji-forth の構文を呼び出すもので，最後の引数以外をスタックに積み，最後の引数を moji-forth の構文として実行し，スタックから一つ pop した位置を全体の値とする．つづく2行は局所的なキーバインド ^O, ^P を定義している．

これらの定義の下に実行例(1つの Queen を置いた状態)が示されている．新たに Queen を置くには，置きたい位置にポイントを移動し ^Q を入力する．それによって，ポイントに Q が置かれ，その駒が利いている(もう他の Queen を置くことができない)全ての場所に点が置かれる．これを繰り返して，Q が8個に満たないのに，空白が無くなってしまうと，失敗である．

エディタ上で人間がパズルを解くことには，次のような利点がある．1) 盤の状態を文書の複製機能を用いることで，自由に保存することができる．2) 複数の盤を同時に扱える．3) 前の状態に戻るために，エディタの UNDO 機能を用いることができる．4) コメント，解説文などを混在できる．これは，機能の開放性のよい例となっている．

4. 考察

4.1 moji-lisp/forth の限界

moji-lisp/forth はテキストを表現形式としている言

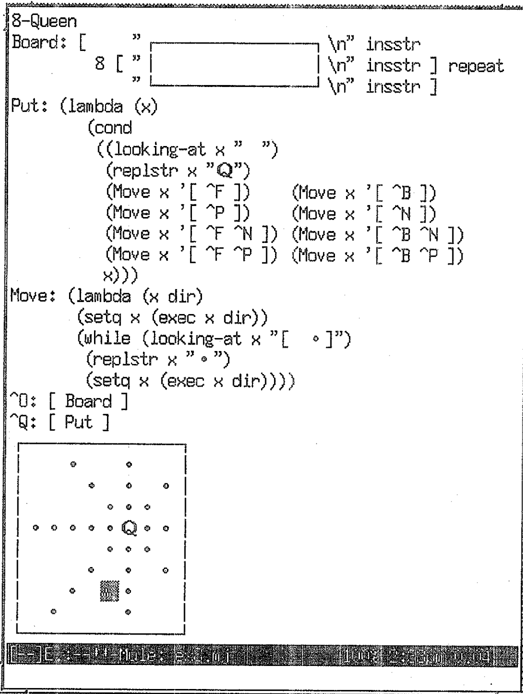


図1 8-Queen
Fig. 1 8-Queen

語であるので，テキストの表現能力の影響を受けてしまう．たとえば，lispでは関数appendを呼び出すと第2引数のリストを共有するようなリストを返す．しかし，テキストではこのようなメモリの共有構造を直接実現できない．

テキストで共有構造を表現するためには，名前(ラベル)を導入して，それらの同一性を用いる間接的な方法がある．たとえば，

```
(A B C D #123
(1 2 #123
#123: E F G)
```

のようにして，共有している2つのリスト(A B C D E F G)，(1 2 E F G)を表現するのである．この場合，forward-char のようなアクセス関数もこの共有構造を理解して動作しなければならない．たとえば1行目で forward-char を何度も呼び出しているうちに，#123まで進むと3行目のEの手前に飛ぶのである．しかし，逆に3行目の後ろの方から backward-char を呼び出した場合，1行目と2行目のどちらに飛べば良いのか決定できない．

このように，共有構造を名前を用いて表現することは可能であるが，1) ユーザにとっての可読性が悪くなる，2) 多くの関数が共有構造を理解して動作するように作られていなければならない，3) 上述のようにテキストの空間がゆがむことで，操作の統一性がとれなくなる，という欠点がある．そのため，moji-lisp/forthのデザインではデータレベルの共有構造を無視した．

4.2 他のVisibility言語(環境)との比較

萩谷らは「編集＝計算」パラダイムを提唱しており[2)]，我々のVisibility Programmingと極めて深い関係にある．その一つ，Boonborg-KEISANはテキスト上のスプレッドシートと呼べるような環境で，単方向制約を解消することで，計算を行う．また，強力なPBE機構によって，繰り返しなどの能力も導入している．

我々のVisibility Programmingの特徴は，コンピュータ側の機能を細かな関数として分離させて考えることにある．人間とコンピュータとのインタラクションだけでなく，関数相互のインタラクションにも注目したことである．それによって，システムに機能の開放性が得られるのである．

Boonborg-KEISANの特徴は記号(文字)に意味を与え，その記号を通じてコンピュータとインタラクションをしている．また，KimのVIEWPOINTは画面の位置に機能を割り当て，位置を通じてコンピュータとインタラクションしている．これらに，新しい機能を導入するためには，新しい記号や位置を用意する必要がある．それに対してmoji-lisp/forthでは，機能の割り当てを特定の画面の位置から切り放し，間接的に関数の定義によって，キーイベントと機能とを対応つけた．このため，新しい機能の導入は容易になった．

書き換えをベースにしたVisibility言語も数多く存在する．ビットマップを対象としたBITPICT[4)]，Visulan[3)]，アイコンを対象としてCocoa[8)]，ベクトルグラフィックを対象としてChemTrains[5)]，VISPATCHなどである．これらに共通した課題は，純粋な書き換えだけでプログラミング言語として強力な記述力を与えることができるかである．他のパラダイムを導入すれば，強力な記述力が得られるのは当然である．

また，テキスト以外を対象としたVisibility言語の現状の欠点は，編集能力がテキストに比べて極めて低いことである．これは，よいエディタを開発すれば，解決できると思われる．

4.3 非データ抽象

VPとオブジェクト指向との比較は重要である．オブジェクト指向プログラミングは，データ抽象，多相，継承などの要素技術に分解できる．このなかで，VPと対立概念にあるのは，データ抽象である．それ以外の多相，継承はVPにも導入可能である．

データ抽象は異なったレベルのデータを共通の空間にマップする際に必要な技術である．一方，VPは共通に

なった空間上でのプログラミング技術である．例えば，UNIXのファイルシステムは，デバイス，ファイル，ソケットなどの異なったリソースを一つの名前空間にマップしている．このマッピングにはデータ抽象が用いられる．一度，様々なリソースが共通の空間にマップされると，そこでVPが可能になり，それらのリソースを操作する機能に拡張性が生まれる．

データ抽象が重要であったのは，データ構造が実行効率に与える影響が大きかった場合である．もちろん現在でも高効率なプログラミングが求められている領域は多いが，逆に対話システムのようにユーザの反応速度が一定である応用では，ハードウェアの高速化による恩恵はより柔軟なプログラミングに与えられるべきである．そのような意味で，効率が悪いけれども統一されたデータ構造を用いることは(たとえば数値までも文字列で表現した)意味のあることである．

データ抽象をせずに多相と継承を行なう方法はすでに開発されている[6)]．クラスに条件式に対応させることで，任意の構造のデータにクラス階層を割り当てることができる．クラスには，メソッドを定義することができる．実装は，ある名前のメソッドに対して，一つの関数を割り当て，そのメソッドを持つクラスの条件式により分岐しながら，クラスを判定し，メソッドの本体を呼び出す．

この技術は，moji-lisp/forthには使われていないが，一般のVisibilityに従ったプログラミング言語に適用可能である．この場合，条件はパターンマッチングのように，データがある状態にあるかどうかを判定することである．そのパターンが存在すると，それがオブジェクトのように振舞うことができるのである．

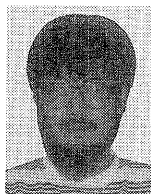
5. まとめ

Visibilityに従ったプログラミング言語moji-lisp/forthをデザインし，それによってVisibilityに関する理解を深めることができた．今後は，アルゴリズムアニメーション，インタラクティブリフレクションといったVPの別の側面も探って行きたい．

参考文献

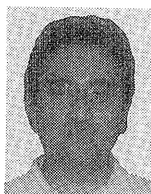
1) Kim, S.: *Viewpoint: Toward a Computer for Visual Thinkers*, Stanford University, 1988.
2) 萩谷昌己, 白取樹知:「計算＝編集」パラダイムに基づく例によるプログラミング, コンピュータ・ソフトウェア, Vol.13, No.3, 1996, pp.64-78.
3) 山本格也: ビットマップに基づくプログラミング言語 Visulan, WISS'95.
4) Furnas, G. W.: *New Graphical Reasoning Models for Understanding Graphical Interfaces*, CHI'95, 1995.
5) Bell, B. and Lewis, C.: *ChemTrains: A Languages for Creating Behaving Pictures*, IEEE Symposium on Visual Languages, 1993.
6) 原田, 山崎：条件クラス:データ抽象なしの多相と継承，プログラミング研究会, 1996/1.
7) 原田，宮本: 見えて触れるプログラミング言語, プログラミングシンポジウム， 1997.
8) Cypher, Allen and Smith, David C.: *KidSim:End User Programming of Simulations*, In Proceedings of CHI 1995, ACM, New York, 1995. http://cocoa.apple.com/cocoa/home.html
9) Harada, Y., Miyamoto, K. and Onai, R.: *VISPATCH: Graphical rule-based language controlled by user event*, Proceedings 1997 IEEE Symposium on Visual Languages, pp.162-163, 1997.
10) 原田康徳, 宮本健司, 尾内理紀夫: Visible Dispatch: 視覚的イベント駆動型アプリケーション構築法, コンピュータソフトウェア, Vol.15, No.4, 1998 (掲載予定).
11) 原田康徳, 宮本健司: 見えて触れるプログラミング言語, プログラミングシンポジウム， 1997.

(平成10年6月4日受付)
(平成10年9月14日採録)



原田 康徳（正会員）

1963年生．1987年北海道大学工学部応用物理学科卒，1992年同大学大学院工学研究科情報工学専攻博士課程修了．博士(工学)．1992年より，日本電信電話株式会社基礎研究所勤務，現在に至る．プログラミング言語，ユーザインタフェース，開放システムなどに興味を持つ．日本ソフトウェア科学会，ACM各会員



宮本 健司

1963年生．1986年東京大学教養学部基礎科学科第一卒，1991年同大学大学院理学系研究科相関理化学専攻博士課程修了．理学博士．同年NTT(株)入社．現在，同社基礎研究所勤務．プログラム自動合成，帰納推論，型理論の応用などに興味を持ちつつユーザインタフェースの研究に従事．