操作の繰返しによるマクロの自動生成

増井 俊之
シャープ株式会社 技術本部
情報技術研究所
masui@shpcsl.sharp.co.jp

太和田 誠
長岡技術科学大学

テキストエディタ上のユーザの操作の繰返しから次の操作を予測してマクロとして定義し実行するシステムを作成 / 評価した。ユーザは自分が同じ操作を繰返していることに気付いた時点で「繰返し実行キー」を入力することにより、システムが以前のユーザの処理から繰返しパタンを抽出して実行する。ユーザは操作列のマクロ定義法やその実行法などの知識は必要なく、「繰返し実行キー」により以前の操作が繰り返されることのみ記憶しておけばよいためキーボードマクロのような従来の手法よりも簡便に使用することができる。

Automatic Creation of Keyboard Macro

Toshiyuki MASUI
Information Technology Research Laboratories
Corporate Research and Development Group
SHARP Corporation
masui@shpcsl.sharp.co.jp

Makoto TAWADA
Nagaoka University of Technology

We have developed and evaluated a text editing system where “keyboard macro” is defined automatically from repetitive user inputs. A keyboard macro is a command defined by the user to abbreviate a sequence of keys. In our system, the macro is defined automatically by the system by typing the “repeat key” after typing the same key sequences twice. The user can instruct the system to continue his repetitive task just by typing the repeat key, without knowing how to define keyboard macros.

1　予測、例示を用いたユーザインタフェースへ

ユーザの計算機操作の負担を減らすための手法のひとつとして、以前のユーザの操作から次の操作を予測する方法がある。システムが自動的に予測結果を実行したりユーザに提示したりする方式もあるし、ユーザが明示的に予測を指令する方式もある。近年はグラフィックインタフェースをユーザが操作するときの繰り返しパタンをシステムが自動的に検出して次の操作を予測するシステム [3] や、ユーザの示す例からの汎化により操作手順を示すプログラムを生成したりするシステム [2] [6] [7] [8] [10] の研究が盛んである [11] [13]。

一方、予測や例示の手法を用いてファイル編集やコマンド行入力のようなテキスト操作を簡単化する試みも以前から行われている。 Darragh らの Reactive Keyboard システム [5] では、テキストエディタ操作において以前のキーストロークの頻度から次の入力文字列を予測し、表にしてユーザに提示してユーザが選択可能とすることにより障害者やタイピングが苦手な人の補助に成功している。 Nix の Editing by Example システム [12] では、テキストファイルの多くの部分に同じような修正を行なうとき、修正前と修正後のテキストの例をユーザがシステムに示すことにより、システムがその変換規則を推論して残りのテキストに対しその変換を適用することができる。また Mo らのシステム [9] では簡単なテキストエディタ上でのユーザの繰返し操作から汎化によりプログラムを生成して以後の操作に適用可能とするが、その際間違いが発見されるとユーザが操作を訂正することによりインクリメンタルにプログラムを修正することができる。ヒューリスティクスにより一番もっともらしい操作が選択されるようになっているうえに修正により正しい条件判断ができるようになっているため、 Nix のシステムでは不可能であったような複雑な処理も例示のみを用いて指定可能となっている。

しかしこのように各種の研究が行なわれているにもかかわらず、予測を使用したインタフェースは実際にはほとんど使われていないのが現状である。

2　予測インタフェースの必要条件

予測インタフェースは絶対必要な機能ではなく、あれば便利といった付属的な機能である。ある手法の採用によりひとつの点で便利になっても他の点で不便になっては困るので、予測を利用したインタフェースに限らず付属的なインタフェースやシステムが有用と認識され普及するためにはいろいろ条件が必要である。予測インタフェースが普及するためには以下のような要件が満たされる必要があると考えられる。

1. 予測を行なわせるための手間が少ないこと

   予測しないより場合よりも手間が増えるのでは予測を使用する意味が無い。予測を実行させる手段を覚える手間も少ないことが望ましい。

2. 正しく予測される確率が高いこと

   間違った予測をすることが多いとかえって取消しや修正などの手間がふえてしまう。

3. 予測機能を利用しない場合の障害とならないこと

   予測機能の付加によって予測を使用しない場合の使い勝手が悪くなってはいけない。

4. 予測結果が勝手に行使されないこと

   予測実行の手間は少ない方がよいが、システムが予測した処理を勝手に実行するのは困ることが多い。例えば従来のワープロでは入力した文字列をどんどん自動的に漢字に変換していくものもあったが、このようなインタフェースはわずらわしいため近年はあまり使われていない。

5. ユーザの邪魔にならないこと

   予測の処理のためにシステムの通常の動作が遅くなってしまっては困るし、また予測がユーザにとってわずらわしいものであっては困る。予測結果をユーザに示すことが必要な場合はなるべくユーザの邪魔にならない方法を使う必要がある。

6. 予測結果の実行にユーザが不安を感じないこと

   システムの予測を実行させることにより何が起こるかがユーザにとって明らかであると感じられ、また仮に間違った予測が行なわれても簡単に回復できることが必要である。

これらの要件は基本的と思われるが、前述のシステムはいずれかの要件を満足しないものがほとんどである。

3　Dynamic Macro

前述の条件を満たしつつ予測をインタフェースに活用する手法を考案した。

テキストファイルを編集するとき、連続した複数の行の先頭にコメント記号を入れたり、文字列を別の文字列に修正したりする場合のように、同じ操作を何度も繰り返さなければならないことがよくある。このような作業を簡単化するため、ユーザの指示によりシステムが繰返しを検出し、繰返されるパタンを自動実行するように以下のような仕様を設定した。この仕様はユーザが「自分は同じ処理を繰り返しているな」と感じたとき「繰返し実行キー」を入力すると予想通りの結果が得られることを目指したものである。以下では繰返し実行キーを REPEAT で表現する。予測を用いたインタフェースのほとんどは複雑な推論機構を用いているが、今回のものは非常に単純な規則のみを使用している。

- システムは常にキー入力の履歴を記憶しており、ユーザが REPEAT を入力したときそれまでのキー操作列からシステムが繰り返しパタンを検出してそれを実行する。予測されたシーケンスはユーザに確認を求めることなくすぐ実行する。

- 繰り返しパタンはそれまでのキーシーケンスのみから判断する。
- 直前に 2 回続けて全く同じ操作列が実行されていたとき REPEAT が入力されるとその 1 回分を繰り返す。そのような繰返しパタンが複数存在するときは最長の繰返しパタンを選択する。

  例えば直前の操作が「“abc” の挿入」「“abc” の挿入」であったとき REPEAT が入力されると「“abc” の挿入」を実行する。
- 上の条件があてはまらず、かつ直前の操作列と全く同じ操作列が以前の操作列中に存在するとき、それらの間の処理を実行する。

  例えば直前の操作が「“abc” の検索」「“abc” を “def” に修正」「“abc” の検索」であったとき REPEAT が入力されると「“abc” を “def” に修正」を実行する。ここでもう一度 REPEAT が入力されると、「“abc” の検索」「“abc” を “def” に修正」を実行する。
- REPEAT が入力されない限り何も行なわれない

  Eager[3] や Reactive Keyboard[5] のように予測結果が自動的に表示されるシステムと異なり、繰返し検出機能の存在を知らない人は知らないままになってしまうが、少なくともユーザの邪魔をすることはない。
- 間違った予測を実行した場合は undo 機能 (処理を実行する前の状態に復帰する機能) で対処する。

上述の仕様は明らかに 2節で示した条件のうち 1, 3, 4, 5を満たしている。 (2, 6については 6節で議論する。) このような機能を**動的マクロ生成実行機能** (Dynamic Macro: 以降 **DM** と表記) と呼ぶことにする。「動的マクロ」と呼ぶ理由は、上述の仕様は「キーボードマクロ」を自動的に定義することと同等とみなすことができるからである。キーボードマクロとは一連のキー操作を別の簡単なキー操作で置き換えてしまうという機能である。 GNU Emacs のキーボードマクロではまずユーザが特別のキー操作 (通常 ^X ([1]) によりマクロ定義開始を指示し、その後のキー操作でマクロとして定義したい編集作業を行ない、最後に別のキー操作 (通常 ^X)) によりマクロ定義終了を指示する。これにより定義されたマクロはマクロ呼び出しを指令するキー操作 (通常 ^X e) により実行される。キーボードマクロは便利な機能であるが、登録の開始と終了の指示という作業が必要である。同じ操作が繰り返されるということが最初からわかっていれば良いが、実際に操作を行なって初めて同じ操作が繰り返されることにユーザが気づくことも多く、このとき改めてキーボードマクロを登録するのはわずらわしい。 **DM** の手法では操作の繰り返しはシステムが判断するため登録の開始と終了をユーザが指示する必要がなく、実際に操作の繰り返しが発生した後でその繰り返しを登録することができる。このような意味で、繰り返しの実行においては **DM** はキーボードマクロよりも使い勝手が良いことが期待できる。

[1] ^X は Control-X キーを示す。

4 実装

DM は GNU Emacs 上に Emacs LISP を用いて実装した。制御キーのうち比較的使用頻度が少ないと思われる ^T キーを REPEAT として使用した。仕様が単純なので Emacs LISP で約 50 行の小さなプログラムである。

5 実例

5.1 単純な繰返し操作

まず **DM** で単純な処理の繰返しが簡単に指示できることを示す。図 1のような TEX のテキストの行頭にコメント記号 “%␣” を加える処理を考えてみる。

```
This is a sample \TeX text file.
The area from this line to the end
of this paragraph should be commented
out using the comment character '%.'
```

図 1: テキスト 1 初期状態

この状態で % ␣ ^N ^A % ␣ ^N ^A を入力するとテキストは図 2のように変化する。 (␣は空白記号、^N は次の行への移動、 ^A は行頭への移動を示す。)

```
% This is a sample \TeX text file.
% The area from this line to the end
of this paragraph should be commented
out using the comment character '%.'
```

図 2: 行頭にコメント記号を挿入

ここで、ユーザが同じ操作を繰り返していることに気付いて REPEAT (= ^T) キーを入力したとする。するとシステムはそれまでのキーストローク列を参照し、 % ␣ ^N ^A というシーケンスが繰り返されていることを検出してこれらのキー操作列に対応するコマンドを実行する。この結果テキストは図 3のように変化する。

```
% This is a sample \TeX text file.
% The area from this line to the end
% of this paragraph should be commented
out using the comment character '%.'
```

図 3: REPEAT 入力後の状態

以後 REPEAT を入力する度に 1 行ずつコメント記号と空白 “%␣” が行頭に挿入される。

5.2 複雑な繰返し操作

ここでは Nix による例 [12] を **DM** で実行してみることにする。

```
Recent research is said to have traced the earliest known use
of @i[O.K.] to the @i[Boston Morning Post] of 23 March 1839. it
was not until nearly a hundred years later that, greatly helped
by radio and television, it won its present popularity in England.
It is made to serve as an adjective (@i[That's O.K.]) and
occasionally attributive (@i[Advertising is in these days a
socially O.K. profession]); it supersedes the old formulas of assent
@i[Very well], @i[All right], and @i[Right on], ... It has bred
a jocular variant @i[Okidokey].
```

図 4: Scribe 型式の原文書 ([12])

```
Recent research is said to have traced the earliest known use
of {\sl O.K.} to the {\sl Boston Morning Post} of 23 March 1839. it
was not until nearly a hundred years later that, greatly helped
by radio and television, it won its present popularity in England.
It is made to serve as an adjective ({\sl That's O.K.}) and
occasionally attributive ({\sl Advertising is in these days a
socially O.K. profession}); it supersedes the old formulas of assent
{\sl Very well}, {\sl All right}, and {\sl Right on}, ... It has bred
a jocular variant {\sl Okidokey}.
```

図 5: 目的の TEX 型式の文書

```
Recent research is said to have traced the earliest known use
of {\sl O.K.} to the @i[Boston Morning Post] of 23 March 1839. it
was not until nearly a hundred years later that, greatly helped
by radio and television, it won its present popularity in England.
It is made to serve as an adjective (@i[That's O.K.]) and
occasionally attributive (@i[Advertising is in these days a
socially O.K. profession]); it supersedes the old formulas of assent
@i[Very well], @i[All right], and @i[Right on], ... It has bred
a jocular variant @i[Okidokey].
```

図 6: 最初の修正結果

```
Recent research is said to have traced the earliest known use
of {\sl O.K.} to the @i[Boston Morning Post] of 23 March 1839. it
was not until nearly a hundred years later that, greatly helped
by radio and television, it won its present popularity in England.
It is made to serve as an adjective (@i[That's O.K.]) and
occasionally attributive (@i[Advertising is in these days a
socially O.K. profession]); it supersedes the old formulas of assent
@i[Very well], @i[All right], and @i[Right on], ... It has bred
a jocular variant @i[Okidokey].
```

図 7: ふたつめの修正の途中

```
Recent research is said to have traced the earliest known use
of {\sl O.K.} to the {\sl Boston Morning Post} of 23 March 1839. it
was not until nearly a hundred years later that, greatly helped
by radio and television, it won its present popularity in England.
It is made to serve as an adjective (@i[That's O.K.]) and
occasionally attributive (@i[Advertising is in these days a
socially O.K. profession]); it supersedes the old formulas of assent
@i[Very well], @i[All right], and @i[Right on], ... It has bred
a jocular variant @i[Okidokey].
```

図 8: REPEAT 入力結果

5.2.1 パタン置換

図 4は文書整形言語 Scribe の文書の例である。これを図 5のような TEX の文書に変換することを考える。

ここではイタリックを示す Scribe の制御指令 “@i[文字列]” を TEX の制御指令 “{\sl␣文字列}” に変換する作業が必要になる。 Nix のシステムでは、ユーザは普通の編集操作により一部分の編集作業を行ない、編集前のテキスト (例えば “@i[O.K.]”) と編集後のテキスト (例えば “{\sl␣O.K.}”) の組をシステムに与えることによりシステムに変換規則を推論させる。ひとつの例だけでは不充分な場合は複数の例をシステムに与えることにより正しい推論が得られるようにする。

DM を使って同様の作業を行なう場合の手順は以下のようになる。図 4の状態から次のようなキー操作によりまず最初の編集を行なう。

^S @ i ESC (“@i” の検索)
DEL DEL ^D (“@i[” の消去)
(\ s l ␣ (“{\sl␣” の挿入)
^S] ESC (“]” の検索)
DEL (“] の消去)
} (“}” の挿入)

この結果テキストは図 6のように変化する。ここで次の修正をするために ^S @ i ESC (“@i” の検索) までの作業を行なうとテキストは図 7のように変化する。このときユーザが同じ処理の繰返しであることに気付いて REPEAT を入力すると、繰返し操作が検出実行され、テキストは図 8のように変化する。この後 REPEAT を 6 回入力すると図 5のテキストが得られる。

このように、 Nix のシステムと異なり、繰返し操作の開始も終了も指示せず、普通の編集操作を繰返している途中で REPEAT を入力するだけで目的の処理が実行できたことになる。

5.2.2 コメントの追加

[12] ではさらに複雑な例として、図 9のような Lisp プログラムのすべての関数の先頭にコメントを追加して図 10のようにする例が示されている。

```
(define (factorial n)
  (if (<= n 1) 1 (* n (factorial (- n 1)))))
(define (halts f)
  ( .... ))
```

図 9: コメント追加前のプログラム

```
;*** (factorial n)
;*** ----------------------------------
;*** <perspicuous description here>
;***
(define (factorial n)
  (if (<= n 1) 1 (* n (factorial (- n 1)))))
;*** (halts f)
;*** ----------------------------------
;*** <perspicuous description here>
;***
(define (halts f)
  ( .... ))
```

図 10: コメント追加後のプログラム

この例も前の例と同様に **DM** で簡単に実行することができる。最初の関数の処理は「“(define” の検索」「その位置から行末までの文字列 (この場合 “(factorial␣n)”) をテンポラリバッファに格納」「一行上に “;***␣” を挿入」「その後にテンポラリバッファ内の文字列を挿入」「残りのコメントを挿入」となるが、この後で次の関数の処理の最初の「“(define” の検索」を実行した後 REPEAT により繰返し実行を指示すると図 10のような結果が得られる。

6 評価

ここでは **DM** の主観的な評価を述べる。

- 有用性

  DM は文書の編集作業において大変便利であると感じており、他のユーザにもおおむね好評である。文字列の繰返し修正や行頭への文字列の連続挿入のようによく必要になる繰返し処理作業は Emacs では最初から用意されているものが多いが、 Emacs の機能をあまり知らないユーザでも REPEAT を押すだけでこのような繰返し処理を簡単に行なうことができる点がありがたい。

- 予測の正しさ

  著者の使用経験ではほとんどの場合意図どおりに予測が行なわれている。予測が間違っていた場合でも undo すればよいので問題はあまり発生していない。

- ユーザの手間

  ユーザが覚える必要のあるのは「繰返し操作が認められた場合 REPEAT を入力すると自動的に操作が繰返される」ということだけである。繰返し検出機能の存在を覚える必要はあるが、実際の操作をこれより簡単にすることは不可能であろう。

- 速度

  Emacs は常に直前の 100 個のキーストロークを記憶するようになっているため REPEAT を押さない限り余分な処理は必要にならず、予測機能の追加による速度の劣化は起こらない。 **DM** は単純な処理しか行なわないため REPEAT を押した際の繰返しの検出及び実行にはほとんど時間はかからない。繰返しの検出に時間がかかる

場合でも、予測を使わない場合 (操作を何度も自分で繰り返す場合) よりも速ければかまわないと考えられる。

- 操作のわずらわしさ

  REPEATを押さない限り予測機能は働かないのでユーザが **DM** の存在をわずらわしく感じることはない。これに対し、 UNIX 上に実装された Reactive Keyboard [4] のように予測を常にユーザに提示するシステムの動作は著者にはわずらわしく感じられる。

7 議論

7.1 DM がうまくいく理由

グラフィックインタフェースに予測を用いる研究が最近盛んに行なわれているが、グラフィックインタフェースにおいてはユーザの意図を正しく理解することは本質的に困難であるため [2]、システムが間違った推論を行なわないようにするため問題を簡単化したりヒューリスティクスを使ったり頻繁にユーザに問いあわせを行なったりしているものが多い [13]。

これに対し **DM** はテキスト操作に限ってはいるものの簡単な規則で予測インタフェースが成功している。この理由としては、 Emacs ではユーザの意図と操作が 1 対 1 対応している点が大きいと思われる。例えばユーザはカーソルを行頭に動かしたい場合は^Aを入力するし、左に何文字か移動させたい場合は^Bを文字の数だけ入力する。このように、Emacs では同じようにカーソルを左に動かす場合でもその意図によりユーザの操作は異なっているのが普通である。これに対しグラフィックの直接操作インタフェースでは、例えば図形をマウスで左にドラッグしたとき、それが画面の右に場所を空けるためなのか、移動量が重要なのか、別の図形と位置を揃えるためなのか、などが操作からは判断できないため推論が困難になっているわけである。 Emacs ではユーザの意図を反映した機能が別々の操作としてあらかじめ用意されておりそれをユーザが実際に使用するため、システムがユーザの意図をくみとって予測を行ないやすいということができる。

7.2 予測結果のユーザへの提示

DM のようにREPEATを入力すると予測結果をすぐに実行する方式では、何が予測されるかわからないためユーザが不安に感じることが考えられる。 Undo 機能が常に正しく働けばいつでも回復が可能なはずであるが、ユーザによってはこのような不安は確かに存在するようである。このようなユーザのためには、REPEATを入力すると実際に何が起こるのかを予測の実行前にユーザに提示することが必要になるかもしれない[2]。

[2]テキスト処理の場合は予測をユーザに提示することは簡単であるが、グラフィック操作の場合はこれが難しいためテキストを使用する [6]、自然言語を使用する [2]、絵コンテを使用する [7]、など各種の工夫がなされているが困難さは否めない。

7.3 テキストの新しい入力 / 編集方式

DM の例にみられるように、テキストの入力 / 編集方式はまだまだコロンブスの卵的な新しい方式が残されていると考えられ、現状の方式と異なる各種の方式の実験が有益と思われる。 Emacs では `completion` (単語の最初の部分だけ入力するとシステムが残りを補填する機能) とか `abbrev` (単語の最初の部分だけ入力するとシステムがテキスト中から同じ文字列で開始する単語を捜してユーザに提示する機能) といったインタフェースがよく使われているが、これらの機能は **DM** と同様に仕組みは単純であるが実用的である。仕組みが複雑でも有用さが疑問なインタフェースよりも単純でも役にたつインタフェースの発見が望まれる。

7.4 テキスト処理に予測を適用した他システムとの比較

日常的に多量のテキスト入力 / 編集を行なっているユーザにとってキーボードマクロのような機能は有用であるが、使用の前に定義が必要であることが難点である。 **DM** ではマクロ定義を明示的に示す必要が無いためキーボードマクロより受け入れられ易いと考えられる。

Nix のシステム [12] は、ユーザの操作の列から予測を行なわず、もとのテキストと編集後のテキストのみからその関係を推論する。変換プログラムの推論は面白い試みではあるが、実際の使用を考えると、 Nix のシステムでは操作前後の例を (場合によっては何度も) 明示する操作が大変そうであるのに対し、 **DM** では例示システムを動かしているという自覚すら必要ないのでユーザにとっては **DM** の方が便利であると思われる。 Nix の論文に示されている例はすべて **DM** で実行できる。

Mo のシステム [9] はユーザの操作から繰り返しや条件判断を含むプログラムを作成するため、 **DM** では不可能な複雑な操作も可能になっている。しかし汎化を行なうために少なくとも 2 度似た操作を繰り返す必要があるし常にユーザの意図を正しく汎化することは困難である。 **DM** は与えられた操作を忠実に繰り返すだけであるが、ユーザにより汎化されたキー操作が使われるため間違った汎化を行なうことはほとんど無く、また長い操作を 2 回繰り返さなくても良いという利点がある。

7.5 予測インタフェースの失敗例

Reactive Keyboard[5] と似たインタフェースとして、以前の入力パタンから次の入力を予測してユーザに提示するシステムを Emacs 上に実装し実験してみたがこれは以下の理由で失敗であった。

- 予測があまり当たらない

  テキスト圧縮に使用される PPM 法 [1] を用いて予測を行なったが、正しい予測が行なわれる確率が低すぎた。

- 予測が常に表示されるのがわずらわしい

  現在のカーソルの後ろに常に予測文字列が表示されるようにするとキー入力の度にそれがちらついて見にく

かった。

- 確認、確定がわずらわしい

  カーソルの後ろに表示される予測が正しい場合確定キーを用いてそれを承認する方式をとったが、キーを入力する度に予測結果が正しいかどうか判断し確定キーを入力するのは大きな負担になる

- 遅い

  キー入力パタンの統計をとるために、キー入力の度に文字列の発生頻度をトライ辞書に格納するようにしたところキーの反応が非常に遅くなってしまった。

このように一見便利そうな機能も実際に使ってみると大きな問題があることが判明するため注意が必要である。

7.6 DM の拡張

DM の仕様を拡張して、繰返しから自動的に定義されたマクロを後でまた使用できるように一時的にセーブしたり名前をつけてセーブしたりすることも可能である。このようにすると繰返し操作の途中で一時的に別の操作を行なってもまた前回の繰返し操作を継続することが可能になる。

8 結論

以前の操作列からの繰返しパタンの抽出という単純な仕組みでユーザの繰返し処理を簡単化することができることを示し、例示を用いたユーザインタフェースとの関係を示した。テキストの入力 / 編集作業を簡略化するための単純で新しい手法の発掘が望まれる。

謝辞

DM の実装、評価にあたり数多くの有益な助言をいただいたシャープ株式会社の市川雄二氏と舟渡信彦氏に感謝します。

参考文献

[1] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[2] Edwin Bos. Some virtues and limitations of action inferring interfaces. In *Proceedings of the ACM SIGGRAPH and SIGCHI Symposium on User Interface Software and Technology (UIST '92)*, pp. 79–88. ACM Press, November 1992.

[3] Allen Cypher. Eager: Programming repetitive tasks by example. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'91)*, pp. 33–39. Addison-Wesley, April 1991.

[4] John Darragh, Dan Freedman, Mark James, Dejan Mitrovic, Jason Penney, Doug Taylor, and Ian Witten. *Reactive Keyboard (UNIX 版)* マニュアル. University of Calgary, Calgary T2N 1N4, Canada, October 1990. University of Calgary にて配付.

[5] John J. Darragh, Ian H. Witten, and Mark L. James. The reactive keyboard: A predictive typing aid. *IEEE Computer*, Vol. 23, No. 11, pp. 41–49, November 1990.

[6] Daniel. C. Halbert. Programming by example. Technical Report OSD-T8402, Xerox Office Systems Division, December 1984.

[7] David Kurlander and Steven Feiner. A history-based macro by example system. In *Proceedings of the ACM SIGGRAPH and SIGCHI Symposium on User Interface Software and Technology (UIST '92)*, pp. 99–106. ACM Press, November 1992.

[8] David L. Maulsby, Ian H. Witten, and Kenneth A. Kittlitz. Metamouse: Specifying graphical procedures by example. In *Proceedings of SIGGRAPH'89*, volume 23, pp. 127–136, Boston, MA, July 1989.

[9] Dan H. Mo and Ian H. Witten. Learning text editing tasks from examples: a procedural approach. *Behaviour & Information Technology*, Vol. 11, No. 1, pp. 32–45, 1992.

[10] Brad A. Myers. Creating interaction techniques by demonstration. *IEEE Computer Graphics and Applications*, pp. 51–60, September 1987.

[11] Brad A. Myers. Demonstrational interfaces: A step beyond direct manipulation. *IEEE Computer*, Vol. 25, No. 8, pp. 61–73, August 1992.

[12] Robert P. Nix. Editing by example. *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 4, pp. 600–621, October 1985.

[13] 萩谷 昌己. ビジュアルプログラミングと自動プログラミング. コンピュータソフトウェア, Vol. 8, No. 2, pp. 27–39, March 1991.

付録: DM プログラムリスト

```
;;
;; dmacro.el - 繰返し検出実行    by 増井、太和田
;;
;; .emacs に以下の行を入れる
;; (defvar *dmacro-key* "\C-t") ; 繰返し指定キー
;; (global-set-key *dmacro-key* 'dmacro-exec)
;; (autoload 'dmacro-exec "dmacro" nil t)
;;
(defvar *dmacro-str* nil "繰返し文字列")
(defun dmacro-exec ()
  "キー操作の繰返しを検出し実行する"
  (interactive)
  (let ((s (dmacro-get)))
    (if s (execute-kbd-macro s)
      (message "操作の繰返しが見つかりません")
      )
    ))
(defun dmacro-get ()
  (let ((rkeys (recent-keys)) str)
    (if (string= (substring rkeys -2)
                 (concat *dmacro-key* *dmacro-key*))
        *dmacro-str*
      (setq str (dmacro-search (substring rkeys 0 -1)))
      (if (null str)
          (setq *dmacro-str* nil)
        (let ((s1 (car str)) (s2 (cdr str)))
          (setq *dmacro-str* (concat s2 s1))
          (if (string= s1 "") *dmacro-str* s1)
          )))))
(defun dmacro-search (string)
  (let* ((str (string-reverse string))
         (sptr  1)
         (dptr0 (string-match (substring str 0 sptr) str sptr))
         (dptr dptr0)
         maxptr)
    (while (and dptr0
                (not (string-match *dmacro-key* (substring str sptr dptr0))))
      (if (= dptr0 sptr)
          (setq maxptr sptr))
      (setq sptr (1+ sptr))
      (setq dptr dptr0)
      (setq dptr0 (string-match (substring str 0 sptr) str sptr))
      )
    (if (null maxptr)
        (let ((predict-str (string-reverse (substring str (1- sptr) dptr))))
          (if (string-match *dmacro-key* predict-str)
              nil
            (cons predict-str (string-reverse (substring str 0 (1- sptr)))))
          )
      (cons "" (string-reverse (substring str 0 maxptr)))
      )
    ))
(defun string-reverse (str)
  (concat "" (reverse (mapcar (function (lambda (x) x)) str))))
```