インターフェイスの街角 (4)
予測インターフェイス

増井俊之

予測インターフェイスの効用

計算機は単純作業の効率化に向いているはずですが、仕事の内容によっては、人間が単調な作業を繰り返さなければならないことがまだまだ多いようです。たとえば、次のような場合を考えてみましょう。

- 文書中の何行ぶんかの先頭にコメント記号を入れたい
  10 行程度なら、〝行頭へ移動→コメント記号を入力→次の行へ移動〟といった基本操作を繰り返してもたいして手間はかかりませんが、100 行、1,000 行ともなると、なんらかの方法で自動化したくなります。このような機能がエディタにない場合は、マクロやプログラムを書く必要があります。かといって、この種の特殊な機能をエディタにむやみに詰め込むとアプリケーションが肥大化します。
- 前回と同じ条件で再コンパイル/リンクを実行したい
  同じ条件で何度もコンパイル/リンクすることがあらかじめ分かっていれば、Makefile を書くのが普通です。しかし、コマンド行でコンパイルして、予想外のエラーなどでうまくいかなかったときなど、Makefile を書いておけばよかったと後悔することがあります。
- 値を変えて前の計算を再実行したい
  電卓でいろいろな計算をしたあとで、値を変えて同じ計算を実行したいことがあります。最初からプログラミング電卓を使えばよいのですが、ちょっとした計算のためにわざわざプログラムを作るのも面倒です。けっきょくは、同じ操作を何度も繰り返して計算することが多いのではないでしょうか。

このように、単純で特殊な作業を繰り返す場合、自動化プログラムの作成にかかる手間に見合わないために、けっきょくは人間が作業せざるをえないことがよくあります。

このような問題を解決するために、2 種類のアプローチが考えられています。

1 つは、誰もが自力でプログラムを作ってこれらの作業を自動化できるようにする、いわゆる**エンドユーザー・プログラミング**を支援しようというアプローチです。たしかに、誰もがエディタのマクロや Makefile などを簡単に書けるようになればこういった問題は減るかもしれません。しかし、〝100 行の行頭にコメントを入れるマクロ〟を誰もがすぐに書けるような時代が到来するとはちょっと考えられませんし、たとえそうだとしても、プログラミングの手間が減るわけではありません。

もう 1 つは、**予測インターフェイス**を用いて同じような反復作業を少ない手間で実行しようというものです。たとえば、人手で何行かにコメント文字を挿入する操作を実行したあとでは、システムは〝おそらく次もコメントを入れるだろう〟と予測できますから、それ以降の操作を〝あとはやっといて〟と計算機に任せることができるかもしれません。このように、ユーザーの操作の履歴などから次の操作をうまく予測できれば、単調な操作を何度も繰り返さずにすむかもしれません。

UNIX の予測インターフェイス

UNIX のシェル上での仕事は単純な繰返し作業の巣窟ですから、シェル・スクリプトによるエンドユーザー・プログラミングに加えて、各種の予測インターフェイスがひろく使われています。もっとも普及している予測インター

フェイスはコマンド履歴を利用した ``!'' 系のヒストリー置換でしょう[1]。コマンドやファイル名の補完 (completion) も有用な予測インターフェイスの例で、単純作業の省力化に役立ちます。

コマンドやファイルの名前を利用した予測インターフェイスはたいへん便利ですが、これでは不十分な場合も考えられます。たとえば、

```
% ftp abra.cadabra.co.jp
% make
……（たくさんの仕事）……
% telnet abra.cadabra.co.jp
```

といった作業をおこなう場合、ヒストリー置換や補完ではホスト名の入力は効率化できません。しかし、キーストロークの履歴を利用して予測対象をひろげれば、``abra'' と入力しただけで ``abra.cadabra.co.jp'' をシステムに予測させたりすることも可能になります。

この方法を用いた予測インターフェイス・システム ``Reactive Keyboard'' を紹介します。

Reactive Keyboard

Reactive Keyboard[2] (以下、RK と略) は、シェル上のキーストロークの履歴をもとにユーザーの次の入力を予測/提示する予測インターフェイス・システムです。ユーザーがシェル上でなんらかの入力をするたびに次の入力文字を予測し、その結果をコマンド行に反転文字で提示します。予測結果が妥当であれば、ユーザーはそれを採用して入力の手間を減らすことができます。

使用例

シェルに sh を指定して[3]RK を起動すると、

```
% setenv SHELL /bin/sh
% rk
Welcome to the Reactive Keyboard, ...
...
$ ^J
```

のように、プロンプトの後ろに反転表示された ``^J'' が表示され、RK が改行キーの入力を予測していることを示します (網掛け部はカーソル位置を示しています)。RK では、予測結果がつねにカーソルの後ろに反転表示されますが、初めて RK を起動したときは予測に使える情報がまったくないので、改行だけが予測/表示されます。

ここで ``echo masui'' と入力すると、表示は、

```
$ echo masui^J
```

になりますが、改行キーを押して echo コマンドを実行すると、

```
$ echo masui^J
```

と変化します。``echo masui'' というコマンドが直前に実行されているため、同じコマンドの起動を予測して表示するわけです。この予測結果が正しい場合は、Control-S を押せば、予測結果を全面的に採用して ``echo masui'' を再度実行することができます。

予測結果が求めるものと違うときは、表示を無視してキー入力を続けます。たとえば date コマンドを実行する場合、``d'' を入力すると表示は下記のように変化します。

```
$ decho masui^J
```

この場合、システムは ``d'' の後ろに何が続くかを予測できないため、とりあえず ``e'' とそれに続く文字列を提示しています。この予測は正しくないので、続けて ``a'' を入力すると、表示は次のように変化します。

```
$ dasui^J
```

最初の echo の例から推測し、``a'' の後ろには ``s'' が入力されることが多いと判断し、さらに ``s'' の後ろには ``u'' が入力されることが多いと判断し、…… という計算によってこのような予測結果が表示されます。

予測を無視して ``te'' まで入力すると、

```
$ datecho masui^J
```

という表示になりますが、ここでリターンキーを押すと date コマンドが実行され、表示は次のようになります。

```
$ date^J
```

ここで ``e'' を押すと、

1 メカニズムがはっきりしているので ``予測'' というのは変かもしれませんが、間違うこともときどきありますし、当たる確率がきわめて高くて役立つ予測インターフェイスの例といえるでしょう。

2 ftp://ftp.cpsc.ucalgary.ca/pub/projects/the.reactive.keyboard/から入手できます。論文 [3] や書籍 [2] も出版されています。

3 Reactive Keyboard にはコントロールキーによる編集機能があり、編集機能のあるシェルでは動かないことがあるので、ここでは /bin/sh を使います。

```
$ ec[ho masui^J]
```

と変化します。

このように、ユーザーが何かを入力するたびに、その並びの統計情報を利用して、その次に入力される文字列を予測して表示する仕組みです。

最初に挙げた例のように、それ以前に、

```
$ ftp abra.cadabra.co.jp
```

を実行したことがある場合には、〝ab〟まで入力すれば、

```
$ telnet ab[ra.cadabra.co.jp^J]
```

のように残りの部分が予測されて表示されます。

Reactive Keyboard の仕組み

RK は、ユーザーの入力したキーストロークの履歴から次に入力される文字を予測して提示します。次のキー入力を予測するために、テキスト圧縮の技法の 1 つである PPM 法を使っています。

PPM 法

PPM (Prediction by Partial Match) 法 [1] は、ある時点までのデータの並びから次のデータを予測するアルゴリズムで、可逆的なファイル圧縮[4]に利用されています。ファイルを先頭から眺めていき、すでに見たデータから後続のデータをうまく予測できれば、次のデータを表現するために必要なビット数が少なくてすむので[5]、予測手法とデータ圧縮法には密接な関連があります[6]。

PPM 法の基本は、直前の文字列からの次の 1 文字の予測です。たとえば、〝abracadabra〟という文字列では、

- a の後ろには b が 2 回、c が 1 回、d が 1 回出現
- ra の後ろには c が 1 回出現
- bra の後ろには c が 1 回出現

といった実績を考慮して次の文字の出現確率を予測します。直前の 1 文字だけを考慮すると、b の出現確率は a、c の倍程度と考えられますし、また a、b、c 以外の文字が出現する可能性もあります。これらを勘案して、たとえば、

文字	出現確率
b	2/5
a	1/5
c	1/5
その他	$1/5 * 1/(26-3)$

のように出現確率を予想できます。直前の 2 文字や 3 文字を考慮する場合も同様の予測が可能なので、これらに適当な重みをつけたうえで足し合わせて最終的な各文字の出現確率を計算します。データ圧縮をおこなうときは出現確率に応じて符号化し、予測インターフェイスに使うときはもっとも値の大きいものを候補として提示します。

PPM 法は予測手法として優れているため、算術符号化と組み合わせれば、場合によっては gzip などよりも圧縮率の高い可逆圧縮が可能になります。

Reactive Keyboard の困ったところ

RK はおもしろい特徴をもつシステムですが、計算機ユーザーのあいだではあまり人気がないようです。これは、おそらく次のような理由によるものでしょう。

- 予測の表示がうっとうしい
  何か入力するたびに予測結果が表示されるため、高速にタイピングすると行がちらついてかなり気になります。
- 予測が正しいとはかぎらない
  つねに正しい予測結果が表示されるのならばいいのですが、前述のような PPM 法では、字面が似ているというだけで無意味な文字列を予測して提示してしまいます。
- 予測結果が突然変わる
  RK では、統計的な確率計算にもとづき、もっとも確率の高い文字列を候補として表示します。したがって、たとえば date コマンドを続けて何回か実行すると、次のコマンドとして date を予測します。ここで ps コマンドを実行しても、次のコマンドとしてはやはり date が予測されます。ところが、ps を何回か実行し続けると、ある時点で突然 ps を次のコマンドとして予測、提示し

4 gzip や compress のようにもとのデータを復元できる圧縮方式を〝可逆的 (lossless) 圧縮〟と呼び、JPEG などのように復元不可能な圧縮方式を〝不可逆 (lossy) 圧縮〟と呼びます。

5 たとえば、次の文字が a、b、c、d である確率がまったく分からない場合、これらをコード化するためには 2 ビットが必要です。しかし、予測や統計情報により、a の確率が 50%、b が 25%、c と d がそれぞれ 12.5%であることが明らかなときは、a を 0、b を 10、c を 100、d を 101 などとコード化し、$1*0.5+2*0.25+2*3*0.125 = $ 平均 1.75 ビットでこれらの 4 文字を表現できます。情報の生起確率から計算されるこのような最短平均符号長は〝情報のエントロピー〟と呼ばれ、情報理論や圧縮理論の基礎となっています。

6 テキスト圧縮によく使われる compress や gzip などの圧縮アルゴリズムは、圧縮したいファイルを最初から眺めていった場合、前と同じようなパターンが出現したときにそれを短い符号で表現することにより、結果的に予測にもとづく短い符号化をおこなっています。

図 1　コメント挿入の繰返し

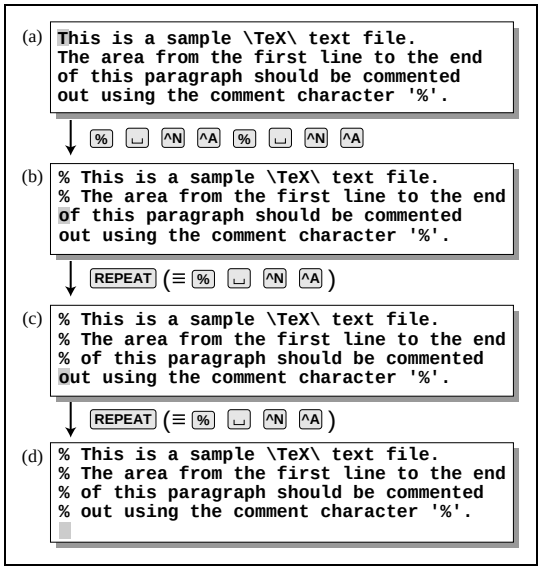
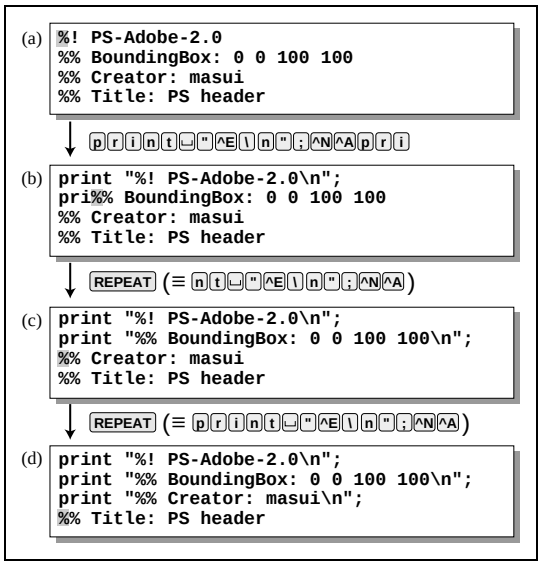


図 2　print の挿入



始めます。この例からも分かるように、RK の予測結果を予測するのはかなり困難です（できの悪いかな漢字変換システムに学習させるような感じになります）。

- 予測結果を採用するより、自分で入力するほうが速い

これが一番の理由かもしれませんが、キーボードを高速にタイプできる人にとっては、RK のような予測手法はそれほど便利だとはいえません[7]。

RK などのシェル上での字面履歴を用いた予測システムは以上のような理由であまり普及していませんが、もっと簡単な仕組みでも場合によってはかなり役に立つ予測インターフェイスが作れます。その一例として、私が開発した〝Dynamic Macro〟という予測システムを紹介しましょう。

Dynamic Macro

Dynamic Macro[4] は GNU Emacs 上で動作する予測インターフェイスで、直前に繰り返した操作を〝繰返しキー〟によって再実行させる Emacs Lisp プログラムです。どのような操作列でも、直前に繰り返したものであれば繰返しキーを押すだけで何度でも続けて実行できるため、さまざまな場面に応用できます。

7 RK の開発者は、手が不自由であったために有用だったそうです。

使用例

コメントの挿入

ある行の先頭にコメント記号を挿入し、次の行の先頭にもコメント記号を挿入した状態（図 1-b）で繰返しキーを押すと、次の行の先頭にもコメント記号が挿入されます（図 1-c）。さらに繰返しキーを押すと、その次の行にもコメント記号が挿入されます（図 1-d）。

フォーマット変更

Dynamic Macro は、もっと複雑な作業の繰返しにも使えます。図 2-a は PostScript のヘッダの例ですが、これを出力するプログラムを作るために print 文を入力していくような繰返し作業（図 2-b）も、繰返しキーにより自動的に実行させることができます（図 2-c〜d）。

Dynamic Macro の特徴

Dynamic Macro の原理はきわめて単純で、繰返しキーが押された時点までのキーストロークの履歴を調べ、同じシーケンスが繰り返されている場合にはそれをキーボードマクロとして登録し、実行するだけです。GNU Emacs では recent-keys 関数を用いて 100 個までのキーストローク履歴を取得できるので実現は簡単です。Dynamic Macro のソースを末尾に示します。コメント部に書いてあるように、.emacs などで繰返しキー (∗dmacro-key∗)

に適当なキーを割り当てて使います。

Emacs 上で同じ作業を何度も繰り返す場合、キーボードマクロとして登録して呼び出すのが一般的です。しかし、最初の例のような簡単な作業の繰返しでは、繰り返される作業よりもマクロ登録開始/終了/呼出しの手間のほうが大きくなりがちです。この点、Dynamic Macro は、マクロ登録の手間がいらず、しかも繰返しキーを押すだけで作業を繰り返し実行させることができるので便利です。また、キーボードマクロでは繰り返される作業の開始と終了を正確に指定しなければなりませんが、Dynamic Macro ではその必要はありません。たとえば、最初の例では、間違えて〝次行に移動→行頭に移動→コメント文字挿入→次行に移動〟というシーケンスをマクロとして登録しがちですが、Dynamic Macro ではそのような心配はありません。操作の実行後に繰返しに気づいた場合も適用できるという特徴もあります。

このように、Dynamic Macro は原理が単純なわりにうまく働く応用範囲の広い予測インターフェイスです。

自動再計算電卓

予測インターフェイスのもう 1 つの例として、〝Undo にもとづく自動再計算電卓〟を紹介します。

普通の電卓でパラメータを変えながら同じ計算をするときは、同じ操作を何回も繰り返す必要があります。プログラム電卓を使ってもいいのですが、わずかな計算のためにわざわざプログラムを作るのは面倒ですし、プログラミングそのものが苦手な人も多いと思います。しかし、すでに 1 度計算がうまくいっている場合、パラメータを入れてから計算結果が出るまでの経路を Undo によって逆にたどり、パラメータを入れなおして残りを再計算させることができれば、再度同じ操作をしなくても前回と同じ計算が実行できます。このような再計算電卓 **ebcalc**（Perl スクリプト）について説明します。

再計算電卓の使用例

いくつかのセ氏温度をカ氏に変換する仕事を考えてみましょう。セ氏 Cとカ氏 Fには、

$$F = C * 9/5 + 32$$

という関係があります。ebcalc を起動してセ氏 10 度をカ氏に変換するときは、次のキー入力をおこないます。

```
% ebcalc
0
（10、*、9、/、5、+、32、=を入力）
50
```

この結果、セ氏 10 度はカ氏 50 度に対応することが分かります。続いて、セ氏 20 度、30 度 …… について同じ計算をする場合、通常は全部の操作を繰り返す必要があります。しかし、ebcalc は計算履歴を記憶しているので、後退キーによって計算状態をもとに戻し、数値を変更してから〝=〟を押して再計算させることができます。

上の計算が終った時点で後退キーを押すと、ebcalc の表示は以下のように変化します。

```
32
```

これは最後に入力した値を示しており、ここで値を変えて=を押すと前述の式の〝32〟をその値に変えて再計算した結果が表示されます。

さらに後退キーを 3 回押すと、表示は、

```
10
```

のように最初に入力した値（セ氏 10 度）になります。ここで〝20〟と入力して値を 20 に変えてから=を押すと、

$$20 * 9/5 + 32$$

が計算され、

```
68
```

のようにセ氏 20 度はカ氏 68 度に相当することが分かります。ここで〝30〟と入力して=を押すと、今度は、

$$30 * 9/5 + 32$$

が計算され、

```
86
```

と表示されます。

このように、わざわざプログラミング電卓を使わなくても、パラメータを変えるだけで何回でも再計算できます。

ebcalc を試行錯誤的に使えば、通常は計算できない関数の計算も可能です。たとえば、10 の 3 乗根を計算してみましょう。四則演算機能しかない電卓ではこのような計算はできませんが、ebcalc を使えば次のようにおおよその値が得られます。

まず、 ~2**=~と入力して 2 の 3 乗を計算すると、表示は以下のようになります。

```
8
```

ここで後退キーを押すと、最初に入力した ~2~が表示されるので、代わりに ~2.2=~と入力して 2.2 の 3 乗を計算すると、

```
10.648
```

と表示されます。つまり、10 の 3 乗根は 2.2 より小さいことが分かります。今度は ~2.16=~と入力すると、

```
10.078
```

と、かなり 10 に近づいてきたので、10 の 3 乗根はおおよそ 2.16 弱であることが分かります。

ebcalc の仕組み

ebcalc のソースを末尾に示します。ebcalc は、キー入力の履歴を変数 $exp に記録しています。後退キーは、通常は入力した数字の訂正に使われますが、計算実行の直後に押されたときは、上例のように計算履歴をたどって以前入力した数値を表示し、訂正可能にします。=が押されると、訂正された値を用いて全体を再計算して表示します。

おわりに

今回は、いくつかの予測インターフェイスを紹介しました。現在のところ、予測インターフェイスはひろく普及しているとはいえませんが、状況によってはたいへん役に立ちます。予測インターフェイス全般についてのサーベイを http://www.csl.sony.co.jp/person/masui/papers/PBESurvey/で、Dynamic Macro と ebcalc のソースを http://www.csl.sony.co.jp/person/masui/UnixMagazine/で公開していますのでご参照ください。

今回とりあげた予測インターフェイスは操作の手間を軽減するためのものでしたが、文書入力の手間を減らすために予測インターフェイスの手法を応用することもできます。次回は、文書入力に予測インターフェイスを適用したシステム POBox を紹介します。

（ますい・としゆき　ソニー CSL）

[参考文献]

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

[2] John J. Darragh and Ian H. Witten, *The Reactive Keyboard*, Cambridge University Press, 1992

[3] 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

[4] 増井俊之、中山健「操作の繰返しを用いた予測インタフェースの統合」、コンピュータソフトウェア、Vol.11、No.6、pp.484–492、1994 年 11 月

• Dynamic Macro のソース (dmacro.el)

```
;;
;; ● 設定方法
;;  .emacsなどに以下の行を入れてください。
;;
;; (defconst *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")
(setq dmacro-keys (concat *dmacro-key* *dmacro-key*))

(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 (null s)
      (message "操作の繰返しが見つかりません")
      (execute-kbd-macro s)
      )
    ))
```

```
(defun dmacro-event (e)
  (cond
   ((integerp e) e)
   ((eq e 'backspace) 8)
   ((eq e 'tab) 9)
   ((eq e 'enter) 13)
   ((eq e 'return) 13)
   ((eq e 'escape) 27)
   ((eq e 'delete) 127)
   (t 0)
   ))

(defun dmacro-recent-keys ()
  (let ((s (recent-keys)) )
    (if (stringp s) s
      (concat (mapcar 'dmacro-event s))
      )
    ))

(defun dmacro-get ()
  (let ((rkeys (dmacro-recent-keys)) str)
    (if (string= dmacro-keys (substring rkeys (- (length dmacro-keys))))
        *dmacro-str*
      (setq str (dmacro-search (substring rkeys 0 (- (length *dmacro-key*)))))
      (if (null str)
          (setq *dmacro-str* nil)
        (let ((s1 (car str)) (s2 (cdr str)))
          (setq *dmacro-str* (concat s2 s1))
          (setq last-kbd-macro *dmacro-str*)
          (if (string= s1 "") *dmacro-str* s1)
          )))))

(defun dmacro-search (string)
  (let* ((str (string-reverse string))
         (sptr  1)
         (dptr0 (string-search (substring str 0 sptr) str sptr))
         (dptr dptr0)
         maxptr)
    (while (and dptr0
                (not (string-search *dmacro-key* (substring str sptr dptr0))))
      (if (= dptr0 sptr)
          (setq maxptr sptr))
      (setq sptr (1+ sptr))
      (setq dptr dptr0)
      (setq dptr0 (string-search (substring str 0 sptr) str sptr))
      )
    (if (null maxptr)
        (let ((predict-str (string-reverse (substring str (1- sptr) dptr))))
          (if (string-search *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))))

(defun string-search (pat str &optional start)
```

```
(let* ((len (length pat))
       (max (- (length str) len))
       p found
       )
  (setq p (if start start 0))
  (while (and (not found) (<= p max))
    (setq found (string= pat (substring str p (+ p len))))
    (if (not found) (setq p (1+ p)))
    )
  (if found p nil)
  ))
```

• **ebcalc のソース**

```
#!/usr/local/bin/perl
require 'cbreak2.pl';

&cbreak;
$| = 1;

while(1){
  $s = getc;
  next if $s eq '';
  last if $s eq "\003";
  &calculator($s);
}
&cooked;
print "\n";

sub calculator {
  local($_) = @_;
  if(/[\d\.]/){
    if($prev =~ /=/){
      $value = $operator = '';
      $exp = $rest = '' unless $undomode;
    }
    $arg = '' if $prev !~ /[\d\.]/;
    $arg .= $_;
    $exp .= $_ unless $undomode;
    &display($arg);
  }
  elsif(/[\010\177]/){
    if($prev =~ /[\d\.\010\177]/ &&
       ! $undomode){
      if(length($arg) > 0){
        chop($arg);
        &display($arg);
      }
    }
    else {
      if(! $undomode){
        $undomode = 1;
        $rest = $d = '';
      }
      if($exp =~
       /^(.*[^\d\.])?([\d\.]+)([^\d\.]*)$/){
        $exp = $1; $rest = $3.$d.$rest;
        $d = $2;
        &display($d);
      }
    }
  }
  elsif(/[\+\*\-\/]/){
    if($undomode){
      $undomode = 0;
      $value = $operator = '';
      $exp = $arg; $rest = '';
    }
    unless($prev =~ /=/){
      $value = $operator ?
        eval "$value $operator $arg" : $arg;
    }
    $operator = $_;
    $exp .= $_;
    &display($value);
  }
  elsif(/=/){
    if(! $undomode){
      $value = eval "$value $operator $arg";
      $exp .= $_;
      &display($value);
    }
    else {
      $expbak = $exp; $restbak = $rest;
      $t = $exp . $arg . $rest;
      $exp = $rest = $value =
      $operator = $arg = '';
      $undomode = 0;
      $prev = '';
      while($t =~ s/.//){
        &calculator($&);
      }
      $exp = $expbak; $rest = $restbak;
      $undomode = 1;
    }
  }
  elsif(/c/i){
    $value = $operator = '';
    $exp = $rest = '';
    $undomode = 0;
    &display(0);
  }
  $prev = $_;
}

sub display {
  local($v) = @_;
  print "$v\n";
}
```