インターフェイスの街角 (47) — 例示プログラミング

増井 俊之

エンドユーザー・プログラミング

9 月初めにイタリア北部のストレサで開催された Symposium on End-User Programming[1]に参加してきました。プログラマーではない普通のユーザーでもプログラミングができるようなシステムについて議論するカンファレンスで、この分野についての会議が開かれたのは初めてだそうです。

かつて、パソコン教室では、計算機ハードウェアの原理や BASIC などのプログラミング言語を教えたりするのが普通でした。最近は様変わりし、アプリケーションやインターネットの利用方法の学習が中心のようです。もちろん、プログラミングの知識がまったくなくても、とくに苦労することなく計算機を扱えるようになったのはいいことだと思います。しかし、プログラミングは難しいとか、プログラムを書く人と使う人は完全に別でかまわないというような考え方が定着してしまうと、せっかくの計算機の能力が十分に活かせなくなってしまうのではないでしょうか。

計算機を使いやすくするために、ソフトウェアに各種のオプションを用意しておき、ユーザーが好みに合わせてカスタマイズできるようにする方法がよく用いられています。しかし、計算機を本当に使いこなすには、多少なりともプログラミングの知識があるほうが望ましいのは明らかでしょう。そのためにも、エンドユーザー・プログラミングがもっと一般的になってほしいものです。

本格的なプログラミングは難しいかもしれませんが、決まった時刻になったら目覚まし時計のアラームを鳴らすといった〝プログラミング〟であれば、誰もが毎日のように実行しています。エンドユーザー・プログラミングといっても、こういった簡便なシステムなら普及する可能性があるかもしれません。計算機のプログラミングとくらべて、目覚まし時計の設定が簡単に感じられるのは、おそらく次のような理由からでしょう。

- 機能が限られている。
- 針の移動が直感的で目に見える。
- 針の状態を示すことにより、将来の時刻を表現できる。

一般的なプログラミング言語では、〝時刻を表す変数 t に値 7 を代入する〟とか、〝t の値が 5 より大きくなったときに alarm() を実行する〟といった抽象的な概念や、それに関連する知識が必要になります。さらに、実行手順も指定しなければなりません。

これに対して目覚まし時計の場合には、針の移動をシミュレートする〝動作例〟を示すだけでよく、実際に針を動かしてアラームが鳴るかを確認することもできます。

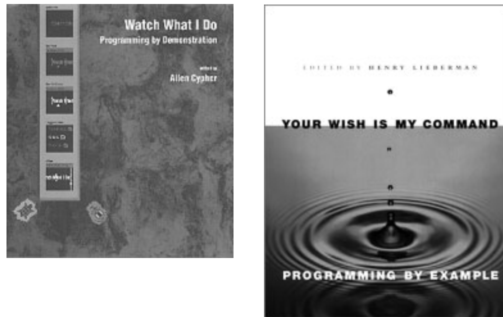
つまり、通常のプログラミングでは、ユーザーが実行したいことをシステムに伝えるのではなく、それをどのように実行するかをシステムに教えなければならないわけです。プログラミングの難しさの本質は、このあたりにあるのではないでしょうか。

どのように実行するかをあまり考える必要がなく、実行結果または動作例だけをシステムに示すことでプログラムが作れるのなら、エンドユーザー・プログラミングはもっと身近なものになるかもしれません。

このように、手順を細かく指定するのではなく、実行例や操作例にもとづいてプログラムを生成する手法を例示プログラミング、PBE (Programming by Example)、あ

1 http://iis.cse.eng.auburn.edu/~EUP/

図1　PBE 関連書籍



るいは PBD (Programming by Demonstration) などと呼びます。

PBE に関する書籍

Allen Cypher は、PBE に関するさまざまな研究をまとめた『Watch What I Do』[1] を著しています（図 1 左)。刊行時（1993 年）までの PBE の研究の多くは、この本にまとめられています。

その後も、PBE に関する研究は活発に続けられています。MIT の Henry Lieberman が編纂した『Your Wish is My Command』[2]（図 1 右）は前記の本の続編ともいえるもので、最近の研究動向がまとめられています。

PBE の実例

これらの書籍で紹介されている PBE システムをいくつか紹介します。

Stagecast Creator

Apple で PBE について研究していた Allen Cypher と David Smith は、この手法を応用して、子どもでも簡単にテレビゲームのようなプログラムが作れる「KidSim」[4] というシステムを開発しました。

図 2 は、KidSim で魚が泳ぐアニメーション・プログラムを作成しているところです。左上のマス目付きのウィンドウがこれから作る画面で、その下にあるウィンドウが規則を表現しています。ここでは、〝2 × 2 のマス目の左上に魚がいて残りに何もない場合には魚が右下に移動する〟という書換え規則を定義しています。この規則を繰り返し適用することにより、最初は画面の左上にいた魚がだんだん右下に泳いでいくアニメーションが表示されます。このように、KidSim のプログラムは具体的で分かりやすいので、子どもでもいろいろな規則を作り、複雑なアニメーションやゲームを簡単に作成できます。

図2　KidSim

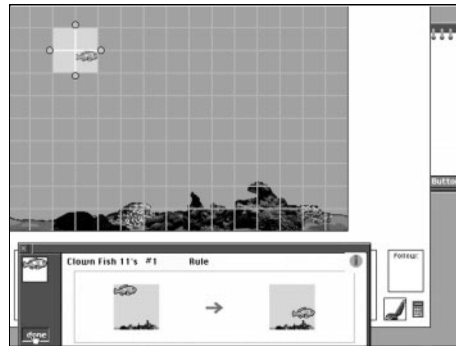
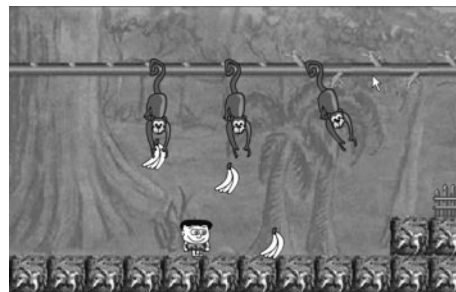


図3　Stagecast Creator で作ったゲーム



現在、Cypher らは Stagecast という会社を設立し、KidSim の機能を拡張した「Stagecast Creator」という製品を販売しています[2]。Stagecast Creator は、KidSim と比較してさまざまな機能拡張がなされていますが、キャラクターの動作例を記述してプログラムを完成させる点は同じです。

Stagecast のサイトには、Creator を使って子どもたちが作ったプログラムがたくさん掲載されています。

ToonTalk

ToonTalk[3]は Animated Programs の Ken Kahn が開発したビジュアルな PBE システムです。

図 4 からも分かるように、ToonTalk ではレゴに似た世界のなかでロボットやブロックを使ってプログラムを作っていきます。ロボットはメソッドに対応し、ブロッ

2 http://www.stagecast.com/
3 http://www.toontalk.com/

図 4　ToonTalk の使用例

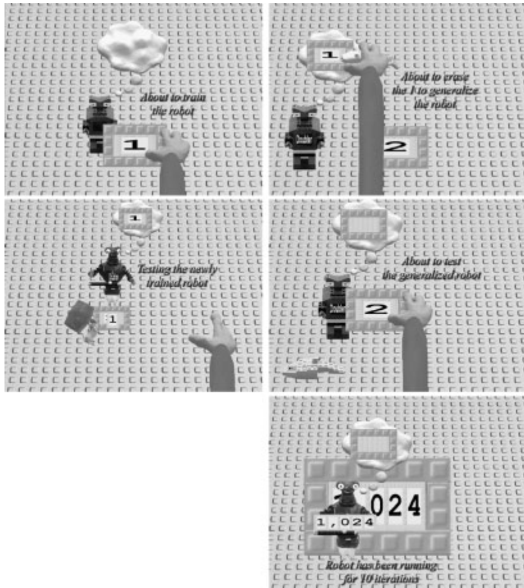
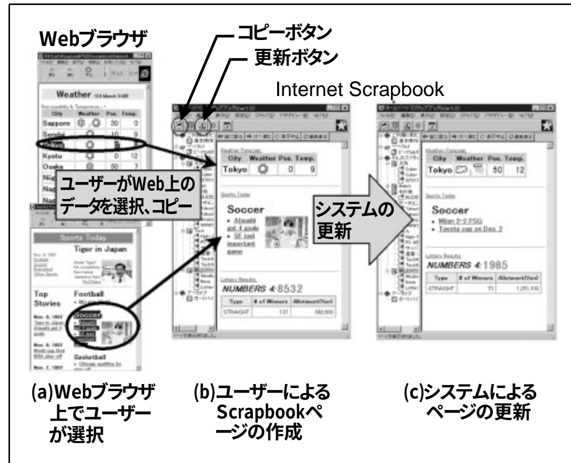


図 5　Internet Scrapbook



クはメッセージや配列に対応するといったふうに、あらゆるプログラムの要素が画面上のオブジェクトに対応しています。トラックを走らせてプロセスを起動したり、鳥を飛ばしてメッセージを配送するなど、高度なプログラミング要素もすべて画面上のキャラクターやオブジェクトとして表現されます。

図 4 は、数の 2 倍をどんどん計算していくプログラムを例示によって作成している様子を示しています。

左上の図は、ロボットに〝1〟を与えて挙動を学習させようとしているところです。与えた 1 をコピーしてもとの数字に重ねると、ハンマーを持ったネズミが現れて、それらの値を足しあわせるようになっています。入力として 1 が与えられるとそれを自分に足して計算し、2 を導くといったメソッドが生成されます。

このとき、ロボットは入力として 1 を期待しているので、1 以外の数字を与えても何もおこないません。〝1 を期待している〟ことは、ロボットの右上の吹出しに表現されています。ここで、図 4 の右上のように〝クリーナー〟を使ってロボットの〝期待値〟を消すと、ほかの数字が与えられても加算処理を実行します。一般的な PBE システムでは、システムが汎化 (generalization) をおこなうものが多いのですが、ToonTalk の特徴はユーザーが明示的に汎化を指定することにより、プログラムを作成していける点にあります。

ToonTalk も、子どもでもプログラミングができるようにすることを目標としています。

Internet Scrapbook

日本電気の杉浦 淳氏は、ユーザーが Web ブラウザ上で興味のある箇所を切り貼りすることにより、それらの部分だけを表示するページを簡単に作れる「Internet Scrapbook」を開発しました [5]。

図 5 は、異なる Web ページ (a) から必要な部分だけを取り出して別のページに貼り付けている様子を表しています (b)。Internet Scrapbook は、データそのものを切り貼りするのではなく、切出しテンプレートを自動的に生成して使うようになっています。もとのページのデータが変更された場合も、貼り付けられたページが (c) のように自動的に更新される点に特徴があります。

頻繁に更新されるニュースページなどでは、見出しを示すタグやテーブルのタグの形式はほぼ同じであることが多いようです。したがって、簡単な規則を適用してテンプレートを作成しても、たいていはうまく内容を切り出せるそうです。

Internet Scrapbook は、「ホームページスクラップブック」という名前で製品化されました。その後、いわゆるオートパイロット・ソフトと統合され、現在は「ホームページスクラップブック EX」[4]という製品名で販売され

4 http://www.amuseplus.com/product/scbookex/

ています。

SmartMake

私も、例を明示的に指定しなくてもよい簡単なシステム「SmartMake」を作ってみました。これは、Makefile を自動的に生成するための Perl スクリプトです。

小さなプログラムをスクラッチから書く場合には、単純なものであれば途中で、

```
cc test.c
```

などとして実験することが多いと思います。ところが、プログラムが大きくなってきて複数のファイルに分けざるをえなくなったり、あるいは各種のライブラリやコンパイル・オプションの指定が必要になってくると、それらを 1 つ 1 つコマンドラインで実行するのは大変な作業になってしまいます。そんなときこそ、Makefile の出番です。

Makefile は、エディタで記述してもいいでしょう。しかし、コンパイル・オプションなどをすでに何回もコマンドラインで指定しているのであれば、これらをうまく利用したいものです。SmartMake は、コマンドラインでの作業をシステムに与えられた〝コンパイル例〟とみなし、必要とされる Makefile を自動的に生成するシステムです。

たとえば、

```
% vi test.c
% cc test.c
```

という 2 つのコマンドを実行した場合、各コマンドの起動、ファイル test.c、a.out の参照・更新は図 6 の順でおこなわれます。

ここで、コマンド起動後のファイルの参照・更新時刻を調べることにより、それぞれのコマンドがどのファイルを参照、生成したかが分かります。たとえば、コマンド〝vi test.c〟の起動によってファイル test.c が参照され、かつ更新されていることが分かれば、Makefile ではファイル test.c の依存関係を以下のように表現できます。

```
test.c:
        vi test.c
```

また、図 6 の 4〜6 から、コマンド〝cc test.c〟の起動によりファイル test.c が参照され、さらに a.out が更新されることが分かります。したがって、これらのコマンドとファイルの依存関係は以下のように記述できます。

図 6　コマンド起動とファイル参照/更新の順番

1. vi　test.c コマンドの起動
2. test.c の参照
3. test.c の更新
4. cc　test.c コマンドの起動
5. test.c の参照
6. a.out の更新

……

図 7　Makefile の生成規則

更新されたファイルのリスト　':'　参照されたファイルのリスト
TAB文字 **コマンド**

```
a.out: test.c
        cc test.c
```

このように、コマンドを起動するたびにファイルの参照・更新時刻を調べれば、ファイルとコマンド間の依存関係が簡単に分かります。依存関係は、図 7 に示す簡単な規則で生成できます。

なお、同じファイルが 1 つのコマンドで参照・更新されている場合には、〝参照されたファイルのリスト〟には加えません。

SmartMake の使用例

一例として、UNIX 上のファイル test1.c、test2.c をまとめたアーカイブ・ファイル test.tar を作り、これを圧縮して test.tar.Z とし、さらにこれを uuencode 形式に変換したテキストファイル test.uu を作成してみましょう。

この作業は、以下の手順でおこなわれます。

1. test1.c と test2.c から test.tar を作成
   `% tar cf test.tar test1.c test2.c`
2. test.tar を圧縮し、test.tar.Z を作成
   `% compress < test.tar > test.tar.Z`
3. test.tar.Z から test.uu を作成
   `% uuencode test.tar.Z < test.tar.Z > test.uu`

これらの一連のコマンドの起動およびファイルの参照・更新を時間順に並べると、以下のようになります。

1. tar コマンドの起動
2. test1.c の参照

3. test2.c の参照
4. test.tar の更新
5. compress コマンドの起動
6. test.tar の参照
7. test.tar.Z の更新
8. uuencode コマンドの起動
9. test.tar.Z の参照
10. test.uu の更新

ファイルの参照・更新時刻は、stat() で知ることができます。また、tcsh を利用している場合は、~/.history ファイルからコマンドの起動時刻が得られます。これらを調べてソートし、smartmake コマンド（末尾のリスト 1）を実行する仕組みです。具体的には、以下のようにします（誌面の都合上、⇒ で折り返しています。以下同様）。

```
% history -S
% smartmake
test.tar: test1.c test2.c
        tar cvf test.tar test1.c test2.c
test.tar.gz: test.tar
        gzip < test.tar > test.tar.gz
test.uu: test.tar.gz
        uuencode test.tar.gz < ⇒
test.tar.gz > test.uu
%
```

まず、〝history -S〟コマンドを実行し、コマンド実行の履歴を ~/.history ファイルに書き出します。そして smartmake コマンドを実行すると、図 7 の規則に従った Makefile（図 8）が生成されます。

図 8　自動生成した Makefile

```
test.tar: test1.c test2.c
        tar cf test.tar test1.c test2.c
test.tar.Z: test.tar
        compress < test.tar > test.tar.Z
test.uu: test.tar.Z
        uuencode test.tar.Z < test.tar.Z > ⇒
        test.uu
```

PBE システムの現状と課題

さきほども述べたように、PBE の目標は、エンドユーザーもプログラムを作ることを通じて、計算機を自由に操作できるようにすることにあります。もちろんこれはこれで正しいのですが、残念ながら、現状では実用的な PBE システムはほとんどないようです。

PBE システムが思ったほど広まらないのは、以下のような理由からではないでしょうか。

- プログラミングやマクロ定義、カスタマイズには普通のテキスト言語を使ったほうが楽である
  プログラミングに慣れた人はもちろん、エンドユーザーであっても、例示プログラミングの手法を学ぶより、簡単なプログラミングやカスタマイズ方法を勉強したほうが結果的には効率がよさそうです。
- 例からのプログラム作成が難しい
  これは、PBE の本質的な問題です。限られた例を手本として、必要とするプログラムを作成するのはそう簡単ではありません。分野によっては、Internet Scrapbook のようにヒューリスティクスを多用することでうまくいく場合もあります。しかし、あらゆるケースに適用できる推論方式はありません。
- 操作に手間がかかる
  例にもとづくプログラム生成に失敗した場合は、ユーザーがそれに気づいて修正しなければなりません。これは面倒ですし、そもそもシステムの推論方式をよく知らないと、うまく修正できない可能性もあります。

Agilent の Bonnie Nardi は、『A Small Matter of Programming』[3] という著書のなかで、エンドユーザーがカスタマイズの延長線上でプログラミングもおこなえるようなシステムの重要性は認めつつも、例示プログラミングには以下のような欠点があると指摘しています。

- 終了条件や条件分岐が示せない。
- 例として利用する操作が間違いを多く含んでいる場合には混乱してしまう。
- 推論エラーの修正ができない。
- 推論に高いコストがかかる。

さきほど紹介した Stagecast Creator や ToonTalk の場合は、子ども向けのプログラミング教育ツールとしての用途に限定し、そのうえで例示プログラミングの有用性を主張しています。たしかに、これらのツールを使いこなせる子どもなら、いずれは普通のプログラミング言語も扱えるようになる気がしないでもありません。以前、この連載で紹介した LEGO の「MindStorm」でも、子どもも

使えるビジュアル・プログラミング環境が提供されています。とはいえ、Nardiが指摘した点を考えると、汎用的なシステムの実現はそう簡単ではなさそうです。

現状のPBEシステムには解決すべき課題が数多く残されていますが、Internet Scrapbookのように実用化されたものもあり、分野を限定すれば有用なPBE手法はまだまだ考えられるでしょう。たとえば、ユーザーが明示的に〝例〟を与えなくても自動的に取得できるような分野では、システムが勝手に賢くなっていく適応的インターフェイスとして利用することができます。あるいは、ユビキタス計算機環境であれば、人間のあらゆる行動が〝例〟となりうるので、PBEの新たな応用分野が出てくるかもしれません。

おわりに

人間が学習する場合も、〝例〟の利用は効果的です。たとえば、外国語を学ぶには、文法規則をいくら学んでも、それだけで話したり書いたりすることはできません。数多くの文例を読んだり憶えたりすることで、はじめて上達への道が開けます。

応用分野によっては、数多くの例を与えて鍛えた計算機のほうがうまく働く場合も多くなってくるかもしれません。

（ますい・としゆき　ソニーCSL）

[参考文献]

[1] Allen Cypher (ed.), *Watch What I Do — Programming by Demonstration*, The MIT Press, 1993

[2] Henry Lieberman (ed.), *Your Wish is My Command — Programming by Example*, Morgan Kaufmann, 2001

[3] Bonnie A. Nardi, *A Small Matter of Programming*, The MIT Press, 1993

[4] David C. Smith, Allen Cypher and Jim Spohrer, “KidSim: Programming Agents Without A Programming Language”, in *Communications of the ACM*, Vol.37, No.7, pp.55–67, July 1994

[5] Atsushi Sugiura and Yoshiyuki Koseki, “Internet Scrapbook: automating Web browsing tasks by demonstration”, in *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST’98)*, pp.9–18, ACM Press, November 1998

1/2AD スペース
（ノンブル段階で小口寄りに）

リスト 1　smartmake プログラム

```
#!/usr/local/bin/perl
#
#  SmartMake: ヒストリーからMakefileを生成

#  時刻付きヒストリー出力からコマンドリストを作成
$histfile = "$ENV{'HOME'}/.history";
open(h,$histfile) || die "Can't open $histfile\n";
while(<h>){
  chop;
  s/^..//;
  $time = $_ - 0.5;
  chop($_ = <h>);
  $time{"C " . $no++ . " $_"} = $time;
}
close(h);

#  カレント・ディレクトリのファイルの参照・修正情報を取得
opendir(DIR,".");
for $file (readdir(DIR)){
  next if $file eq '.' || $file eq '..';
  @stat = stat($file);
  $time{"A $file"} = $stat[8]; # 参照時刻
  $time{"M $file"} = $stat[9]; # 更新時刻
}
closedir(DIR);

#  コマンド実行とファイル参照を時間順に並べる
@history = sort { $time{$b} <=> $time{$a} } keys %time;

#  コマンドとファイル更新/アクセス履歴からMakefile生成
for (@history){
  if(s/^C \d+ //){
    last if /^(cd|pushd|popd)\s/;
    $command = $_;
    if(! $commands{$command}){
      $commands{$command} = 1;
      @mfiles = keys %mfiles;
      foreach $mfile (@mfiles){
        next if $afiles{$mfile};
        print "$mfile: ";
        for (keys %afiles){
          print "$_ ";
        }
        print "\n\t$command\n";
      }
    }
    %afiles = %mfiles = ();
  }
  elsif(s/^A //){
    $afiles{$_} = 1;
  }
  elsif(s/^M //){
    $mfiles{$_} = 1;
  }
}
```