Pancode Assessed

Dan Jonsson

Dept. of Sociology, University of Göteborg
Brogatan 4, S-413 01 Göteborg, Sweden

– Yet another collection of control constructs?
– Yes, but this one is different...

In his assessment of Pancode control constructs, Kovats [4] seems to argue that there is no need to introduce radically new control constructs, since the traditional constructs, if amended as described by Kovats, have sufficient expressive power.

I feel that Kovats' analysis is somewhat biased in favor of traditional control constructs at the expense of radical alternatives such as Pancode, and to support that claim I shall briefly review some important differences between Pancode control constructs and the set of control constructs proposed by Kovats.

Pancode control constructs

Since it was first described [2], Pancode has been augmented with so-called panstack labels. It has also been refined in other respects. A revised formal syntax is presented below.

The undefined symbols used in this specification of Pancode syntax may be divided into four goups.

◊ Six basic control words: **break**, **repeat**, **when**, **unless**, **also**, and **else**.

◊ Six "statement brackets": (a) <inc> (indentation increment) and <dec> (indentation decrement), two implicit brackets enclosing pans; (b) { and }, two explicit brackets surrounding panstacks regarded as pan units; and (c) **do** and **done**, two brackets enclosing panstack units, optionally used to improve Pancode readability. (The brackets <inc> and <dec> are used to specify the pattern of indentation of program lines used in Pancode as illustrated in the listings below.)

◊ A <panstack label>, i.e., an identifier used as a panstack reference, and a <label terminator>.

◊ Finally, three "host language primitives": <action statement>, <condition>, and <sep> (a statement separator such as a semicolon or a carriage return).

Note that in this revised Pancode syntax, **if** is not a terminal symbol but a macro, separately defined to represent **break unless**.

The Pancode examples presented below should be fairly self-explanatory, so Pancode semantics will not be discussed here.

Pancode syntax

1a	<statement list>	::=	<statement unit> | <statement list> <sep> <statement unit>
1b	<statement unit>	::=	< action statement> | <panstack unit>
2a	<panstack unit>	::=	[**do**] [<panstack label> <label terminator>] <panstack> [<sep> **done**]
2b	<panstack>	::=	<pan unit> | <panstack> <sep> (**also** | **else**) <pan unit>
3a	<pan unit>	::=	<pan> | { <panstack> }
3b	<pan>	::=	<inc> [<head>] [<sep> <statement list>] [<sep> <tail>] <dec>
4a	<head>	::=	**break** [<panstack label>] [<condition clause>]
4b	<tail>	::=	**repeat** [<panstack label>] [<condition clause>]
5	<condition clause>	::=	(**when** | **unless**) <condition>

Control structures proposed by Kovats

The control constructs proposed by Kovats include:

◊ Conventional selection and iteration constructs such as **if... elsif... else... endif**, **while... endwhile**, **repeat... until**, and **loop... endloop**.

◊ The control word **exit** (in Ada) or **break** (in C) used to effectuate an exit from a loop. This exit statement may be deeply embedded in nested **if**-blocks inside the loop.

◊ The innovative control construct **skip** or **fail** (Kovats' term) proposed by Elliott [1]. Within a nest of **if**-blocks, an executed fail command jumps to the lexically nearest following **elsif** or **else** statement in the *same* block or in an *enclosing* block (if such exists – otherwise an error condition results).

Kovats' constructs provide great power of expression in the sense that they make it possible to represent quite complex control structures without using **goto**-statements or resorting to (other) **goto**-patches [3]. They do not necessarily provide equally great power and economy of expression in the sense that algorithms using these constructs represent the logical structure of the underlying problem as simply and clearly as possible, however. I shall try to substantiate this claim by considering a sample problem.

Solutions to Rubin's problem

A problem posed by Rubin [5] has attracted considerable attention and may serve as a tentative benchmark for comparing different control constructs. The problem reads:

"Let X be an N x N matrix of integers. Write a program that will print the number of the first all-zero row of X, if any".

A Pancode solution and a solution in terms of Kovats' constructs are given in Listing 1.

The relationship between Pancode constructs and Kovats' constructs may be clarified by noting that (a) conventional selection and iteration constructs can be represented by means of Pancode constructs [2], and (b) a (labeled) Pancode **break** can mimic both **exit** and **fail**. Consequently, any routine employing Kovats' constructs can be translated on a statement-by-statement basis into an equivalent Pancode routine. For example, the solution of Rubin's problem in terms of Kovats' constructs has a Pancode equivalent, shown in Listing 2.

Listing 2

```
i:=1
do check_row: if true
   if i<=N
      j:=1
      do check_number:
         break check_row when j>N
         /* skip to else matching label 'check_row' */
         break check_number when X[i,j]≠0
         /* quit since no matching else */
         j:=j+1
         repeat
      i:=i+1
      repeat
else
   print('Row ',i)
done
```

Listing 1

Pancode

```
i:=1
do check_row: if i<=N
   j:=1
   do check_number: if j<=N
      if X[i,j]=0
         j:=j+1
         repeat check_number
      i:=i+1
      repeat check_row
   print('Row ',i)
```

Kovats' constructs

```
i:=1;
if true then
    while i<=N
       j:=1;
       loop
          if j>N then fail endif;
          if X[i,j]≠0 then exit endif;
          j:=j+1
       endloop;
       i:=i+1
    endwhile;
else
    print('Row ',i)
endif
```

Comparing the two Pancode solutions, it is easy to see that the first one is superior. The second may even be said to exemplify bad programming style. This style is, however, forced upon the programmer using Kovats' constructs in this case.

From a Pancode perspective, what Kovats proposes is that only a subset of Pancode control structures should be used. This restriction seems to lack a clear rationale, and in certain cases it prevents solutions to programming problems from being expressed as simply and clearly as the problem would permit.

Ad hoc additions to sets of control constructs

One might be tempted to add some new construct to those proposed by Kovats' in order to be able to handle Rubin's problem better. There are also other types of situations which seem to call for additional constructs. Consider, for instance, the example in Listing 3.

The representation in terms of Kovats' constructs is not clean. For example, a redundant '**else** nop' code segment has been added to provide a target for **fail**. A more natural representation would become possible if **fail** was replaced by a control word that transferred control to the statement following the last **endif**.

I am not actually suggesting that Kovats' constructs should be 'improved' as indicated above. The point is that it is possible to suggest additions to Kovats' set of constructs, and then new additions, and there seems to be no way of telling when such a set of constructs is 'complete', so that no additional constructs should be allowed. (It is even possible that if I had given other Pancode examples in the original presentation [2], then Kovats would have come up with some other set of constructs.) While perhaps not a serious problem from a pragmatic point of view, this apparent lack of definiteness is unsatisfactory from a theoretical point of view.

Symmetry and coherence of Pancode constructs

As suggested by the formal syntax presented above, Pancode constructs exhibit several symmetries, including those shown below:

$$\begin{tikzcd} \textbf{break (if)} \arrow[r, leftrightarrow] & \textbf{repeat} \\ \textbf{also} \arrow[r, leftrightarrow] & \textbf{else} \end{tikzcd}$$

$$\begin{tikzcd} \langle \text{inc} \rangle \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & \{ \arrow[d, leftrightarrow] \\ \langle \text{dec} \rangle \arrow[r, leftrightarrow] & \} \end{tikzcd}$$

Break and **repeat** represent forward and backward jumps, respectively. **Also** and **else** are essentially binary operators used to combine two or more pan units into one panstack. { and } are brackets used to distinguish, say, {x **also** y} **else** z from x **also** {y **else** z}. (Left-to-right evaluation is assumed, so that x **also** y **else** z is equal to {x **also** y} **else** z.) Finally, <inc> and <dec> are brackets used to combine panstacks and/or action statements into one pan.

It should be evident from these observations that Pancode constructs constitute a coherent whole. Any additional constructs must preserve or generalize the symmetries and maintain the coherence of the Pancode scheme. It is not permissible to add new constructs in an *ad hoc* manner. This situation contrasts with that described above with respect to Kovats' constructs.

Listing 3

Pancode

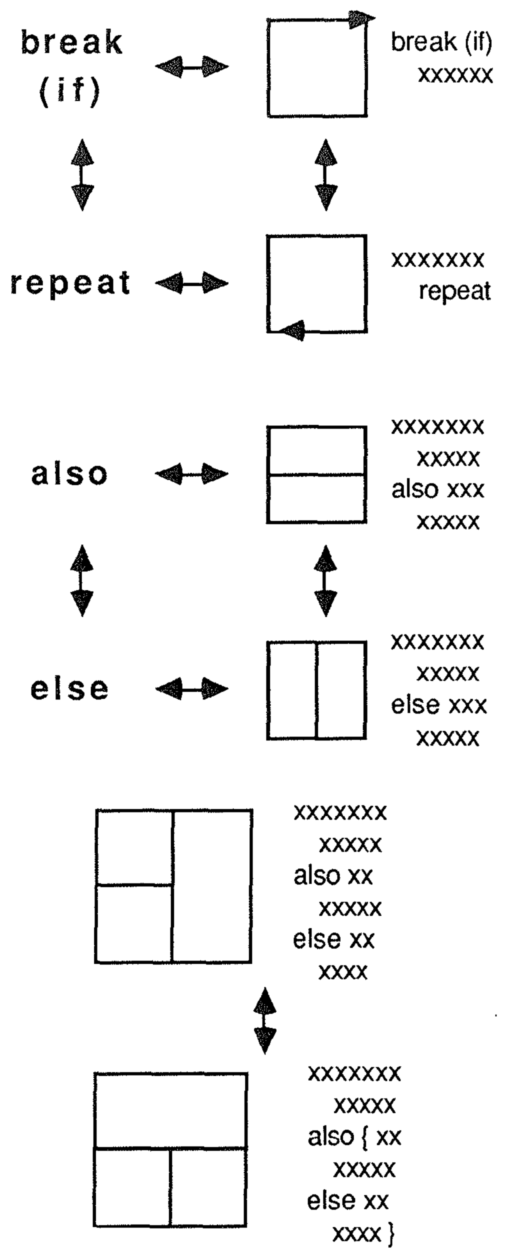
```
do
   t:=0
   read(x)
also if x>0  /* else skip remainder */
   t:=t+1
   read(x)
also if x>1  /* else skip remainder */
   t:=t+1
   read(x)
also if x>2  /* else skip remainder */
   t:=t+1
```

Kovats' constructs

```
t:=0;
read(x);
if x>0 then
    t:=t+1;
    read(x);
    if x<=1 then fail endif;
    t:=t+1;
    read(x);
    if x<=2 then fail endif;
    t:=t+1
else
    nop
endif
```

The Boxchart representation of Pancode

Goto-statements, to be fair, have one important advantage: Programs employing such statements possess natural visual representations in terms of flowcharts. It is desirable to retain this capacity of visual representation when introducing other control structures. How to provide a two-dimensional representation for Kovats' constructs is not clear, however. Pancode, by contrast, has a natural representation in terms of Boxcharts [2]. Pancode and Boxcharts have been developed together and may be regarded almost as two sides of the same coin. As a consequence, syntactical and semantical symmetries present in Pancode are reflected in geometrical symmetries found in Boxcharts, as illustrated below:



Pancode and Boxcharts work together, and to evaluate Pancode constructs without taking their Boxchart representations into account (or vice versa) is like evaluating the performance of one member of a duet without relating it to what the other member is doing. It makes some sense, but does tend to miss the point.

Conclusions

Kovats' constructs probably represent the state of the art in the area of conventional control structures, so they provide a useful reference for a comparative evaluation of Pancode. Kovats argues, quite reasonably, that the introductory examples of Pancode use presented in [2] do not prove that Pancode constructs are superior to Kovats' constructs in terms of power and economy of expression. However, other examples can be given which strongly suggest that Pancode is superior in these respects. In addition, the set of Pancode constructs presented here is more powerful (due to the introduction of panstack labels) than that in [2]. Finally, Kovats does not take certain significant Pancode features into account, specifically the notable symmetry and coherence characterizing its constructs, and the convenient and congenial visual representation of Pancode text in terms of Boxcharts.

References

[1] Elliott I.B. The EPN and ESN Notations. *SIGPLAN Notices*, **19**, 7 (Jul. 1984), 9-17.

[2] Jonsson, D. Pancode and Boxcharts: Structured Programming Revisited. *SIGPLAN Notices*, **22**, 8 (Aug. 1987), 89-98.

[3] Jonsson, D. Next: The Elimination of Goto-Patches? *SIGPLAN Notices*, **24**, 3, (Mar. 1989), 85-92.

[4] Kovats, T.A. Comments on Innovative Control Constructs in Pancode and EPN. *SIGPLAN Notices*, **23**, 12 (Dec. 1988), 151-157.

[5] Rubin, F. "GOTO Considered Harmful" Considered Harmful. *Comm. ACM*, **30**, 3 (Mar. 1988), 195-196.