Coping With Deeply Nested Control Structures

G.R. Perkins R.W. Norman S. Danicic
Department of Mathematics, Statistics, and Computing
Polytechnic of North London
Holloway Road
London N7
U.K.

Abstract

Over the past three or four years SIGPLAN Notices has published a number of papers on the problem of deeply nested IF-THEN-ELSE structures. We maintain that the current editorial ban on proposals for new control structures is correct, not only for reasons of space in SIGPLAN Notices, but on the grounds that programmers (like any other workers) should resist the temptation to blame their tools when they come up with poor products.

Here, we briefly indicate a few techniques for dealing with deeply nested structures, and suggest that working programmers develop other such techniques and publish papers about them, rather than merely ask the language designers for yet more constructs.

Introduction

Several papers and letters have appeared in SIGPLAN Notices in which the problem of the deeply nested IF-THEN-ELSE is defined (usually by giving an abbreviated form of a worst-case nesting) and solutions are proposed - often by introducing new control structures or using existing ones (e.g., FOR loops) in a rather contorted way. We argue that both solution methods are inappropriate and that the real problem is often in the programming technique that led to a deeply nested structure in the first place.

New Control Structures

These suggestions are the easiest ones to deal with. Every non-trivial problem in programming has the potential to give rise to control and data structures not available in the target language. If the target language is a conventional, imperative, algorithmic one (such as Pascal, Cobol, PL/1, Algol 68, Fortran, C, Basic) then there are maybe a dozen structuring mechanisms available (e.g. records, arrays, loops, procedures) and a couple of crude mechanisms for the simulation of unavailable structures (usually goto for control and pointers for data). However these languages derive their power and popularity from their simplicity and generality: adding new structures will mitigate against these strengths.

Obviously languages should not be fixed once and for all, and then allowed to fossilise over the years. But great care should be taken by standards committees to ensure that a new feature is added to a language because it offers additional expressive power to all programmers, not just because someone came up with an example program that seemed to need it.

Cobol is an interesting case in point: many extensions have been added over the years, often with arbitrary syntax. Many of the features are slight variations on each other, and could easily be managed without (who needs paragraphs, sections, and sub-programs?) The most unfortunate aspect of Cobol extensions is that most of them could have gone in as libraries of standard procedures/functions if only a good procedure/function mechanism had been introduced early on.

By contrast many of the recent extensions to Cobol add genuine power for little extra complexity. The EVALUATE statement introduces only two new keywords but allows programmers to write decision tables directly into their code. The general form is

```
EVALUATE exprl   expr2 ... exprn
    WHEN matchl  match2 ... matchn   statement-list
     .        .         .              .
     .        .         .              .
    WHEN matchl  match2 ... matchn   statement-list
END-EVALUATE
```

where each match can be a simple value (WHEN 5 X+l ..), a range (WHEN 5 THRU X Z+l ..), or the default successful match ANY. The single expression version gives you everything you ever wanted from the Pascal case statement but couldn't have.

Existing Control Structures

We cannot condemn the use of existing control structures to resolve deep nesting problems since this is what we are proposing anyway. Equally, we cannot condone the contorted use of existing control structures: some of the published suggestions lead to programs in which control flow is completely opaque. We actually encourage the use of the deeply nested IF-THEN-ELSE under certain circumstances, just as we would encourage the use of the goto.

As an aside we presume everyone realises that deep nesting on the THEN side only is perfectly acceptable (simulates short circuit AND evaluation) as is deep nesting on the ELSE side (simulates the Lisp COND). If deep nesting is still considered ugly then both of these constructs can be flattened out using goto's.

Programming and Coding

We suspect that many programmers end up with deeply nested IF-THEN-ELSE structures because they do not make a sharp enough distinction between the tasks of coding and programming. Coding is the translation of specified algorithms into a target programming language and is only a tiny isolated sub-task of the whole discipline of programming. If a programmer encounters a deeply nested IF-THEN-ELSE during coding it is often a symptom of unclear thinking at the analysis and design stage, and the solution is to scrap the code and start again.

The other point we would like to make is that there is no such thing as the "self-documenting program". Hopefully most professional programmers are aware of this, but in education the student is often led to believe that the use of a good clean language, plenty of comments, and "meaningful" variable names will automatically lead to well-structured programs that can be understood and maintained from the source code alone. This is a terrible fallacy: programs are unmaintainable unless external documentation clearly describes the transformations between problems and solutions, and between solutions and program structures.

Programming Techniques

Obviously we are being fairly critical of programmers who end up with deep nesting problems (or any other opaque data and control structures) at the coding stage. However we are not unsympathetic and would like to suggest a number of useful techniques, all of which are already in use and well understood but unfairly restricted to particular languages or methodologies that utilise them directly.

The first technique to try when confronted with any nasty data or control structure is to go back to analysis and design. Colleagues can often be very helpful here but it is important to communicate the problem to them without constraining their thinking by explaining bits of the failed strategy. We refer readers to the excellent paper by Rosenbloom.

If it does appear that some complicated decisions have to be made by the program then it is important to find a clear way of expressing this in external documentation. There are several powerful notations that can be used:

- EVALUATE statements (even if unavailable in the target language)
- decision tables
- boolean expressions with implicit short-circuiting
- finite state machines
- statement and unit level exits
- recursion and pattern matching

The reader can probably think of more. If an expressive external notational form cannot be found then there is no point in writing the code since it will be unmaintainable: the programmer <u>must</u> return to analysis.

Coding Techniques: EVALUATE and Decision Tables

The first two notations above can be implemented directly in Cobol via EVALUATE and/or level 88's. In other languages the original table should remain as the external documentation and a standard transformation technique (possibly automated) be used to produce source code. It doesn't matter if deep nesting or excessive goto's result since programmers will treat the original table as the source code, and the actual source code as if it were object code. It may be sensible to retain the original table as comments, though installation standards must be developed to keep external documentation, macro source code, actual source code, and comments in step.

Actually we have used VAX extensions to Pascal to write EVAL, WHEN, and END-EVAL procedures/functions which simulate the simple decision table version of the Cobol EVALUATE, e.g.,

```
eval3    (  (x+1)<y,   n>2,   p and not q  );
  if when('   T         F         F         ')  then STMT1;
  if when('   T         F         T         ')  then STMT2;
  if when('   T         -         -         ')  then STMT3;
  if when('   F         T         -         ')  then STMT4;
  if when('   F         -         -         ')  then STMT5;
end_eval;
```

The extensions used are local static variables and default parameter values, though standard Pascal could be used with a little ingenuity. A stack is maintained to allow nested eval's.

Coding Techniques: Short-Circuit Booleans

Short circuit booleans are easily implemented:

IF (a and b) THEN c1 ELSE c2
becomes
IF a THEN IF b THEN c1 ELSE c2 ELSE c2

IF (a or b) THEN c1 ELSE c2
becomes
IF a THEN c1 ELSE IF b THEN c1 ELSE c2

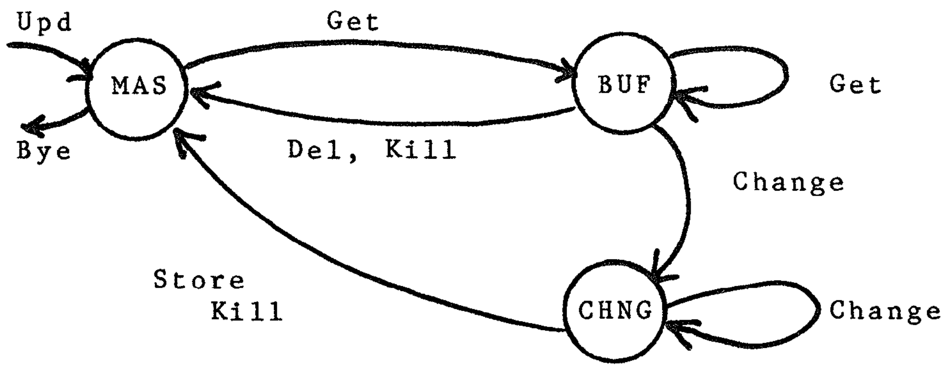
If the ci's are large statement groups rather than single statements or procedure calls, then goto's can be used to avoid excessive source code repetition. Compound expressions merely require recursive applications of the same rule, which is easy with a macro-processor. Again the resulting code should never be read or maintained - the programmer should maintain the external notation and then re-generate the code. Many awkward control structures arising from the need to avoid array, pointer, or file access violations can be modelled as short-circuit booleans.

Coding Techniques: Finite State Machines

We have discovered that Finite State Machines offer an extremely powerful method of coping with all sorts of problems. They almost always eliminate structures which would have contained IF's and WHILE's nested four or five deep. So far we have used FSM's in three major application areas, which we will describe briefly.

Many text processing problems can be reduced to some form of lexical analysis. In these cases it is convenient to express input strings as regular expressions, then convert to FSM's and associate program actions with transitions. As an example our first year students used a 2-state 4-transition FSM to split a text file into its component words. The resulting program was extremely simple and short, with virtually no nesting of control structures. It was then used as the first filter in a series of simple programs and O.S. commands that together provided some quite powerful text processing.

FSM's can also be used to model the sequencing aspects of interactive dialogues. The following FSM models the legal command sequences during a simple master-file update session:



This simple machine expresses a number of rules about the order in which commands are applicable:

- Exit from system is only possible when it is in the same configuration as when it was entered

- A record can only be changed if it has been Got into a working buffer area

- A record cannot be deleted if it has been changed

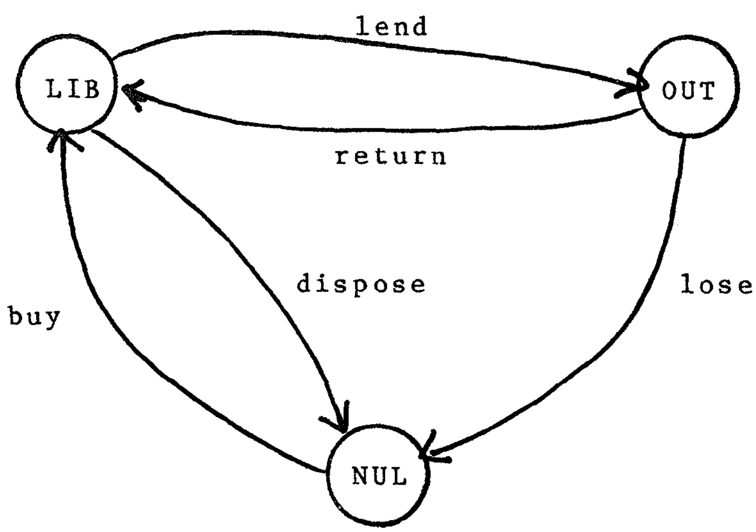
- A record cannot be stored unless it has been changed

- Once a record has been changed, the user must explicitly Kill or Store it before dealing with another record

A programmer reading the above rules would probably end up writing a program which contained a number of status variables and a complicated IF/WHILE structure to ensure that the rules weren't broken. With luck, the program would be a precise simulation of the FSM we defined above. However, it is more likely that the value space for the status variables would be much larger than the three states required by the FSM model, thus making condition evaluation much more complicated than is necessary.

The other application for FSM's is in the area of process-oriented systems analysis, of which the Jackson System Design methodology is an example. Here one is concerned with the sequential behaviour of objects in a system, rather than static relationships between data objects as in more conventional database or file processing oriented methodologies. Every object in the system (including interfacing objects such as users) is modelled as a sequential process with the ability to send and receive messages in order to report or update its current status. The above example of interactive dialogue control is a special case of this modelling technique in which only one aspect of a system (namely user interaction with master-file) has been modelled as a sequential process.

We have found that the JSD "entity structure diagrams" can be dispensed with since the FSM is easier to construct and read; the following machine describes the life-cycle of a library book:



In all three applications of FSM's discussed above, implementation can be effected without resorting to complex control structures. Each FSM can be represented by its corresponding transition table, which can be kept on file and loaded into run-time data structures at the start of each program run. A simple interpreter can be written which merely obtains the next message and looks in the table to fire the appropriate transition and select the appropriate action code with a CASE construct. The interpreter (or FSM simulator) looks something like:

```
state := startstate
repeat
  getmessage(mcode)
  <action, state> := transtable[ state, mcode ]
  service(action)    (* procedure to select service code *)
until state in haltstates
```

where

```
procedure service( action:integer )
  case action of
     1:
     2:
     ..
     ..
  end
end service
```

Note the paucity of control structure nesting! In the library book example above each book would be represented by a record which contained its current state as well as its usual data. If a system contains several FSM's then a global message handler is needed to select the correct transition table for the simulator to use. The complicated "inversion" technique of the implementation stage of JSD does not use transition tables, but leaves all the sequencing control in the actual code. Using FSM's at this stage is much easier, and keeps all sequencing control in the transition table on a data file (thus simplifying program maintenance).

Coding Techniques: Exits

Both statement level and unit level exits can be extremely useful ways of simplifying control structures. Many programmers are reluctant to use such exits as they have misinterpreted (or more likely, have been instructed by people who have misinterpreted) the "goto considered harmful" arguments. But since many languages provide both types of exit there is no reason why programmers should not use such structures in their program specifications, so long as they have the discipline to use or design installation standards for their simulation.

Typical statement level exits provided directly in various languages are: Cobol's SEARCH verb with its WHEN conditions, DEC-10 Pascal's LOOP - EXIT IF structure, C's BREAK and CONTINUE statements, and Ada's EXIT WHEN statement. These, and variations on them, can be cleanly simulated with the goto statement (the single-entry single-exit rule does not need to be broken).

The most common unit level exit is the RETURN statement provided by many langauges, with the notable exception of Pascal. We have shown students RETURN can easily be simulated:

```
function f( x:real ) : real;
  label 99;

     procedure return(ret:real);
       begin  f:=ret; goto 99  end;

  begin
     ..
     if .. then return(sqrt(x));
     ..
  99: end;
```

More sophisticated unit level exits can be utilised to handle errors detected by the program at run-time. The EXCEPTION raising and handling facility of Ada can easily be simulated in Pascal:

```
procedure p;

type exception = (null, overflow, underflow, zerodiv);
var  cond : exception;

     procedure raise( raisecond:exception );
       begin  cond:=raisecond; goto 90 end;

begin
  cond := null;
  ..
  if n=0 then raise(zerodiv)
  ..
90: case cond of
      null:;
      overflow:  ..
      underflow: ..
      zerodiv:   ..
      end
end; (* procedure p *)
```

Actually this particular strategy only allows an exception to be trapped by the closest enclosing block containing exception declarations, since "raise" refers to the closest enclosing declaration of a "raise" procedure. As part of a Pascal course for postgraduate students we have discussed more useful error trapping strategies which, for example, have all error codes, messages, and trapping information stored in an easily modifiable text file.

Coding Techniques: Recursion and Pattern Matching

Recursion is an extremely powerful programming technique but is commonly thought to be inefficient in comparison with iteration. Nevertheless several languages (Lisp, Hope, Miranda) provide recursion as the principal structuring mechanism and programmers using these languages are quite happy to write all their programs in terms of mutually recursive functions. The run-time inefficiency of such programs is due to inappropriate computer architectures, garbage collection overhead, and the use of interpreters or non-optimising compilers. The recursive functional style itself is not at fault.

Our experience with this technique has been in the teaching of a BSc Data Structures unit which, as in many other educational establishments, forms the core of the Computer Science curriculum (after the very introductory programming and information representation units). Lack of space prohibits a full discussion here, suffice it to say that we made liberal use of a simplified version of Z, the data type specification method developed by Bernard Sufrin and others in the Oxford Programming Research Group.

We found in general that specifications and implementations of data types were extremely short and simple, and that standard methods could be used to transform specifications into Pascal type declarations and functions. Pattern matching, such as

```
length(emptylist) = 0
length( a :: alist )  = 1 + length(alist)
```

turned out to be a nice specification technique that could be implemented by using an extra IF in the function definition. Pointers (and the use of NEW) practically disappeared. For example, the function to insert an item into a sorted list has no pointer references at all and consists of an IF statement nested two deep on the else side only. By contrast, the "insert" procedure found in standard textbooks has two local pointer variables, contains numerous pointer references, is about twenty lines long, and usually contains IF's and WHILE's nested five deep (some of the IF's nested on both sides). The same elimination of deeply nested control structures was encountered with all our data structures.

The power of these techniques allowed us to cover applications involving list of trees of records with no more dificulty than lists of characters. Some students had trouble understanding recursion early on in the unit, but nobody got tangled up with deeply nested control or data structures.

For those concerned with space-time efficiency we would point out that most of our functions involved tail recursion only, which can easily be flattened out. Even the need for garbage collection can be reduced by using "replace" functions (e.g., the Lisp rplaca and rplacd) which have side-effects but do not spoil the functional style too much. The "mark" stage of garbage collection can also be eliminated by careful recording of information in the "cons" functions.

Conclusions

Programming is clearly a difficult task but the stream of papers on the deep nesting problem seems to indicate that many people are pushing their difficulties down into the coding phase without realising that they have only partially solved a difficult problem at a higher level. The way in which Structured Programming is covered in many books is partly to blame for giving people a false sense of security. We have discovered that many students can get full marks on an exam question which says "Describe what is meant by structured programming and explain how it eases the tasks of program writers and maintainers" and yet very few can apply structure to a design before turning it into code. We have yet to find a text book that even explains how to develop a structured naming scheme for variables, though they all exhort us to use meaningful variable names!

There is also an unrealistic expectation that programming languages should provide solutions to everyone's coding problems in a "stand-alone" fashion. This cannot be the case (witness Cobol and PL/1) and it must be accepted that external documentation and standard transformation schemes form an integral part of any program, as well disciplined programmers in good installations know full well.

Our improvement over deeply nested "improvement over..." papers is that we should stop trying to define, generalise, and "solve" awkward lumps of syntax. Instead we should publish papers on the problem spaces we find and the modelling techniques and transformation methods we develop. Language designers should look carefully at the notational structures that arise from these developments and see if any of them are required often enough to warrant them being turned into new languages or upgrades to existing languages. In other words language design should be "problem driven" rather than "code driven".

Bibliography

The titles get longer and longer so we just give the authors and SIGPLAN Notices references (in date order).

Hill, G.P.	V17 #8, August 1982	
Lau, D.	V18 #3, March 1983	(correspondence)
Marks, R.E.	V18 #3, March 1983	(correspondence)
den Hertog, E.H. Gerbscheid, H.J.C. Kersten, M.L.	V18 #3, March 1983	(correspondence)
Hill, G.P.	V18 #4, April 1983	(correspondence)
Lakhotia, A.	V18 #5, May 1983	
Taylor, D.	V18 #10, October 1983	
Rosenbloom, M.H.	V18 #10, October 1983	
Amit, N.	V19 #1, January 1984	
Nelson, D.F.	V19 #2, February 1984	(correspondence)
Amit, N.	V19 #4, April 1984	(erratum to above)
Baldwin, R.R.	V20 #10, October 1985	
Baldwin, R.R.	V21 #9, September 1986	