[10] ZISLIS, P. M., An Experiment in Algorithm Implementation, CSD TR 96, Purdue University, Dept. of Computer Science, June 1973.

[11] ZWEBEN, S. H., Software Physics : Resolution of an Ambiguity in the Counting Procedure, CSD TR 93, Purdue University, Dept. of Computer Science, April, 1973.

AN UNSTRUCTURED VIEW OF STRUCTURED PROGRAMMING

Richard H. Karpinski

Office of Information Systems
University of California, San Francisco

Several complementary notions dealing with writing computer programs have been introduced in the last few years. The first of these to cause comment was the idea of avoiding the GO-TO instruction. While considerable debate has ensued, the finest spokeman both for and against have agreed that the unrestrained use of the GO-TO should be discouraged.

After further consideration, it is becoming clear that the GO-TO question is a minor issue. The major issue is, of course, how to write better programs. Let us then examine the various aspects of better programs.

Better programs should run quickly and take little space. Since computing resources have always been limited (not small necessarily, but never limitless) these aspects have always been recognized. Many questionable programming practices have been justified on these grounds. Since analysis of a program's use of system resources is (increasingly) convenient, and since such analysis does, in practice, yield surprising results, the important consideration becomes the ease with which the program can be modified to employ new strategies.

Better programs should be easy to write and debug. Again, these attributes of good programs have been long sought. The problem oriented languages have attempted to promote better programs by assuming some of the clerical tasks and by detecting certain classes of errors. The continuing proliferation of such languages suggests that an unrecognized problem may exist. Of this, more later.

Better programs should be easy to understand and to modify. These aspects have been recognized mostly by sophisticated programming managers and by unfortunate programmers who have been asked to change someone's program. Even today, many of us tend to blame the damn user for arbitrarily changing specifications, while taking scant heed of that likehood in creating the program.

We shall be concerned then with how to write programs which are easy to write, debug, understand, and modify. The key point is understanding. This is the end to which NO-GO-TO advocates aspire. They argue that more limited control structures such as DO-WHILE and IF-THEN-ELSE are easier to understand than GO-TO (which does not

announce its purpose). Dijkstra points to complexity as the major issue. (See references 2 & 3.) He notes that digital computers require perfection in many aspects of their programs, but his point is that perfection is achievable. The key to perfection is seen to be the control of complexity.

This brings us to structured programming. Formally, a program which uses only control structures from some collection (S) is said to be S-structured. As such, structured programming is not very interesting. However, it has been observed that adherence to certain principles in the choice of control mechanisms leads to the development of clean, tidy, correct programs. Briefly, the constraint is to use blocks of code which are either nested (DO-WHILE & IF-THEN-ELSE) or sequential (A-then-B) and have one entrance and one destination, although perhaps several exits to it. That destination is always the next item in the containing block. The point of the constraint is to ensure that one need not understand the code in a lower block to follow the program flow of control. Notice that this does not require elimination of the GO-TO but constrains the target to lie in the same block as the GO-TO.

Within the confines of structured programming, it becomes sensible to decompose the problem, and hence the program, from the outside in (or from the top down). Wirth(7) and Dijkstra(2) discuss this way of approaching the programming task. Essentially, the tree-like (or outline-form, i.e. indented, if you prefer) program which results from top-down decomposition, is shown to reduce the complexity of the program locally. Two parts of such a program may communicate only through data established at a higher point in the tree (a more major item in the outline) than either user.

Use of the techniques mentioned so far will lead one to write programs which are easy to understand. several problems remain. Chief among these is that of making programs that are easy to modify. Of course, any program which can be understood can be modified. However, in practice, care should be taken that likely changes will each be confined to a small part of the program. To do this, one must consider probable changes while making the initial decomposition of the problem, and hence of the program. Parnas(6) discusses criteria for making such decompositions. The principle involved is called information hiding. The point is to isolate major decisions (which may, from time to time, need revision) within their own modules. This permits their revision without changing other modules.

In reading about these principles, I have been lead to notice that many people assume that the module of modular-programming and the block of structured-programming both correspond to a separately compilable procedure. This, in turn, leads them to assume modules ranging from say thirty lines to upper limits of one page (fifty lines), a few hundred lines, or even thousands of lines of code, in some cases. I would consider, however, that the call-procedure boundary is yet another control structure which may be employed for various purposes at various points within the overall tree structure of a structured program. This view permits sensible use of very small blocks. As a rule of thumb, I view with suspicion any blocks larger than ten lines, or containing more than one IF-THEN-ELSE, loop, or GO-TO.

Using these very small blocks containing references to named blocks proved to be an easy way to write programs. In general, I succeeded in using only data defined in this block or in a containing block.

Programs so constructed proved easy to modify. In fact, only two difficulties were encountered: the clerical task of creating a compilable procedure from such blocks is tedious and error prone and, secondly, I still left off some end statements. To solve these problems, I have developed a program (STRUCTR) which performs this construction (producing a PL/I-style source stream to be compiled, with PL/I comments replacing the references to nested blocks and indentation corresponding to nesting depth). An example of the input and output of this program is included below.

STRUCTR provides an argument mechanism which permits one to develop, for example, an "if" block or a "do-while" block, which need only be corrected once in case of a missing end.

```
==$if(test,$then,$else);
if test
then do /* (test) is true */;
success, act accordingly =:$then;
end /* (test) was true */;
else do /* (test) is false*/;
failure, act accordingly =:$else;
end /* (test) was false */;
```

In this case, "test", "$then", and "$else" are to be passed in each reference to $if. Whether this mechanism will prove sufficient is not yet clear. In any case, the use of automatic transformation of individual blocks into a usable source stream permits the maintenance of both forms of such a program. This appears to provide a substantial improvement in ease of program modification. Successful use of top-down, structured, program decomposition guarantees that any "hidden" block can be replaced with impunity. This emphasis on maintenance may seem strange until one notes that maintenance begins with the first change to a previously written line of code. Thus I find, for example, new freedom to change a design decision as soon as problems with that decision become apparent.

The construction mechanism, STRUCTR, is basically a line oriented system for representing structured programs. That is, it will provide the following facilities:

1. Iterative Refinement (outside-in or top-down programming)

   refine ::= intent '=:' name ';'

   The "refine" will appear as a comment followed by the text of the module called name, appropriately indented.

   This permits the use of very small modules which are built up without any necessary linkage overhead. These small modules, in turn, permit capturing the decisions made in designing the program in their proper order. This tends, in appropriate use, to show directly the approach to writing programs suggested by Dijkstra & Wirth.

2. Structured Programming

   control_refine ::= intent '=:$' name';'

This form of a "refine" makes use of a pre-coded control structure such as a do-while, do-index, do-case, or if.

Each control structure may be used only when the base language in use supports that structure. However, note that a great deal of a program can thus be written in a language independent fashion. Exclusive use of some such set of system modules for control functions constitutes a clear example of structured programming. (It would be possible to enforce that restriction, but perhaps at the cost of reserved words.).

In general, a control structure will require one or more arguments to specify the details or to name the modules to be included within that structure.

3. Unexpected Arguments to Contained Modules

argument ::= name ':=' value ';'

The argument will be available to all descendents of this module. That is, an appearance of name as an identifier within a contained module (one referred to in a "refine") will be replaced by value. One exception is that an "intent" will not be altered in this way.

4. Information Hiding

hidden_refine ::= intent '=:' name ';*'

Such a module and its offspring are said to be hidden. Ideally, the interior of such modules would not be listed or would be obscured, however, among the compilers at hand, only PL/I & assembler support such literal hiding. Each such module is expected to be associated with a documentation and declarations module to be included (via "shared_refine") at some appropriate point in the program, anteceeding all uses of the hidden module.

5. Information Sharing

shared_refine::= intent '=:' name ';$'

This form of a "refine" makes the named module and its offspring available for use by the other descendents of this module (the one containing the "shared_refine"). The usual indentation to show refinement is suppressed to indicate that status, i.e. to indicate the logical position of the module within the tree structure of the program.

```
/* SAMPLE PROGRAM TO STORE AND RETRIEVE DATA EFFICIENTLY =:$MAIN */                001
   CRAM:PROCEDURE OPTIONS(MAIN);                                                   002
   /* INCLUDE THE BODY=:CRAM */                                                    003
      /* ESTABLISH LINE STORAGE FACILITY =:LINE_STORE */                           004
         /* ESTABLISH SHARED DATA =:LINE_SET_UP */                                 005
            /*LINE STORAGE FACILITY */                                           006
            /* TO CHANGE NAMES OR SIZES:     */                                   007
            /*   LINE:=NEW_NAME_FOR_LINE; */                                      008
            /*   250:=NEW_LINE_SIZE;   */                                         009
            /*   10000:=NEW_LINE_AREA_SIZE; */                                    010
            DCL EMPTIED CONDITION;                                                011
            DCL FILLED CONDITION;                                                 012
            DCL LINE CHAR(250) VAR;                                               013
            DCL NULL BUILTIN;                                                     014
                                                                                  015
         /* DESCRIBE THE FACILITY =:LINE_STORE_DESC */                            016
            /*TO STORE "LINE" AWAY AND MAKE IT THE CURRENT LINE */               017
            /*   (IN CASE THERE IS INSUFFICIENT ROOM, */                         018
            /*    "FILLED" WILL BE SIGNALLED.)       */                         019
            /*   STORE_LINE TO STORE AFTER THE LAST LINE AND MAKE CURRENT */     020
                                                                                  021
            /*TO RETRIEVE A LINE MAKE IT THE CURRENT LINE & SET "LINE" TO IT */  022
            /*   (IN CASE THE LINE DOES NOT EXIST, "LINE" WILL BE SET TO */      023
            /*      THE NULL STRING, AND "EMPTIED" WILL BE SIGNALLED.) */        024
            /*   GET_NEXT TO GET AND MAKE CURRENT THE LINE AFTER CURRENT */      025
            /*            (IF BOTH EXIST, OTHERWISE SIGNAL EMPTIED) */          026
            /*   GET_PREV TO GET AND MAKE CURRENT THE LINE BEFORE CURRENT */     027
            /*            (IF BOTH EXIST, OTHERWISE SIGNAL EMPTIED) */          028
            /*   GET_CURRENT TO GET THE CURRENT LINE AGAIN, IF ANY */            029
            /*   GET_LAST TO GET AND MAKE CURRENT THE LAST LINE, IF ANY */       030
            /*   GET_FIRST TO GET AND MAKE CURRENT THE FIRST LINE, IF ANY */     031
                                                                                  032
         /* DECLARE REQUIRED DATA STRUCTURES =:LINE_DATA */                       033
            DCL LINE_AREA AREA(LINE_ASIZ);                                        034
            DCL 1 LINE_INST BASED(LINE_CURR),                                     035
            2 LINE_PREV OFFSET(LINE_AREA) INIT(LINE_LAST),                        036
            2 LINE_NEXT OFFSET(LINE_AREA) INIT(NULL),                             037
            2 LINE_LEN FIXED BIN(15),                                             038
            2 LINE_VAL CHAR(LENGTH(LINE) REFER (LINE_LEN)) INIT(LINE);            039
            DCL (LINE_CURR, LINE_LAST, LINE_FIRST) OFFSET(LINE_AREA) INIT(NULL);  040
                                                                                  041
      /* CYCLE FILLING LINE & STORING IT =:STORE_LINES */                         042
         /* PASS THE INPUT FILE =:$PASS_FILE */                                   043
            DCL SYSIN FILE RECORD;                                                044
            OPEN FILE(SYSIN) INPUT SEQL;                                          045
            ON ENDFILE(SYSIN) GO TO $FILE_ENDED;                                  046
            /* CYCLE THROUGH THE FILE=:$CYCLE */                                  047
               DO WHILE('1'B);                                                    048
               /* DO WHAT IS REQUESTED=:$PASS_READ */                             049
                  READ FILE(SYSIN) INTO(LINE);                                    050
                  /* PROCESS THIS RECORD=:STORE_LINE */                           051
                     ON AREA SIGNAL CONDITION(FILLED);                            052
                     ALLOCATE LINE_INST;                                          053
                     IF LINE_FIRST = NULL                                         054
                     THEN LINE_FIRST = LINE_CURR;                                 055
                     ELSE LINE_LAST->LINE_NEXT = LINE_CURR;                       056
                     LINE_LAST=LINE_CURR;                                         057
               END /* WHILE('1'B) */;                                            058
            $FILE_ENDED:                                                          059
            CLOSE FILE(SYSIN);                                                    060
```

```
   /* CYCLE RETRIEVING LINES & PRINTING THEM =:PRINT_LINES */                    079
      /* GET THE LINE_FIRST LINE =:GET_FIRST */                                  080
         LINE_CURR=LINE_FIRST;                                                   081
         /* FILL IN THE LINE =:GET_CURRENT */                                    082
            IF LINE_CURR =NULL                                                   083
            THEN DO;                                                             085
            LINE = '';                                                           087
            SIGNAL CONDITION(EMPTIED);                                           089
            END;                                                                 091
            ELSE LINE=LINE_VAL;                                                  093
      /* LOOP PRINTING LINE AND GETTING NEXT =:$DO_WHILE */                      098
         DO WHILE(LENGTH(LINE)>0);                                               099
         /* DO WHAT IS REQUESTED=:PRINT_LINE */                                  101
            DCL PRINT FILE RECORD OUTPUT;                                        102
            WRITE FILE(PRINT) FROM (LINE);                                       104
            /* GET THE NEXT LINE =:GET_NEXT */                                   106
               IF LINE_CURR ^=NULL                                               107
               THEN LINE_CURR=LINE_NEXT;                                         109
               /* FILL IN THE LINE =:GET_CURRENT */                              111
                  IF LINE_CURR =NULL                                             112
                  THEN DO;                                                       114
                  LINE = '';                                                     116
                  SIGNAL CONDITION(EMPTIED);                                     118
                  END;                                                           120
                  ELSE LINE=LINE_VAL;                                            122
         END /* WHILE(LENGTH(LINE)>0) */;                                       130
END /* PROCEDURE CRAM */;                                                       135
```

```
==prog;
sample program to store variable data efficiently =:$main(cram);

==cram;
establish line storage facility =:line_store;$
cycle filling line & storing it =:store_lines;
cycle retrieving lines & printing them =:print_lines;

==store_lines;
pass the input file =:$pass_file(sysin,into(line),store_line);

==print_lines;
get the line_first line =:get_first;
loop printing line and getting next =:$do_while(length(line)>0,print_line);

==print_line;
dcl print file record output;
write file(print) from (line);
get the next line =:get_next;

==line_store;
establish shared data =:line_set_up;
describe the facility =:line_store_desc;
declare required data structures =:line_data;*

==line_set_up;
 /*line storage facility */
 /* to change names or sizes:     */
 /*   line:=new_name_for_line; */
 /*   line_size:=new_line_size;   */
 /*   line_asiz:=new_line_area_size; */
```

```
dcl emptied condition;
dcl filled condition;
dcl line char(line_size) var;
dcl null builtin;
line_size:=?250;
line_asiz:=?10000;

==line_store_desc;
 /*to store "line" away and make it the current line */
 /*   (in case there is insufficient room, */
 /*    "filled" will be signalled.)        */
 /*   store_line to store after the last line and make current */

 /*to retrieve a line make it the current line & set "line" to it */
 /*   (in case the line does not exist, "line" will be set to */
 /*      the null string, and "emptied" will be signalled.) */
 /*   get_next to get and make current the line after current */
 /*            (if both exist, otherwise signal emptied) */
 /*   get_prev to get and make current the line before current */
 /*            (if both exist, otherwise signal emptied) */
 /*   get_current to get the current line again, if any */
 /*   get_last to get and make current the last line, if any */
 /*   get_first to get and make current the first line, if any */

==line_data;
dcl line_area area(line_asiz);
dcl 1 line_inst based(line_curr),
2 line_prev offset(line_area) init(line_last),
2 line_next offset(line_area) init(null),
2 line_len fixed bin(15),
2 line_val char(length(line) refer (line_len)) init(line);
dcl (line_curr, line_last, line_first) offset(line_area) init(null);

==store_line;
on area signal condition(filled);
allocate line_inst;
if line_first = null
then line_first = line_curr;
else line_last ->line_next =  line_curr;
line_last=line_curr;

==get_next;
if line_curr ¬=null
then line_curr=line_next;
fill in the line =:get_current;

==get_prev;
if line_curr ¬= null
then line_curr = line_prev;
fill in the line =: get_current;

==get_current;
if line_curr =null
then do;
line = '';
signal condition(emptied);
end;
else line=line_val;
```

```
==get_last;
line_curr=line_last;
fill in the line =:get_current;

==get_first;
line_curr=line_first;
fill in the line =:get_current;

==$pass_file($file,how,$then);
dcl $file file record;
open file($file) input seql;
on endfile($file) go to $file_ended;
cycle through the file=:$cycle($pass_read(how,$then));
$file_ended:
close file($file);

==$pass_read(how,process);
read file($file) how;
process this record=:process;

==$cycle($do);
do while(1b);
do what is requested=:$do;
end /* while(1b) */;

==$do_while(test,$do);
do while(test);
do what is requested=:$do;
end /* while(test) */;

==$main(name);
name:procedure options(main);
include the body=:name;
end /* procedure name */;
```

REFERENCES:

1. Baker, F. T. "Chief Programmer Teams" IBM Systems Journal, Vol. 11, No. 1, 1972

2. Dijkstra, E. W. "Notes on Structured Programming" (1969) Technische Hogeschool, Eindhoven, Netherlands

3. "The Humble Programmer", Turing Lecture Communications of the ACM, October, 1972

4. Leavenworth, B. M. (ed) "Control Structures in Programming Languages" SigPlan Notices, Vol. 7, No. 11, November 1972

5. Mills, Harlan "Mathematical Foundations For Structured Programming"
IBM FSD, Gaithersburg, Maryland 20760

6. Parnas, D. L. "On The Criteria To Be Used In Decomposing Systems Into Modules"
Communications Of The ACM, December 1972

7. Wirth, N. "Program Development By Stepwise Refinement"
Communications Of The ACM, April, 1971

A more extensive bibliography is available.
Write to:

Richard Karpinski
76-U Information Systems
University of California
San Francisco, Calif. 94122

To the Editor:

ERRATUM to

"A Programming Language for Mini-Computer Systems," by Frank L. Friedman and Victor B. Schneider, SIGPLAN Notices (9, 1) :

On page 18 (in the section entitled Group Variables), the line

<group declaration> :: =

should have been followed by the line

group <group name> [<component list>] <group variable list>

This line was inadvertently ommitted from the text.

Frank Friedman