Check for updates

Goal Directed Programming

by Richard J. and Martha J. Cichelli
901 Whittier Drive
Allentown, Penn. 18103

To many experienced programmers the current commotion over structured programming seems, at best, overdone. Structured programming is frequently presented in the commercial literature as a set of coding rules. Most of these rules begin with the word "don't". The SP messiahs add insult to the controversy by deprecating what were thought to be well designed programs with finger pointing - "GOTO's! Tsk, tsk." If there is practical validity in the structured programming philosophy, its rules must be reformulated positively in terms of "do's". The resolution of the matter lies in satisfying the request, "Show me how to do it better."

"Goal directed programming" (GDP) is a positive, consistent, integrated design philosophy which incorporates both structured programming and top-down design principles. For the experienced programmer it is a guide to proper design; for those who have never written a line of code it offers a way to understand and participate in program design. This article is an introduction to goal directed programming.

The programmer's job is to design and formally specify an algorithm to accomplish some task. The resultant algorithm will be composed of primitive operations on data called function code (e.g. MOVE's in COBOL, assignment statements in FORTRAN) and control code which determines whether, or how may times, function code is executed (IF's, PERFORM's, DO-loops, GOTO's, etc.)

Structured programming theory tells us that the control code should be formulated from D-structures (Dijkstra's sequence, if-then-else, and while-do). Assume that we have a language with these and a few extensions (case, repeat-until, for, etc.) Are they sufficient for good programming? If so, how do we get from the problem specification to a proper program solution? We need a problem solving technique which transforms problem specifications into program code.

Let us say we wish to copy a sentence. Defining "sentence" as a sequence of characters ending with a period, a structured program in a PL/Esperanto with D-structures to do this simple task might look like:

```
REPEAT
    CALL READCH(CHAR)
    CALL WRITECH(CHAR)
UNTIL (CHAR = PERIOD)
```

How do we get from the goal "copy a sentence" to the code? Since we know that a sentence is a sequence of items, we need repetition

to process the items. Every iteration needs a termination (otherwise we loop forever!)

The design process in outline is:

Goal	Copy a sentence
Assertion	When the sentence has been copied, the period will have been processed
Condition	(CHAR = PERIOD)

From the goal we derive an assertion that is affirmed when the goal is achieved. From the assertion we derive a logical condition that is true when the assertion becomes true. This logical condition forms the termination condition of our iteration code. At this point in the design the program would look like:

```
REPEAT
    copy a character
UNTIL (CHAR = PERIOD)
```

"Copy a character" can then be elaborated in place into the more primitive

```
CALL READCH(CHAR)
CALL WRITECH(CHAR).
```

How does GDP lead to the top-down design of structured programs? GDP is simply an application of general problem solving techniques to programming. In any problem solving situation large problems are factored into smaller, more manageable ones. At any level of problem decomposition certain sub-problems may be solved by the available primitives while others must be further factored into sub-goals and their corresponding sub-problems. GDP design is an elaboration of the problem solving process; it tends to be top-down because human problem solving tends to be top-down.

Since the top level code is user problem oriented, user participation during the crucial early stages of system design is facilitated. Design and coding take place simultaneously. Traditional flowcharts, at best a poor design tool, can be eliminated.

Goal directed programs develop with an intuitive and natural structuring. Each goal is expanded in place. Since no goal is satisfied by "going" somewhere else, there is no temptation to create spaghetti-like control code. Control code is separated both logically and syntactically from function code; it is created immediately from the logical conditions derived from the goals and is coded before the function code implementing the goals. Of course, the function code may itself contain control code at a lower level. Proper nesting of control structures is insured by the top-down order of the design-coding process.

What about the design of data structures? For every goal-problem we can apply the goal-assertion-condition transformation. If we start with the overall problem statement as our first goal, these transformations will lead directly to a top-down design of the program. The same top-down process should be applied to designing a problem's data structures.

It is evident that proceeding from the most general statement of the problem leads directly to treating the most general or inclusive data structures first. To process large compound data structures we decompose them into smaller components and then process the components. Processing a mailing list file, for example, requires the decomposition of the file into records and the records into name and address lines, etc.

There is a direct correspondence between D-structures and data structures. Records are made up of sequences of elementary data items and of structures. These structures may contain iterated parts or alternative component parts. Data structures are most appropriately processed by their corresponding D-structure. Code sequences or blocks process aggregates of items of different types, WHILE and REPEAT loops process sequences of items of the same type, and CASE and IF selectors process alternative parts.

Goal directed programming is most easily illustrated by an example. Edsgar Dijkstra posed the following programming problem:

You are given two subroutines - READCH which reads a single character and WRITECH which writes a character. Given an input sentence, write a copy of it such that 1) extra blanks are eliminated and 2) every other word is written backwards. The input sentence

this is a silly program

would be written

this si a yllis program.

(Because of this example sentence, the program became known as "The Silly Program". See "Notes on Structured Programming" in Structured Programming by Dahl, Dijkstra, and Hoare - Academic Press).

To simplify the program, sentences are defined as being one or more words, each word separated by one or more blanks. Sentences end with a period, and the maximum word length is twenty characters.

The solution to this problem will be presented in SCOBOL (an extended form of COBOL developed by Lars Mossberg of VOLVO Flygmotor and implemented with a precompiler).

Earlier in this paper we wrote a program to copy a sentence.

```
REPEAT
    CALL 'READCH' USING CHAR.
    CALL 'WRITECH' USING CHAR.
UNTIL (CHAR = PERIOD)
```

The next refinement of the program will copy the sentence and eliminate extraneous blanks. The final version will reverse alternate words.

In the second version of our program our new goal is to eliminate extra blanks. To do this we must first ignore any blanks which precede the first word. Extra blanks between words must also be discarded. Note that we cannot write a blank immediately after copying a word; we must first find the next non-blank character because we do not want to write a blank before the ending period.

```
NOTE - GOAL = SKIP LEADING BLANKS.
REPEAT
    CALL 'READCH' USING CHAR.
UNTIL (CHAR NOT = SPACE)

NOTE - GOAL = COPY REST OF SENTENCE.
REPEAT
    copy rest of sentence
UNTIL (CHAR = PERIOD)
```

The goal "copy rest of sentence" can be decomposed into the three sub-goals "copy a word", "get the next non-blank character", and "write either a period or a blank after the word". This portion of the program is expanded:

```
NOTE - GOAL = COPY REST OF SENTENCE.
REPEAT

    NOTE - GOAL = COPY A WORD.
    REPEAT
        CALL 'WRITECH' USING CHAR.
        CALL 'READCH' USING CHAR.
    UNTIL ((CHAR = SPACE) OR (CHAR = PERIOD))

    NOTE - GOAL = GET NEXT NON-BLANK CHARACTER.
    WHILE (CHAR = SPACE) DO
        CALL 'READCH' USING CHAR.
    ENDDO

    NOTE - GOAL = WRITE PERIOD OR BLANK.
    IF (CHAR = PERIOD) THEN
        CALL 'WRITECH' USING PERIOD.
    OR
        CALL 'WRITECH' USING SPACE-CHAR.
    ENDIF

UNTIL (CHAR = PERIOD)
```

REPEAT and UNTIL are SCOBOL keywords which implement one of the D-structures for conditional repetition. Since the termination test is at the end of the sequence, the block of code will execute one or more times. The WHILE-DO-ENDDO SCOBOL construct implements a repetition with a test at the start of the code sequence; the code will be executed zero or more times. The IF-THEN-OR-ENDIF implements a completely nestable if-then-else; the keyword OR replaces ELSE to permit standard COBOL if-then-else's as well.

The last refinement of our sample program will be to write alternate words backwards. If words are to be written backwards, they must be saved as they are read. Forward words will be read and written; backward words will be read, saved, and then written.

The words which are to be written backwards will be saved in a 20 element table of characters. A variable WORD-LENGTH will be used to indicate the growing number of characters in the table, and a logical variable or "switch" FORWARD will indicate whether the word is to be written forward or backwards.

The "copy a word" routine is now replaced by:

```
NOTE - GOAL = COPY A WORD.

NOTE - FORWARD CASE FIRST.
IF (FORWARD = TRUE) THEN
    REPEAT
        CALL 'WRITECH' USING CHAR.
        CALL 'READCH' USING CHAR.
    UNTIL ((CHAR = SPACE) OR (CHAR = PERIOD))

OR
NOTE - AND NOW THE BACKWARD CASE.
    NOTE - GOAL = SAVE CHARACTERS IN WORD.
    MOVE ZERO TO WORD-LENGTH.
    REPEAT
        ADD 1 TO WORD-LENGTH.
        MOVE CHAR TO WORD (WORD-LENGTH).
        CALL 'READCH' USING CHAR.
    UNTIL ((CHAR = SPACE) OR (CHAR = PERIOD))

    NOTE - GOAL = WRITE IT BACKWARDS.
    REPEAT
        CALL 'WRITECH' USING WORD (WORD-LENGTH).
        SUBTRACT 1 FROM WORD-LENGTH.
    UNTIL (WORD-LENGTH = ZERO)

ENDIF
```

The entire SCOBOL program is listed below. It includes code to initialize FORWARD and to switch it after each word.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. 'SILLY'.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  CHAR                    PIC X.
77  PERIOD      VALUE '.'   PIC X.
77  SPACE-CHAR VALUE ' '    PIC X.
77  TRUE        VALUE '1'   PIC X.
77  FALSE       VALUE '0'   PIC X.
77  FORWARD                 PIC X.
77  WORD-LENGTH  COMP       PIC S99.
01  SAVED-WORD.
    02  WORD  OCCURS 20     PIC X.
PROCEDURE DIVISION.
    MOVE TRUE TO FORWARD.
    REPEAT
        CALL 'READCH' USING CHAR.
    UNTIL (CHAR NOT = SPACE)
    REPEAT
        IF (FORWARD = TRUE) THEN
            REPEAT
                CALL 'WRITECH' USING CHAR.
                CALL 'READCH' USING CHAR.
            UNTIL ((CHAR = SPACE) OR (CHAR = PERIOD))
        OR
            MOVE ZERO TO WORD-LENGTH.
            REPEAT
                ADD 1 TO WORD-LENGTH.
                MOVE CHAR TO WORD (WORD-LENGTH).
                CALL 'READCH' USING CHAR.
            UNTIL ((CHAR = SPACE) OR (CHAR = PERIOD))
            REPEAT
                CALL 'WRITECH' USING WORD (WORD-LENGTH).
                SUBTRACT 1 FROM WORD-LENGTH.
            UNTIL (WORD-LENGTH = ZERO)
        ENDIF
        WHILE (CHAR = SPACE) DO
            CALL 'READCH' USING CHAR.
        ENDDO
        IF (CHAR = PERIOD) THEN
            CALL 'WRITECH' USING PERIOD.
        OR
            CALL 'WRITECH' USING SPACE-CHAR.
        ENDIF
        IF (FORWARD = TRUE) THEN
            MOVE FALSE TO FORWARD.
        OR
            MOVE TRUE TO FORWARD.
        ENDIF
    UNTIL (CHAR = PERIOD)
    STOP RUN.
```

Goal directed programming, as it has been described in this paper, is a technique of problem decomposition that leads naturally to top-down, structured program design and coding. At each step in the process a goal is identified, an assertion is made that will be satisfied when the goal is met, and a logical condition is derived that can be tested in the program's control code.

How does GDP affect the testing and debugging process? Debugging can be the most time consuming part of program development. Since goal directed programs are structured, they share all the error reducing benefits of structured code. Additionally, in debugging a goal directed program the programmer systematically looks for unmet goals, unsatisfied assertions, and untested conditions. Goal directed reasoning helps in testing and localizing errors.

Documentation for goal directed systems is more meaningful and less tedious to produce than for the typical system in the past. It takes the form of a narrative of the design process and presents an idealized version of the solution process (i.e. without describing the backtracking which occurs in the actual process). It is important to remember that the only real documentation of a system is its code. Even in-line comments should be suspect (after all, they aren't executed!) Everything must be done to make the code readable. System documentation should be a designer's guide to the readable code.

Is goal directed programming applicable to large system design? In a large project communication among design team members and between the team and the user client is facilitated by the explicit factoring of the project into functionally oriented goals. At the team level the project can be properly analyzed in terms of goals instead of being mapped into a, possibly inappropriate, personnel organization. (Many systems look more like the organization that designed them than like the problems they solve.)

When users receive a statement of functionally specified goals and interact with the design team to refine these goals, incremental implementation, availability, and use of the system are possible. In this way users get some of the benefits of the new system even before it is completed, and the designers get the feedback of user experience while the design process is still underway. A goal directed presentation of the system design can be understood by even the naive user and can make a synergistic relationship between designers-programmers and end users possible.

What tools make goal directed programming easier? The basic tool of a programmer is his programming language; it has a tremendous influence not only on his coding style but on the way he thinks about problems. It should be small enough to be

intellectually manageable, should compile efficiently for a variety of current hardware, and should be expressive enough for the concise representation of algorithms. Ideally it should have the modern data and control structuring facilities that make it possible to proceed directly from the goal-assertion-condition transformation into program code.

Goal directed programming depends on the facility to group statements into functions or blocks each of which can be treated, at any arbitrary level of nesting, as a single statement. It is this statement bracketing facility that makes it possible to replace any block (e.g. a single statement) with any other block (e.g. "if condition then statement1 else statement2") without disturbing the surrounding code. When this is lacking in a programming language, top-down development by "stepwise refinement" and goal directed programming become more difficult and time consuming.

Languages like PASCAL, C, and BLISS are well suited for goal directed programming (there is probably no language that offers as much data structuring capability as PASCAL). But the great majority of commercial programmers code in FORTRAN or COBOL. Is goal directed programming possible in these languages?

Neither COBOL nore FORTRAN is a block structured language. FORTRAN is particularly deficient in control structures. COBOL gives the appearance of having the necessary structures (it has an if-then-else and the perform-until seems a reasonable variation of the while-do) but they are lacking in several important respects. COBOL's implementation of the if-then-else (with no statement bracketing) does not permit the arbitrary substitution of an if-then-else block for another block. For example, if

```
IF A THEN                          IF A THEN
    STATEMENT1  becomes                IF B THEN
    STATEMENT2.                            STATEMENT1A
                                       ELSE
                                           STATEMENT1B
                                       STATEMENT2.
```

the execution of STATEMENT2 has been affected when it should not have been. With a statement bracketing facility

```
IF A THEN                          IF A THEN
    STATEMENT1.  becomes               IF B THEN
    STATEMENT2.                            STATEMENT1A.
ENDIF                                  ELSE
                                           STATEMENT1B.
                                       ENDIF
                                       STATEMENT2.
                                   ENDIF
```

and STATEMENT2 has suffered no side effects from the modification to STATEMENT1.

COBOL's perform-until (where the performed code must be named and must be out-of-line as opposed to the in-line code bracketed by a while-do and enddo) leads to a proliferation of paragraph names and scattered pieces of code. This may interfere with readability rather than enhance it and may create system inefficiencies due to non-locality in a virtual storage environment.

The programming manager then is faced with a dilemma - his programmers need a better language and yet, for reasons of compatability, portability, etc. he may be reluctant to abandon COBOL or FORTRAN. Extensions to these languages have been discussed (and probably will be discussed for some time to come - committees work slowly!) An interim solution - one that can be implemented immediately - involves the use of a precompiler to translate programs written in a language more directly suited to the GDP technique into the target language (i.e. COBOL or FORTRAN). The program presented as a solution to the "silly program" problem was written in SCOBOL. This language and its precompiler were developed at VOLVO Flygmotor in Sweden. (A companion package for FORTRAN also exists.) SCOBOL permits the goal directed programming principles to be put into practice to speed the design and implementation of superior COBOL programs.

Goad directed programming is not a panacea or an instant solution to all programming problems. It is a language independent problem solving technique which, when carefully applied, can lead to quality programs. As such it deserves our study and experimentation.

For more information about goal directed programming, the SCOBOL and SFORTRAN precompilers, or a suggested reading list contact Martha Cichelli of Software Consulting Services, 901 Whittier Drive, Allentown, Pa. 18103 (215) 797-9690.