LEXICAL & SYNTACTIC CONVENTIONS FOR STRUCTURING PROGRAMS

by: Irving N. Rabinowitz

DCS-TR-41

Written in 1974

Department of Computer Sci.
Rutgers University
New Brunswick, New Jersey
Introduction

It has been recognized since the late sixties that the major source of difficulty and expense in the creation of large computer systems has been the construction and maintenance of the software, while hardware costs have shrunk dramatically in comparison with hardware power. One obvious difference between the software and hardware worlds is in the methodology of design: The hardware designer has a long history of applying well-developed engineering methods to his problem, while the software designer is still in the grip of a pre-industrial, cottage-industry tradition. Where the hardware designer can call on off-the-shelf modules with well-defined characteristics and standardized interfaces, the programmer usually finds himself constructing his modules and designing the interfaces anew for each problem he encounters. Where new machines are often composed of many parts from their predecessors, new programming systems usually contain only very few pieces of their ancestors, and these almost always modified on an ad hoc basis. In short, where hardware is engineering, software is art.

With the growth of machine capability, there has naturally followed a growth of the problems to which these can be applied, but where the difficulty of building stronger machines grows rather slowly, that of building larger and stronger software seems to grow at a much faster rate. A two-thousand line program is a good deal more than twice as hard to construct as a one-thousand line program. Where the probability of correct operation of a large computer is constrained largely by the probability of correct operation of its most elementary components, that of a large program is dependent not only on its elementary components,
but on interactions between them. These probabilities generally manifest themselves, not as numerical estimates, but in very subjective ways, as difficulties of understanding and coordination. Thus we have the spectacle of a group of programmers attempting to work together, often without any clear understanding of each other's problems, constructing modules of code that work "correctly", only to find that the program fails in subtle and often inexplicable ways when these modules are brought together.

The evolution of programming into "software engineering" must be accelerated if the promise of the computer revolution is to be realized. The production of large-scale software systems must somehow be achievable without exponential rates of growth of costs. Although we are still only at the beginning of this evolution, we can already see glimmerings of progress in some areas, namely those involved in the creation and interaction of modules of code, and in the construction of programs which must utilize these modules. These are the areas that have come to be known as "structured programming" and "top-down programming", respectively.

In this paper we will review some of the principles and techniques of structured programming. Section 1 describes a restricted set of control structures whose systematic use allows for easier writing and reading of programs. In Section 2 we present the principles of replacement of the rather rigid syntactic constructions of such structures by simpler lexical ones.
Section 1

Sets of Control Structures

It is by now well-known that all programs representable by flow charts of the "usual kind" can be constructed from a very small set of elementary structures. By flow charts of the usual kind we consider those composed of the following elementary boxes and arrows:

Computation:

Decision:

Path Join:

Start:

End:

All flow charts can be put together out of these elementary constructs. Indeed, standard flow-charting symbols are nothing more than suggestively-shaped variants of them.

Such a set of flow-charting symbols is at once too sparse and too dense. Its sparseness becomes evident when we consider the programming language statements corresponding to the figures. In Fortran terms the computation box is represented by an executable statement with one predecessor and one successor, for example, an assignment, a WRITE,
or a CALL. The start circle is represented by the implied initiation of a program at its first statement, and the end circle by a STOP or RETURN statement. The decision box is effectively

```
IF (?) GO TO ...
GO TO ---
```

while the joining of paths is achieved by the appearance somewhere of

```
GO TO ...
```

Even a language as impoverished of structuring as Fortran provides statements with more power than these. The main purpose of a higher-level programming language is to provide better ways of stating algorithms than by means of such elementary operations.

The other side of the coin is that this small set of constructs is so unconstrained in how they are to be connected to form a complete flow chart that anything can be done. This is well-illustrated in the frequent occurrence of non-planar flow charts, i.e., ones that cannot be drawn on paper without crossing lines. This very simplicity and generality is the cause of so much of our programming headaches. The unrestricted use of such simple elements leads to programs whose control structure is completely opaque.

The main offender in this situation is the seemingly innocuous joining of paths. In the absence of any restrictions on its use there is a requirement that somewhere there must appear a GO TO statement. This statement has, with some justice, been indicted for the crime of
making programs difficult to write, read, and debug. From the theoretical standpoint the GO TO statement can be completely eliminated from programming, in the sense that any program can be rewritten without such statements, sometimes at the cost of introducing auxiliary variables. From the practical standpoint, unrestricted use of GO TO's leads to the "bowl of spaghetti" program, in which paths of control weave tangled ways through the program. Although the use of GO TO cannot be universally condemned, it should be used in reasonable ways. Two restrictions on its use may be expressed simply as:

(a) No GO TO statement should refer to a previous statement in a program; and
(b) No GO TO statement should refer to a statement "very far down" from itself.

This first restriction is quite absolute, but the second is rather less precise as it stands. More precisely, under ordinary circumstances, we mean that a GO TO should never do other than leave a loop in which it is contained.

The achievement of restriction (a) does not by any means imply that there be no loops in a program--such programs would be meager indeed! Rather, we insist on using a set of control structures in which looping is explicitly indicated, as in Fortran's DO or Cobol's PERFORM UNTIL. Such a set of control structures will now be presented.

A minimal set of structures which theoretically suffices to represent all programs, and practically allows almost all programs to be easily constructed is the following:
(1) The composition of boxes:

Precisely, we should insist that the contents of this box be an assignment of the form

$$x \leftarrow f(x),$$

where $\overline{x}$ represents the collection of all variables of the program. Less formally, we consider the contents of this box to be, as before, a simple executable statement which does not alter the sequence of control.

(2) The IF-THEN-ELSE:

In this, $C$ represents a condition which may be either true or false (and nothing else!). If it is true, the "box" $B_1$ is executed; if false, "box" $B_2$. In both cases control then joins.

(3) The DO-WHILE or DO-UNTIL

This, of course, is the construct that allows looping. $C$ is a condition, evaluated on each entry to the loop. As long as $C$ is true, the body of the loop is executed. When it becomes false, we "drop through". The DO-UNTIL uses testing at the end rather than the beginning.

An important point may be noted about each of these structures: Each has exactly one entry line and one exit line. Thus we can allow substitution of any of the constructs (1), (2), or (3) for any box,
achieving our purpose of creating general flow charts. Another result of this one-entry, one-exit characteristic is that a flow-chart can effectively be read as a linear text, while still maintaining the clarity of a two-dimensional representation.

An example of the use of these constructs may be in order. The following flow chart represents the program which, given the integer NMAX, will produce magic squares of odd order from 3 to NMAX. This flow chart is constructed only with the structures described above. That is, none of the backward-pointing arrows represents a go-to statement, but each is a member of a do-while loop. A convention has been adopted in this flow chart of drawing the body of a do-while to the right of the testing box, so that nesting of loops is indicated by a southeasterward placement of boxes. It may be remarked that consistent use of this convention makes the upward-pointing arrows superfluous. Thus, if a series of boxes in a vertical line comes to a dead-end, this can, with only a very little bit of getting-used-to, be seen as the end of a loop, with the closing arrow supplied by the mind's eye almost automatically. However, for the sake of clarity of illustration, we explicitly draw each loop-closing arrow.
START

N = 3

N <= NMAX ?

P = 0; NSQR = N^N
I = (N+1)/2; J = N

P <= NSQR ?

Q = P; P = P + 1
SQUARE(I, J) = P

P <= Q ?

N

I = N ?

I = I + 1
J = J - 1

J = J + 1

J = 1

P = P + 1
SQUARE(I, J) = P

Print SQUARE on N lines

N = N + 2

STOP
Section 2

Lexical Program Structuring

Although the use of well-structured elements can have a major influence on the ease of writing programs, most programming languages have a syntax which fails to take advantage of the two-dimensional nature of writing paper and of machine-readable source documents. Even Fortran and Cobol, whose input format is record-oriented (as opposed to the stream-oriented input format of PL/I and other Algol-like languages), do not include any use of horizontal spacing except as the programmer chooses to use it to improve readability. It is clearly desirable to use such spacing, primarily in the form of indentation of statements, but such indentation is normally completely ignored by a compiler.

A systematic use of indentation can not only make programs easier to write and to read, but can obviate much of the punctuation that is ordinarily required to break up the input stream into recognizable statements. For example, the \texttt{begin-end} of Algol, or the \texttt{DO-\texttt{END}} of PL/I can be eliminated entirely. As Fortran shows us, the semicolon used as a statement separator can be made to disappear completely, except where the programmer specifically chooses to write it. More importantly, the control structure of the program is displayed in a very readable two-dimensional form. As a simple example, consider the PL/I group

\begin{verbatim}
DO WHILE (condition);
   group of statements ;
END;
\end{verbatim}

While there is nothing objectionable in this \textit{per se}, it should be recognized that the three semicolons and the word END are merely punctuation. (Indeed, the semicolon after the END seems to be even more
superfluous than the other two, appearing to be punctuation on top of punctuation.) The meaning of the group of statements is quite well expressed by

\[ \text{DO WHILE (condition)} \]
\[ \text{group of statements} \]
(2)

where the punctuation is completely obvious from the layout.

The systematic use of indentation in writing programs can make the structure of a program clear not only to a human who has to read it, but also to a compiler that must translate it. The basic principle involved here is simply that a group of statements to be executed in sequence should all begin at the same margin, and that a group subordinate to a control statement should be indented to the right of that control statement.

This formatting may be formalized in such a way that the analysis of the control structure of a program can be done on a strictly lexical basis, that is, without reference to the internal structure of the statements. Thus programs written in this style can be processed by a relatively simple pre-scanning processor, and converted into the strict language for compilation by the standard compiler. For example, the group (2) can be converted to the strict form (1) by the scanner simply by having the scanner (a) add terminating semicolons when the end-of-card is met, and (b) emit the statement "END;" when the margin at which statements are typed makes a leftward jog. These decisions may be made without reference to the detailed structure of the statements, but by noting only the key words that indicate control statements.

Informally, the principle is clear. To formalize it, we introduce in the usual syntactic descriptions some new metasymbols for indi-
cating indentation or no indentation, and an interpretation of the productions which corresponds to the record-oriented nature of the language. The interpretation we adopt is that any symbol, whether terminal or non-terminal, on the right side of a production is to be considered as starting a new input record, except where modified by the metasymbols $i$ and $\Omega$.

The metasymbol $i$ preceding another symbol on the right side of a production indicates one level of indentation with respect to that of the enclosing statement. More precisely, consider a production of the form

$$X ::= Y_1 \ i \ Y_2 \ Y_3 \ .$$

In the normal interpretation of such productions, the strings derived from the $Y_i$'s are considered to be concatenated without regard to any layout on the "page". Our interpretation, in the absence of the $i$, is that these three strings are to be vertically aligned. The effect of the $i$ is that the string derived from $Y_2$ (or the symbol $Y_2$ itself, in the case that it is terminal) is indented one level from the margin at which the string derived from $Y_1$ is indented. That $Y_3$ is not preceded by $i$ indicates that the string derived from $Y_3$ is indented at the same level as that of $Y_1$. The level of indentation of the defined symbol $X$ is the same as that of the first symbol on the right side of the production. Thus we restrict the appearance of the $i$ to interior positions on the right, and prohibit it from appearing in the first position. The appearance of multiple $i$'s preceding a symbol indicates multiple levels of indentation relative to that of the first symbol. Thus, the production

$$X ::= Y_1 \ i \ Y_2 \ \Omega \ Y_3 \ Y_4 \ i \ Y_5$$

would give rise to the following derived string:
string derived from Y1

string derived from Y2

string derived from Y3

string derived from Y4

string derived from Y5

We introduce the metasymbol \( \_ \) to suppress the normal interpretation that a symbol on the right side starts a new line. This is needed to allow the freedom to continue statements on the same line so that, for example, in defining an if-statement as IF condition \( \_ \), we do not force this to be interpreted as meaning

\[
\text{IF condition.}
\]

Instead, we will write IF \( \_ \) condition, to allow the interpretation that the condition follow on the same line. More precisely, the appearance of \( \_ \) preceding a symbol forces the usual interpretation, namely that of concatenation of strings, to be made. For example, the production

\[
X ::= Y_1 \_ Y_2 Y_3
\]
gives rise to the derived string:

\[
\text{string from Y1 string from Y2 string from Y3}
\]

The basic structured elements may then be defined as follows, where for definiteness in referring to a language, we adopt the PL/I form of the constructs. Similar definitions can be given for any language in which these constructs are defined.

(1) Composition of statements.

\[
\text{statement-group ::= statement | statement-group statement}
\]

These productions imply that all the statements of a group to be executed sequentially share the same margin. If this were the only definition of
statement-group, we would be forced into the Fortran style of no more than one statement per line. It is often convenient to allow more than one statement per line, as for example in a multiple initialization:

\[ A = 0 ; \quad B = 1 ; \quad C = 0 \]

In such a case, the explicit semicolon is required. To permit this, we add one more production to the definition of statement-group, namely

\[ \text{statement-group ::= statement-group } \quad c ; \quad c \text{ statement} \]

(2) IF - THEN - ELSE.

Using square brackets to indicate an optional appearance of a syntactic element, we define (where the indentation here is only to aid the eye, and is not part of the definition - there the 's have it)

\[ \text{if-then-else ::= IF } \quad c \text{ condition} \]
\[ \quad i \quad \text{THEN} \quad [ \quad c \text{ statement-group } ] \]
\[ \quad [ \quad ii \text{ statement-group } ] \]
\[ \quad i \quad \text{ELSE} \quad [ \quad c \text{ statement-group } ] \]
\[ \quad [ \quad ii \text{ statement-group } ] \]

This definition requires that the condition of the IF be on the same line as the word IF, as is the ordinary practice. (It may be noted that this production is not quite right, as it allows null then- and else-parts. That is, since both statement-groups on each leg are optional, neither need appear. Since this is primarily for explanation rather than strict language specification, let us not get too precise at the expense of obscuring the substance.) This definition allows an if-then-else to appear in different ways, to the programmer's taste, but with the indentation always determining the structure. Thus we may have
IF \( A + B = C - D \)
THEN
\( U = V + W \)
ELSE
\( U = V - W \)

When the two legs of the if-then-else consist of groups of statements, this is the preferable form. However, for a statement like this one, it is certainly more natural to put the single statements of each leg on the same line as the keyword. The definition allows the above to be written:

IF \( A + B = C - D \)
THEN \( U = V + W \)
ELSE \( U = V - W \)

(5) DO-WHILE.

do-while ::= DO \( c \) WHILE \( c \) ( \( c \) condition \( c \) )
\( i \) statement-group

Although not involved in the structuring of control, declarations in a program can be written to either enhance or to obscure the readability of the program. Thus in PL/I, a list of variable declarations sharing the same margin may be considered a single declaration, with the preprocessor putting in the necessary commas. Although this is rather a bit of icing on the cake, the following illustrates how we may impose indentation characteristics throughout a programming language:

declaration ::= DECLARE [ \( c \) variable-declaration ]
[ \( i \) variable-declaration ] ...
(In this, we use the ellipsis to indicate zero or more repetitions of the immediately preceding syntactic element.)

procedure-declaration ::= name \( c \) : \( c \) PROCEDURE \( c \) (
\( c \) parameter-list \( c \) )
\( i \) statement-group
Here indentation is used to specify that the entire body of the procedure is subordinate to the procedure heading, all of which is contained on a single line.

Since our interpretation of a production is that each symbol is to start a new line (except as modified by the $\text{\&}$), the realization of this by the pre-processor is that a new line starts a new statement to be parsed into a single non-terminal symbol. We need some mechanism to suppress this interpretation on the part of the pre-processor. That is, we need "continuation cards". This may be incorporated into our scheme by recognizing that each new statement must start with one of a small group of characters. In PL/1 this small group consists of the letters and the left-parenthesis, which may appear as the first character of a factored declaration. Thus we make the convention that if the first non-blank character of a new line is other than one of these, it indicates that the rest of the card is to be a continuation of the previous one.

As an illustration of the principles described above, we present the "indented PL/1" program corresponding to the flow chart of Section 1.
MAGIC: PROCEDURE (NMAX)

DECLARE (N, NMAX)           FIXED
DECLARE SQUARE(N, N)        FIXED
   (I, J)                  FIXED
   (P, Q, NSQR)            FIXED

N = 3
DO WHILE ( N <= NMAX)
   P = 0 ; NSQR = N*N
   I = (N + 1) / 2 ; J = N
   DO WHILE ( P <= NSQR )
      Q = P + N ; P = P + 1 ; SQUARE(I, J) = P
   DO WHILE ( P <= Q )
      IF I = N
         THEN I = 1
      ELSE I = I + 1
      IF J = N
         THEN J = 1
      ELSE J = J + 1
      P = P + 1 ; SQUARE(I, J) = P
      J = J - 1
   DO I = 1 TO N
      PUT SKIP EDIT ((SQUARE(I, J) DO J = 1 TO N)) ((N)F(6))
   N = N + 2
BIBLIOGRAPHY

The basic result concerning the control structures needed is in
Böhm, C. and Jacopini, G., "Flow Diagrams, Turing Machines, and Lan-

A more detailed, but almost as heavily theoretical, discussion is
Mills, H., "Mathematical Foundations for Structured Programming", Re-
port Number FSC 72-6012, Federal Systems Division, IBM Corporation,
Gaithersburg, Maryland 20760. (February 1972).

A programming text which develops the principles of structured program-
ming and top-down program construction is

A useful overview of the subject is to be found in the December 1973
issue of Datamation, where five articles cover various facets.

An interesting discussion on the "go-to controversy" was held at the
1972 ACM Conference, and the papers have been published in those Pro-
ceedings. They were reprinted in Sigplan Notices, 7, 11, 54-91, which
includes some of the discussion at the ACM session. The papers are
by Leavenworth, Hopkins, and Wulf.

The whole subject of structured programming first saw the light of day
in a letter by Edsger Dijkstra: "Go to Statement Considered Harmful",

The paper by Leavenworth, entitled "Programming With(out) the GOTO",
mentioned above, contains an extensive bibliography on the subject of
structured programming.