Triodic Logic and its Use in Structured Program Design

N. E. Goller
Operational Research Executive, NCB, Coal House, Lyon Road, Harrow, Middlesex, UK

A triode is a program block with one entry and two exits. This paper examines some of the consequences of taking triodes to be the basic units of program structure. The result is a useful generalization of the existing theory and practice of structured programming.

1. INTRODUCTION

The triodic notation developed in this paper is a streamlined, but otherwise quite modest generalization of structured programming notations already in use. Two benefits have been found to result from this in practice. Firstly, the notation is particularly concise and flexible. Therefore it is rewarding to use, and easy to modify and debug. Secondly, the idea of taking triodes as basic building blocks has proved a valuable way of thinking about program design.

Sections 2 and 3 describe the basic structures of triode-based logic; section 4 gives some elementary examples of their use. Section 5 describes how these structures may be implemented in a program design language intended for mechanical translation into FORTRAN. Section 6 describes some changes to the notation which have been found desirable in practice. Section 7 shows how the basic structures may be enhanced by a notation for quantified loops. Sections 8 and 9 give a brief account of some practical experience of using triodic structures for error-handling and for parsing respectively. Conclusions are presented in section 10.

For the past two years the author has made constant use of the methods discussed here in the course of writing applications and systems programs for the Operational Research Executive of the National Coal Board. The programs written by these means have carried out a fair mixture of scientific, algorithmic, and data-processing tasks and have ranged in size from a few lines to a few thousand lines.

2. THE BASIC BUILDING BLOCKS

The swiftest way to define the concepts we need is by flow diagrams. A *triode* is any program block (Fig. 1) that has one entry and two exits. These will be called the *main* and *alternative* exits, and in diagrams here will be shown as emerging from the right and from the bottom respectively. If the alternative exit does not occur we get a special case (Fig. 2) called a *diode*.

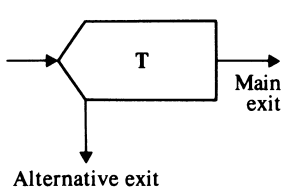

Figure 1. A triode.

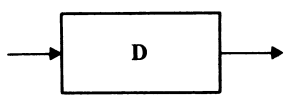

Figure 2. A diode.

If more formal definitions are required, we say that a triode is a triple (P, M, A) where P is a Turing computible procedure and (M, A) is a partition of its halting states into the two disjoint classes M, A. This triode is a diode if A is empty. Any procedure can be naturally identified with a corresponding diode by assigning all its halting states to the main exit.

Let B be a boolean expression. Then **test** (B) will denote a procedure that evaluates the expression; in the event that this procedure halts it will take the main exit if B is true and the alternative exit if B is false (Fig. 3).

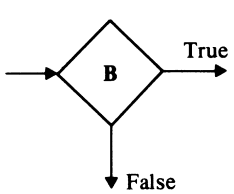

Figure 3. The triode test (B).

When this notation is used in a program text, we shall permit it to be abbreviated to (B) alone; the parentheses are retained but the keyword **test** omitted.

3. NEW TRIODES FROM OLD

Let **T1** and **T2** be triodes. We define two binary operators, semicolon and hash, by displaying their results **T1**; **T2** and **T1 # T2** in Fig. 4. These operators obey the associative law, that is to say

{T1; T2}; T3 = T1; {T2; T3}
and
{T1 # T2} # T3 = T1 # {T2 # T3}

CCC-0010-4620/82/0025-0218 $04.50

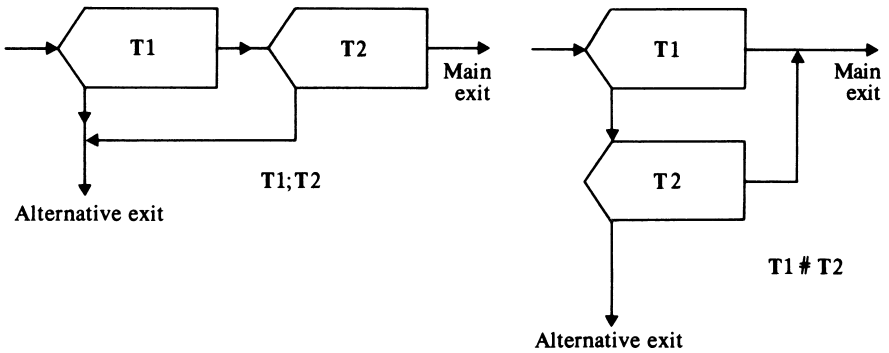

Figure 4. The semicolon and hash operations.

making the brackets in these expressions unnecessary. We shall decree that semicolon is evaluated inside hash, reducing further the need for brackets: thus

T1 # T2; T3 # T4; T5; T6
denotes
T1 # {T2; T3} # {T4; T5; T6}.

When bracketing is still needed we shall use the keywords **tri** . . . **irt** in place of the brackets {. . .}.

The next three operators are unary, and use keywords that imply bracketing; for example **do**. . .**od** is understood as **do** {. . .} **od**.

rev T ver interchanges the exits of **T** (Fig. 5).

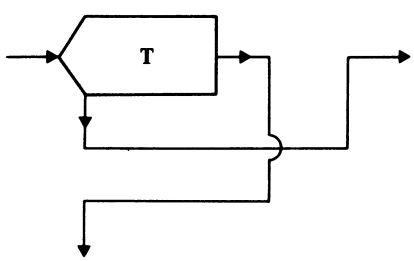

Figure 5. The triode **rev T ver**.

do T od loops the main exit to the entry (Fig. 6)

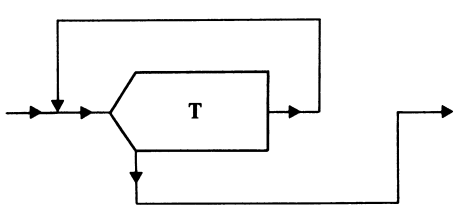

Figure 6. do T od.

if T fi amalgamates the exits (Fig. 7).

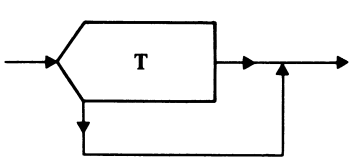

Figure 7. if T fi.

The result of **do**. . .**od** or **if**. . .**fi** is always a diode.

It should be clear that the semicolon and hash operations are analogous to boolean 'and' and 'or'. The **rev**. . .**ver** construct is analogous to boolean 'not'. The analogy is exact if we let B, C be boolean expressions and set

$$\mathbf{T} = \mathbf{test}\ (\mathrm{B}),\ \mathbf{U} = \mathbf{test}\ (\mathrm{C}).$$

Then

$$\mathbf{T; U} = \mathbf{test}\ (\mathrm{B}\ \&\ \mathrm{C})$$
$$\mathbf{T \# U} = \mathbf{test}\ (\mathrm{B}|\mathrm{C})$$
$$\mathbf{rev\ T\ ver} = \mathbf{test}\ (\neg\ \mathrm{B}).$$

Further constructs to be given in section 7 will be analogous to (B implies C), $(\exists x \in \mathbf{S} \cdot \mathrm{B}(x))$, and $(\forall x \in \mathbf{S} \cdot \mathrm{B}(x))$.

4. EXAMPLES

The use of the constructs so far given can be made plain by giving the triodic equivalents of some standard program structures:

Standard structure	*Triodic equivalent*
if B then **D1** else **D2**	**if** (B); **D1 # D2 fi**
while B do **D**	**do** (B); **D od**
repeat **D** until B	**do D**; (¬B) **od**

The next example illustrates the Euclidean algorithm to find the greatest common divisor of two positive integers X and Y, firstly in the notation of Dijkstra (Ref. 1, p. 45):

$x, y := X, Y;$
do $x > y \rightarrow x := x - y$
[] $y > x \rightarrow y := y - x$
od;
print (x)

and next the triodic equivalent:

$x, y := X, Y;$
do $(x > y); x := x - y$
$\#(y > x); y := y - x$
od;
print (x)

The two notations correspond to one another exactly in this example. It should however be noted that Dijkstra's [] notation causes the alternatives to be tested at random (by an unpredictable demon!) whereas my # notation causes the alternatives to be tested in sequence. Therefore differences in meaning between the notations will occur if the boolean tests are not mutually exclusive.

The following program tests successive fields of a

record, performing instead an error routine PX if any test fails. The first version is that of Jackson (Ref. 2, p. 123):

```
PCARD posit good card
  PGOOD seq
          quit PCARD if error F1;
          do P1;
          quit PCARD if error F2;
          do P2;
          quit PCARD if error F3;
          do P3;
  PGOOD end
PCARD admit
        do PX;
PCARD end
```

The triodic version is as follows:

```
if (¬F1); P1; (¬F2); P2; (¬F3); P3
#PX
fi
```

5. AN IMPLEMENTATION

Assume now that the triodic constructs so far given are to be incorporated into a source language for programming, and that this source language is to be translated into a target language such as Fortran which has not the triodic constructs but has labels and gotos.

We may design a program to do this translation as follows. It will read the source text sequentially while manipulating a stack of labels.

We start with the stack instructions shown in Table 1.

Table 1. Operations on stack of labels

Instruction	Meaning
ext	invent new label, push on top of stack
label	produce label in target text matching top element of stack
goto	produce goto in target text whose destination is the (top − 1) element of stack
cgoto (B)	produce statement in target text of the form: if (not B) goto top element of stack
pop	pop top element of stack
copy	push duplicate of top element on to stack
swap	swap top and (top − 1) elements of stack

As the translating program reads the source text it converts triodic notation into the stack instruction sequences shown in Table 2. The bracketing conventions are automatically observed, at the cost of some extra labels that will not always be needed.

This translator, though usable as it stands, is fairly crude. Two improvements are discussed in Appendix 1 (the treatment of 'unreachable code') and Appendix 2 (an optimization that removes some redundant labels). Complete removal of redundant labels requires the use of a 'tree-walking' translator (see Ref. 3, p. 29) rather than the present 'string-walking' kind.

We must add a mechanism for diodic and triodic procedure calls. If the target language is Fortran, a callable diode becomes a Fortran SUBROUTINE called

Table 2. Implementation of triodic constructs by stack instruction sequences

Source text	Instructions
test (B)	cgoto (B)
;	(none)
#	goto; label; pop; ext
do	ext; label; ext
od	goto; label; pop; pop
if	ext; ext
fi	label; pop; label; pop
tri	ext; ext
irt	goto; label; pop; goto; label; pop
rev	ext
ver	goto; label; pop
out	pop; goto; label; pop

in the usual way. A callable triode is written in the following manner as a Fortran LOGICAL FUNCTION.

Suppose that T is the textual sequence for the triode that would be used if it was written in line. Then the corresponding text for the same triode written as a separate callable module will be:

```
LOGICAL FUNCTION functionname (parameters)
if T; functionname: = true
# functionname: = false
fi
```

The calling sequence, which in the calling program replaces the text of T in line, is:

test (*functionname* (*parameters*))

If the callable triode is to have no parameters then in the Fortran version we pass a dummy parameter so that the same method of translation may still be used.

6. USEFUL MODIFICATIONS TO THE SYNTAX

Without increasing the computational power of the language we can modify the syntax in ways that are useful in practice.

6.1. An end of line acts like a semicolon, except when it occurs after a comma or within an unclosed parenthesis. In the latter cases it is non-significant. (This rule provides the 'continuation convention'.)

6.2. A comment is introduced by an exclamation mark and terminated by end of line; the end of line is significant or not just as in 6.1.

6.3. The presence or absence of semicolon is immaterial before or after: hash, another semicolon or (as in 6.1, 6.2) virtual semicolon, or any of the keywords of sections 3, 6.5 and 7. Thus we may if we choose write

do; **if**; T; U; # V **fi**; # W **od** *X*

in place of

do if T; U # V **fi** # W **od** *X*

(The above conventions are adapted from existing languages such as Ratfor[4] and BCPL[5]).

6.4. The **test** keyword may be omitted, as already stated in section 1; the retained parentheses making the abbreviation unambiguous.

6.5. Sooner or later one discovers the frequent need for a construct that joins a diode in series with the alternative exit of a triode (Fig. 8). The notation **tri T # D out** will be

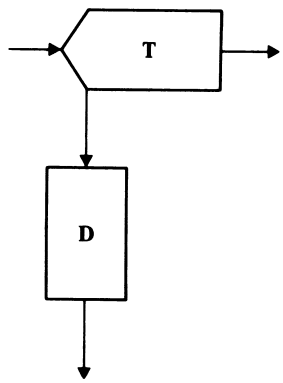

Figure 8. tri T # D out.

used for this construct, this being more convenient than writing **tri T # rev D ver irt**. More generally **tri T1 # T2 # . . . Tn # D out** is to mean **tri T1 # T2 # . . . Tn # rev D ver irt**.

Note that this slightly distorts the grammatical structure developed so far, in that **tri T # D out** cannot be thought of as **tri** {**T # D**} **out** since the connection between **T** and **D** is no longer that expressed by **T # D** in isolation. This distortion has no serious consequences, however; the reason for this is that we can choose to regard **out** as an abbreviation for ;**test** (false) **irt** in which case the grammar fits neatly into place.

(It is worth mentioning that the present notation was only arrived at after much experiment. Some proposed notations were too fiddly. Others turned out to entail grammatical changes of a serious nature, for example having to replace Table 4 by a more complicated grammar that kept diodes and proper triodes rigidly separated).

6.6. Various forms of 'assertion' statement[6] are useful for purposes of debugging and error-handling. We therefore define **canthappen** to be a diode whose effect, when executed, is to cause a message to be written saying 'false assertion at line . . . of source text'.

If B is a boolean expression, we define **assert** (B) to be a triode equivalent to:

tri (B) # **canthappen out**

This construct results in the same flow of control as **test** (B) but additionally causes a false assertion message to be written if B is false. The programmer encountering the false assertion message can then refer to the designated line of source text in order to find out what unwanted circumstance caused the message to be written.

Other forms of 'assertion' statement can set a globally accessible error flag instead of writing a message. This is particularly useful when the error occurs inside a low-level module, so that the programmer can call the module, then test the flag, and thereby construct error messages that relate to the intended application of the module rather than to its internal workings.

7. QUANTIFIED LOOPS

Constructions resembling the following are among the commonest in programming:

$i := 1$; **do** $(i \leq n)$; (array (i) = value); $i := i + 1$ **od**; $(i \leq n)$

$i := 1$; **do** $(i \leq n)$; array $(i) :=$ value; $i := i + 1$ **od**

The first searches an array for a specified value, ending with i set to the index of the first such value found. The final test of $(i \leq n)$ makes the construction into a triode that takes the alternative exit if value is not found.

The second construction sets all elements of array to value.

Such constructions may be cobbled together easily enough using **do . . . od** loops, and this is often the best way to write them. Where the required construction sticks closely to the fixed pattern shown in the above examples, however, it is natural to seek a more concise notation. In the notation to be developed here the above examples will in fact become:

find $i =: 1 \leq n$; (array (i) = value) **found**
for $i =: 1 \leq n$ **exec** array $(i) :=$ value **alldone**

To explain this notation in its generality we must introduce a new kind of building block. A *quantode* is any program block (Fig. 9) that has two exits (*main* and *alternative*) and also two entries (*initial* and *secondary*).

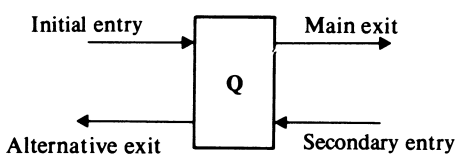

Figure 9. A quantode.

These are positioned conventionally for diagrammatic purposes as shown in the figure.

A quantode may be thought of as an operator on triodes. Figure 10 shows how a new triode results when

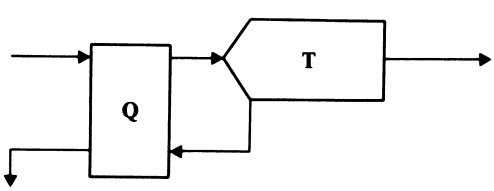

Figure 10. A quantode operating on a triode.

a triode is operated on by a quantode. Each quantode described here acts as a program block that after suitable initialization generates successive elements of some set.

To construct a quantode from scratch we use the notation:

first F next N check T finish

Here **F**, **N** are diodes, **T** is a triode, and the resulting quantode is defined to be that of Fig. 11. Where the set

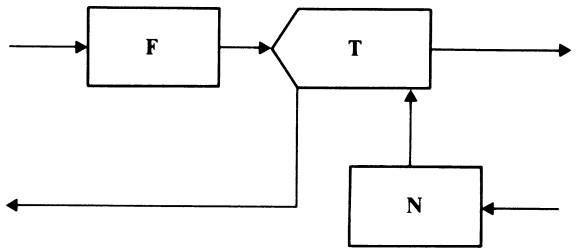

Figure 11. The quantode **first F next N check T finish**.

to be generated is a set of consecutive integers, we define an abbreviated notation:

$i =: m \leq n$ denotes **first** $i := m$ **next** $i := i + 1$ **check** $(i \leq n)$ **finish**

$i =: n \geq m$ denotes **first** $i := n$ **next** $i := i - 1$ **check** $(i \geq m)$ **finish**

(If in fact $m > n$, each quantode generates the empty set, as a consequence of these definitions).

We next define some operations on quantodes. Let **Q1**, **Q2** be quantodes, then **Q1**;**Q2** is defined as in Fig. 12.

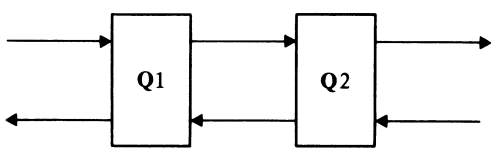

Figure 12. The semicolon operation on quantodes.

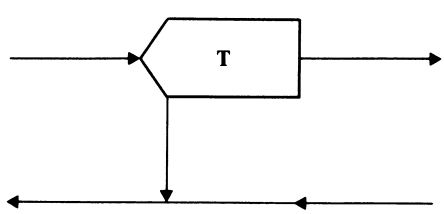

Figure 13. A triode regarded as a quantode.

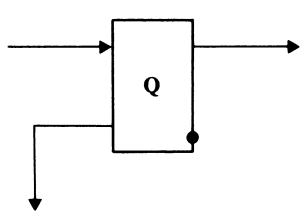

Figure 14. find Q found.

Now if **T** is any triode, we shall allow it also to be regarded as the quantode shown in Fig. 13: the secondary entry coincides with the alternative exit. Under this interpretation the semicolon operation on quantodes is a generalization of the same operation on triodes.

The hash operation is not defined on quantodes. The two remaining operations are the following:

find Q found converts a quantode to a triode by forgetting the secondary entry (Fig. 14). Note now, in view of the preceding definitions, that Fig. 10 represents **find Q**;**T found**.

If **Q** is a quantode and **T** a triode then **for Q exec T alldone** is defined to be the triode shown in Fig. 15, which is equivalent to **rev find Q**; **rev T ver found ver**.

The implementation of quantodic operations by stack instruction sequences is shown in Table 3. Quantodic procedure calls can be implemented as follows for translation into Fortran. Suppose the quantode is **first F next N check T finish**. We invent a new boolean variable *INIT*. The callable version of the quantode is the triode (ready for translation into a Fortran LOGICAL FUNCTION as in Section 5):

if (*INIT*); **F # N fi**; **T**

The calling statement is

first *INIT* := true **next** *INIT* := false
check (functionname (parameters, *INIT*)) **finish**

Table 3. Implementation of quantodic constructs by stack instruction sequences

Source text	Instructions
;	(none)
find	copy
found	pop
for	ext; copy
exec	ext
alldone	goto; label; pop; pop; goto; label; pop
first	ext
next	ext; goto; label; swap
check	label; pop; swap
finish	pop

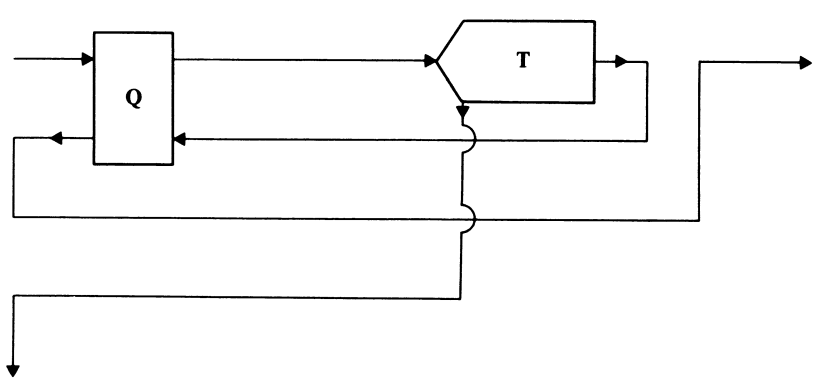

Figure 15. for Q exec T alldone.

Table 4. A BNF grammar for triodic program design[a]

⟨monode⟩	::= **halt**	
	|⟨goto statement⟩	
⟨diode⟩	::= ⟨monode⟩	
	|⟨assignment statement⟩	
	|⟨procedure call⟩	
	|⟨label statement⟩	
	|**do** ⟨H-group⟩ **od**	
	|**if** ⟨H-group⟩ **fi**	
⟨triode⟩	::= ⟨diode⟩	
	|**test** (⟨boolean expression⟩)	
	|**tri** ⟨H-group⟩ **irt**	
	|**tri** ⟨H-group⟩ # ⟨D-group⟩ **out**	
	|**rev** ⟨H-group⟩ **ver**	
	|**for** ⟨Q-group⟩ **exec** ⟨H-group⟩ **alldone**	
	|**find** ⟨Q-group⟩ **found**	
⟨quantode⟩	::= ⟨triode⟩	
	|**first** ⟨D-group⟩ **next** ⟨D-group⟩ **check** ⟨H-group⟩ **finish**	
⟨D-group⟩	::= ⟨diode⟩ |⟨D-group⟩; ⟨diode⟩	
⟨T-group⟩	::= ⟨triode⟩ |⟨T-group⟩; ⟨triode⟩	
⟨Q-group⟩	::= ⟨quantode⟩ |⟨Q-group⟩; ⟨quantode⟩	
⟨H-group⟩	::= ⟨T-group⟩ |⟨H-group⟩ # ⟨T-group⟩	

[a] All problems concerned with unreachable code have been ignored in the above grammar.

The BNF grammar shown in Table 4 is included to make clear which operations on triodes and quantodes are syntactically allowable.

The following examples will briefly illustrate the uses of quantodes.

Example 1

T and **U** are triodes: then **for T exec U alldone** is equivalent to **rev T**; **rev U ver ver** (Fig. 16), the triodic analogue of boolean implication.

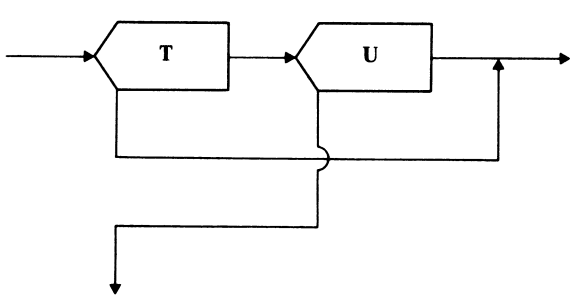

Figure 16. The analogue of boolean implication.

Example 2

Suppose B (x) is a boolean function, **Q** a quantode that generates elements x of a set **S**, and let **T = test** (B(x)). Then **find Q**; **T found** is equivalent to **test** ($\exists x \in \mathbf{S} \cdot \mathrm{B}(x)$) and **for Q exec T alldone** is equivalent to **test** ($\forall x \in \mathbf{S} \cdot \mathrm{B}(x)$).

Example 3

To multiply two matrices:

```
for i =: 1 ≤ m; k =: 1 ≤ p exec c(i, k) := 0;
    for j =: 1 ≤ n exec c(i, k) := c(i, k) + a(i, j)*b(j, k)
    alldone
alldone
```

Example 4

Find non-zero element of matrix to act as pivot. Then carry out pivoting operation on all elements other than those in pivotal row and column. This example illustrates well the economy of the notation.

```
if
    find ipivot =: 1 ≤ m; jpivot =: 1 ≤ n; (a(ipivot,
      jpivot) ≠ 0)
    found;
    for i =: 1 ≤ m; (i ≠ ipivot); j =: 1 ≤ n; (j ≠ jpivot)
      exec a(i, j) := a(i, j) − a(i, jpivot)*a(ipivot,
        j)/a(ipivot, jpivot)
    alldone
# ! handle case of all elements zero
    ...
fi
```

8. BREAKS AND ERROR-HANDLING

The alternative exit of a triode may be used as a kind of 'break' so that when an error is detected it is prevented from causing further damage. The Jackson example at the end of Section 3 illustrates this use of triodic notation.

One might wish to implement other forms of 'break', such as occur in many programming languages. My own shortlist of break constructs worth implementing consists of (i) **halt** (break from program, leaving all files in a standard 'closed' state); (ii) labels and **goto**. The first of these is necessary for example to halt from within an error-handling submodule. The second is not often needed once one has learnt to use triodic constructs effectively, but still worth preserving for occasional use. (It is taken for granted that a main program will halt, and a subprogram will return, at the end of its program text. Only 'halts' and 'returns' from other textual positions are to be regarded as breaks, and only these need to be notated explicitly.)

Other types of break, that have been tested in the triodic context but have proved not worth the trouble of implementing, are the following: **break** (from current construct—jumping over hashes, whereas the ordinary alternative exit just jumps as far as next hash); **break** *n* (to break out of *n* nested constructs); **break** *constructname* (assuming constructs may be named, as in Jackson's notation); **return**.

The trouble with all of these is that they cause more incomprehensibility than, and have no recognizable advantage over, the explicit use of labels and gotos. For example the presence of a label at the end of a subprogram alerts one to the unusual control flow, whereas a **return** somewhere within the text might be overlooked.

On detecting an error, it is almost always best to handle

it locally, rather than 'going to' somewhere else. The following example illustrates this.

Suppose that any instance of triode **T** being activated and terminating at its main exit ought to be followed by an instance of triode **U** being activated and terminating at its main exit. We are prepared to handle alternative exit from **T**, and we are prepared to handle main exit from **U**, but we are not prepared to handle the exceptional case of alternative exit from **U**. If we can guarantee that this exceptional case will not occur, then the natural program structure is **if T;U fi**.

Suppose now that, owing to some error, **T;U** exits from the alternative exit of **U**. We wish to handle this differently from the correct cases of main exit from **U** or alternative exit from **T**. Figure 16—which has a 'break' in just the right place—seems the obvious way of doing this.

Having 'broken' from the alternative exit of Fig. 16 what do we do now? It is improbable that we shall be able to do anything more profound than the following: perform some fudging action **D** (a diode) that deludes the rest of the program into thinking that no error has occurred; and write an error message to warn the human observers that the program has become deluded.

The straightforward way to handle an error is to do so without changing the shape of the error-free flow of control. Therefore we stop thinking about 'breaks' and write **if T; if U # canthappen; D fi fi**. If there is no good way of fudging the error, then **D** simply becomes **halt**.

This advice on error-handling may appear unsophisticated; surprisingly so, perhaps, since the triodic concept might seem to be particularly well suited to dealing with errors. Unfortunately, a program cannot deal with errors in any other than an unsophisticated way unless it maintains within itself a dynamic model of its own execution. This is the real problem of error-handling, and has nothing to do with triodes.

9. PARSING

Considerable experience has been gained with the use of triodic parsers. From the point of view of an applications program, of course, any sequential input file is a sentence in some language, to be interpretively executed; the concept of parsing is not limited to overtly linguistic problems.

The key concept for building triode-based parsers seems to be the following. We say a triode **T** *gets* a language L if the following conditions are met.

(i) If the 'input tape' contains an instance of a sentence in L, starting from the current position of the read head, then **T** advances the read head to the symbol following this sentence, and terminates in main exit.
(ii) If not, then **T** does not advance read head (or having advanced it restores it to its original position) and terminates in alternative exit.

The idea behind this is that if one parsing attempt fails then we are all set to try again with a different parse.

In practice, condition (ii) is not always attainable and we may have to settle for the weaker condition:

(ii′) if not, then *either* **T** does not advance read head, and terminates in alternative exit; *or* **T** writes an error message, after which its behaviour is allowed to be undefined, but (we hope) not too silly.

Suppose now that the triode **T** gets the language L, and the triode **U** gets the language M. Then, modulo quite a number of difficulties, we find that:

T;U gets LM
T # U gets L ∪ M
do T od gets L*
if T fi gets L ∪ null

We can clearly build triodes to get individual symbols. (In practice we parameterize so some of these triodes can get a symbol class.) By induction, then, we can build a triode to get any regular language (i.e. Type 3 language—see Ref. 7). Furthermore, the structure of this triode will exactly model the structure of some regular expression that describes the language. This feature is clearly an asset when the language changes and the program has to be maintained. Of course we are not actually limited to regular languages, since we may store some context on a stack, or arrange to call triodes recursively.

Now for the difficulties. The two most serious are those associated with the semicolon construct and with null strings. Both difficulties seem easier to solve in practice than in theory; perhaps because one wants to write parsers for fairly 'sensible' kinds of language. Only the first-mentioned difficulty, that associated with the semicolon construct, will be discussed here.

The statement above, that **T;U** gets LM, is false. If **T;U** finds a sentence of L not followed by a sentence of M then after alternative exit from **T;U** the read head is in the wrong place. We must use a save-restore mechanism (a 'movable bookmark') to get the read head back to its starting point. Practical save-restore mechanisms are limited to a buffer of finite length. An approach that works is to design the input language to consist of 'commands' each of which may at least be disambiguated without having to read beyond an end-of-line. One line at a time is held in the buffer. Now, up through the levels of character, token, command, and file we use the save-restore mechanism to try alternative parsings within the limits of the current line. Any ambiguity that cannot be resolved within the current line must be due to an input error, and we implement condition (ii′) in whatever way seems best.

It should be clear that from the point of view of linguistic theory the triodic parsing method has no great power: a finite-state machine with a 2-way read head and a finite number of movable bookmarks can still only recognize a regular language. Nevertheless, from a practical point of view a triodic parser delivers a great deal of power for not very much effort. Parsers designed in this way are in my experience sturdy, efficient, pleasant to write, and easy to modify. Some examples follow of the use of these techniques.

Example 1

We have already written the following modules.

triode *getchar* (*buffer*, *readposition*, *char*): get next character from buffer if and only if it is identical to the character passed as char.

triode *geteol* (*buffer*, *readposition*, *fileid*): get next token from buffer if and only if it is an end-of-line

possibly preceded by blanks; and if so, refill buffer from file specified by fileid.

triode *getinteger* (*buffer*, *readposition*, *integer*): get next token from buffer if and only if it is a string of digits, possibly with leading blanks, that represents a legal (non-overflowing) integer; if so, return value in integer, otherwise leave passed value of integer intact.

We can now write:

```
triode getkeyword (buffer, readposition, keyword,
      keywordlength):
   ! get next token from buffer if and only if it is
   ! identical, except possibly for added leading
   ! blanks, to the character-string passed as
   ! keyword.
savedposition := readposition
do (getchar (buffer, readposition, 'b')) od
tri
   for i =: 1 ≤ keywordlength
      exec (getchar (buffer, readposition, keyword (i)))
   alldone
#
   readposition := savedposition
out
```

(**Note**: for translation of this module to some target languages, e.g. many versions of Fortran, the keyword will need to be unpacked before its individual characters *keyword* (*i*) can be made accessible).

```
triode getpartnoscmd (buffer, readposition,
      partnosarray, partnosarraylength)
   ! get command of the form
   ! PARTS p NUMBERS n1 n2 . . . np
savedposition := readposition
tri
   (getkeyword (buffer, readposition, 'PARTSb', 6))
   (getinteger (buffer, readposition,
      partnosarraylength))
   (getkeyword (buffer, readposition, 'NUMBERSb',
      8))
   for i =: 1 ≤ partnosarraylength
      exec (getinteger (buffer, readposition,
         partnosarray (i)))
   alldone
   (geteol (buffer, readposition, fileid))
#
   readposition := savedposition
out
```

Example 2

Schematic program to recognize and execute the language {command 1|command 2|. . . command *n*}* endoffile

```
do (getcmd1(parms)); handle cmd of type 1
# (getcmd2(parms)); handle cmd of type 2
   :
# (getcmdn(parms)); handle cmd of type n
od   ! No more valid commands left, so we must have
     ! reached either end of file or an error
if (getendoffile)
# write message 'error in input at line number . . .'
fi
```

10. CONCLUSIONS

The triodic concept leads to a sound and effective method of program design. A number of operations on triodes are described in this paper. The notation used for these operations is a generalization of existing structured programming notations, and is suitable as a source text for mechanical or manual translation to a language such as Fortran.

The constructs **test (B), T;U, T # U, rev T ver, do T od, if T fi, tri T irt, tri T # D out**, together with diodic and triodic procedure calls, and **halt**, provide an excellent base for designing well-structured programs. Once experience is gained in using these constructs, the use of 'gotos' or 'breaks' is seldom necessary. This remains true even when the program includes error-handling.

The further constructs **find Q found** and **for Q exec T alldone** form a desirable addition to the language, though equivalent constructs can instead be written less concisely using **do . . . od** loops.

The methods described in this paper have been developed and used by the author for the past two years when designing a variety of systems and applications programs in the Operational Research Executive of the National Coal Board. It has been found that programs designed in this way are concise to write, quick to debug, efficient, and easy to modify.

Acknowledgements

I am grateful particularly: to E. W. Dijkstra, whose writings stimulated my interest in the first place; to Mr George Mitchell, Director of the Operational Research Executive, for permission to publish this work; to the referee for his helpful comments on the presentation of this paper; to my colleagues in O.R.E. and to Dr Gautam Mitra of Brunel University for their interest and encouragement.

REFERENCES

1. E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey (1976).
2. M. A. Jackson, *Principles of Program Design*. Academic Press, New York (1975).
3. R. Bornat, *Understanding and Writing Compilers*. Macmillan, London (1979).
4. B. W. Kernighan and P. J. Plauger, *Software Tools*. Addison-Wesley, Reading, Massachusetts (1976).
5. M. Richards and C. Whitby-Stevens, *BCPL—the Language and its compiler*. Cambridge University Press, Cambridge (1979).
6. R. W. Floyd, Assigning meanings to programs. *Mathematical Aspects of Computer Science* **19**, 19–32 (1967).
7. J. E. Hopcroft and J. D. Ullman, *Formal Languages and their Relation to Automata*. Addison-Wesley, Reading, Massachusetts (1969).

Received April 1981

APPENDIX 1. UNREACHABLE CODE

Two types of problem will be encountered relating to 'unreachable code'.

(i) The detection of unreachable parts of the source text. For example if **D** is a diode then **T** is unreachable in **if D # T fi**. Such a program was probably written in error (but it could be a skeleton intended for later enhancement), and the translator should issue a warning.
(ii) The avoidance of unreachable text in the target language, even though all the source text is reachable. This happens for example if Table 2 is used to translate **if** (B); **halt # T fi**. A redundant and unreachable goto statement is produced. Typical Fortran compilers produce correct code from this but complain about it.

One's objectives in doing something about this will typically be the following:

(a) The Fortran translation should avoid any obvious redundancies or stupidities.
(b) The translator should produce warning messages if the source text is not correct.
(c) A correct source text should produce Fortran that compiles as intended without triggering warning messages from the compiler.

A summary will be given here and in Appendix 2 of a method that appears to meet all these objectives to an acceptable standard.

1. Attributes of labels

Stack of attributes corresponds to stack of labels; can be coded in sign of label.

When carrying out the following stack instructions:

(i) ext: label created is given attribute URL ('unreachable label').
(ii) goto, cgoto: label accessed has URL attribute removed.
(iii) copy: pair of duplicate labels given URL attribute in (top), not-URL in (top -1).
(iv) swap: attributes are swapped along with labels.

2. Attribute of current text position

(i) After translating a monode, set current text attribute to URTX ('unreachable text').
(ii) After translating a ⟨label command⟩ in source text, or after executing label as a stack instruction **except** in the ext; label; ext sequence that translates **do**: remove URTX attribute if present.
(iii) If any body text (i.e. not a triodic operator) other than a ⟨label command⟩ is translated while URTX set, then issue warning message 'text unreachable after a halt or goto or after a construct that contained a diode where triode expected'.

3. Modifications to stack instruction sequence if attributes set

(i) Stack instruction: goto
Attribute: URTX
Modification: omit goto instruction
(ii) Stack instruction sequence: label; pop
Attribute: URL
Modification: omit label instruction; set URTX attribute

If this occurs while translating:	**then instead:**
fi (either half)	omit label; don't touch URTX

(iii) Stack instruction: pop (not preceded by label)
Attribute: URL
Modification: except when this occurs while translating **out** keyword, issue warning message 'diode found where triode expected'.

This description of a method for handling unreachable code has assumed that these actions by the translator are additional to the enforcement (by error messages) of the grammar shown in Table 4. This enforcement can be managed by a keyword stack that runs in parallel with the label stack.

APPENDIX 2. DEFERRED LABEL GENERATION

The translating of **irt** and **alldone** keywords by the stack instruction sequences of Tables 2 and 3 is liable in certain circumstances to produce a translation containing alternate labels and gotos in a pattern such as:

```
    :
9 CONTINUE
  GOTO 7
8 CONTINUE
  GOTO 6
7 CONTINUE
  GOTO 5
6 CONTINUE
    :
```

which is obviously objectionable. The translator can improve on this by deferred label generation, which works on the following principle.

The instruction sequence 'goto; label' is replaced by 'goto; deflab'. The deflab instruction stores the value of a label but does not write it. Subsequently translation proceeds normally until an instruction, 'x' say, is encountered that calls for writing more target text. If instruction 'x' is a 'goto', and its relevant label (top -1 of stack) has URL attribute set, then this label is replaced in the stack by the stored deflab value: this completes the execution of both 'deflab' and 'x'. In all other circumstances the deferred label is firstly written to the target text and then translation of 'x' continues normally.