Programming Languages

Multiple Exits from a Loop Without the GOTO

G.V. Bochmann
University of Montreal

Key Words and Phrases: control structures, goto free programming, multiple exits from loops, exit statement
CR Categories: 4.20, 4.22

For several years, there has been discussion about the use of the *goto* statement in programming languages [1, 2]. It has been pointed out that *goto* free programs tend to be easier to understand, allow better optimization by the compiler, and are better suited for an eventual proof of correctness. On the other hand, the *goto* statement is a flexible tool for many programmers. Most programming languages have constructs which allow the programmer to write control flows that occur frequently without the use of a *goto*. In particular, the language Pascal [3] contains, besides the *goto*, the following control structures: *if-then-else*, *case*, *while-do*, *repeat-until*, stepping loop. Wulf [4] has described the use of the construct "*leave* ⟨label⟩" in the language Bliss, where ⟨label⟩ is the name of a program section which is exited when the statement is executed. It is important to note that these constructs are invented to describe control flows that occur frequently in programs. They describe the flow on a higher level [5] than an equivalent construction using a *goto* would do.

We propose here another construct, which, outside the body of a loop, distinguishes between normal and abnormal termination of the loop. This kind of control flow is realized in many programming problems, such as exits on error conditions or search algorithms. We consider the following example: Given a vector V of N integers, and an integer I; if I is one of the integers in V, print its order in V; otherwise print "not in V". The flow diagram of Figure 1 solves the problem. It is equivalent to the following program section, which uses a Boolean state variable FLAG for a test after the exit from the loop. (A theoretical discussion of this method can be found in [4 and 6]):

```
FLAG := false ; K := 1 ;
while (K ≤ N) ∧ ¬FLAG do
  begin FLAG := (I = V[K]) ; K := K + 1 end ;
if FLAG then write ('order =', K−1)
    else write ('not in V')
```

Author's address: Department d'Informatique, University of Montreal P.O. Box 6128, Montreal 101, Quebec, Canada.

Another solution is given by the following program section:

```
for K := 1 step 1 until N do
  if I = V[K] then begin write ('order =', K) ;
                         goto CONTINUE end ;
write ('not in V') ;
CONTINUE :
```

These solutions do not quite reflect the simple structure of the flow diagram of Figure 1, and the use of a new construct which is natural to the problem would be appropriate.

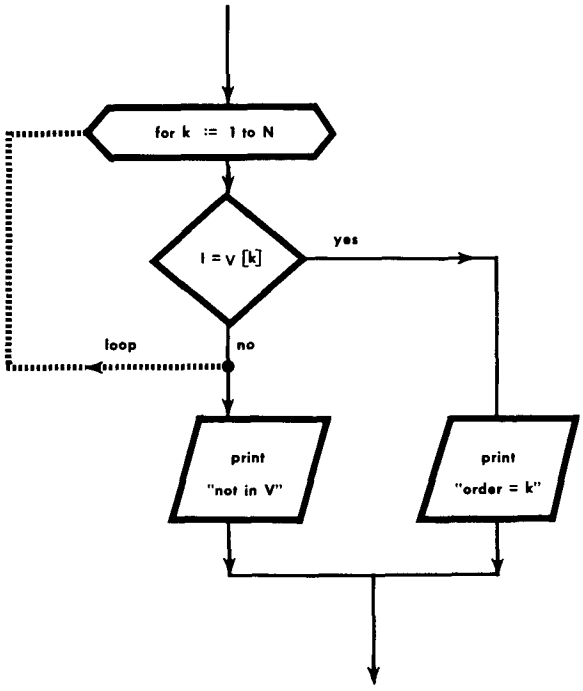
```
for K := 1 step 1 until N do if I = V[K] then exitloop
    exited : write ('order =', K)
    ended : write ('not in V')
```

A more general construct is the multiple exit loop. It can be defined by the following syntax:

```
⟨exit statement⟩ ::= exitloop ⟨label⟩.
⟨loop statement⟩ ::= ⟨simple loop⟩ | ⟨multiple exit loop⟩
⟨multiple exit loop⟩ ::= ⟨simple loop⟩
                         ⟨label⟩ : ⟨statement⟩
                         ...     ...
                         ⟨label⟩ : ⟨statement⟩
                         ended : ⟨statement⟩
```

where the labels of the multiple exit loop must correspond to the labels of the exit statements that occur within the body of the simple loop.

Fig. 1.



The meaning of this construct is that the statement following **ended** is executed after the ⟨simple loop⟩ is terminated in the normal way, whereas the statement following one of the labels is executed after an ⟨exist statement⟩ with the corresponding label, executed within the ⟨simple loop⟩, thus terminating the loop in an abnormal way.

This construct is specially suited to situations where a loop can have several different exit points, for instance whenever some error condition occurs. Compared

with the unrestricted use of *goto* statements, the multiple exit loop imposes a constraint on the control flow. This constraint is given by the fact that the jumps generated by the ⟨exit statements⟩ are jumps to the outside of the ⟨simple loop⟩, which entail the execution of a ⟨statement⟩ before leaving the ⟨multiple exit loop⟩ construct as a whole. Because of this constraint the use of a multiple exit loop results in the following advantages over the use of *goto* statements. (a) Code optimization is easier because it is not necessary to analyze the control flow and loop structure (due to *goto* statements) of a program section, since the structure is given by the multiple exit loop statement. (b) A relatively simple method of correctness proof for a multiple exit loop has been given in [7]. As pointed out there, a correctness proof for programs with unrestricted use of *goto* statements can get quite complicated.

A similar control structure can be used for abnormal exits from procedures. We leave to the calling program to specify the actions to be executed in case of an abnormal termination of the procedure call due to the execution of a ⟨procedure exit statement⟩ ::= **exit** ⟨label⟩ within the called procedure. The multiple exit call statement then has the form:

⟨procedure call statement⟩ ::= ⟨procedure identifier⟩
⟨parameter list⟩ **exits**
⟨label⟩ : ⟨statement⟩
...
⟨label⟩ : ⟨statement⟩
ended : ⟨statement⟩

This structure can, for example, be used to specify the actions to be executed when the called procedure has found some error condition which it cannot deal with by itself.

The multiple exit loop construct is a control structure which appears frequently in programming problems and which otherwise can only be realized using state variables and/or *goto* statements. Using this construct clarifies the program structure, and allows easy correctness proofs as well as code optimization.

Acknowledgment. I am grateful to the referees for several valuable suggestions.

Received October 1972; revised February 1973

References

1. Dijkstra, E.W. Go to statement considered harmful. Letter to the Editor. *Comm. ACM 11*, 3 (Mar. 1968), 147–148.
2. SIGPLAN 4. The go to controversy. Proc. ACM Ann. Conf., 1972, ACM, New York.
3. Wirth, N. The programming language Pascal. *Acta Informatica 1* (1971), 35–63.
4. Wulf, W.A. Programming without the go to. Proc. 1971 IFIP Congress, Ljubljana, Yugovlania, Aug. 1971.
5. Dijkstra, E.W. Notes on structured programming, EWD 249, Technical U., Eindhoven, Netherlands, 1969.
6. Knuth, D.E., and Floyd, R.W. Notes on avoiding "goto" statements. In *Information Processing Letters 1*. North-Holland Pub. Co., Amsterdam, 1971, pp. 23–31.
7. Clint, M., and Hoare, C.A.R. Program proving: jumps and functions. *Acta Informatica 1* (1972), 214.

Equivalence Between AND / OR Graphs and Context-Free Grammars

Patrick A.V. Hall
The City University, London

Key Words and Phrases: artificial intelligence, AND/OR graphs, language theory, context-free grammars
CR Categories: 3.60, 3.64, 5.23

Recent research in artificial intelligence has led to AND/OR graphs as a model of problem decomposition (Nilsson [3]; Simon and Lee [4]). However, AND/OR graphs of a restricted type are equivalent to context-free grammars. This can be set-up formally (the beginnings of a formalism of AND/OR graphs is contained in [4]), but the formalism is so obvious that a brief discussion and example suffice.

To see that an arbitrary context-free grammar is equivalent to an AND/OR graph, see Figure 1. A grammar presented in the formalism of Hopcroft and Ullman [2] is displayed together with the corresponding AND/OR graph. It is this graph which is often explicitly represented in computer storage when it is known as a "syntax graph" (e.g. Gries [1]). The AND/OR graph is alternating, in that AND nodes lead to OR nodes, and vice versa, and is ordered, in that the edges leaving the AND nodes have an ordering on them which is significant for the *language* but *not* the grammar.

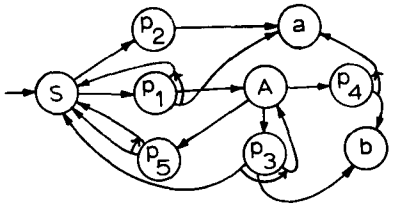
Fig. 1. A context-free grammar and the equivalent AND/OR graph (b) has an AND node for each production and an OR node for each variable, with edges from variable nodes to production nodes indicating choice of substitution for the variable, and edges from production nodes to variable giving an actual substitution.

(a)

$G = (\{S,A\}, \{a,b\}, P,S)$

$P = \{p_1: S \to aAS,\ p_2: S \to a,\ p_3: A \to SbA,\ p_4: A \to ba,\ p_5: A \to SS\}.$

(b)



Conversely, any finite AND/OR graph in which the AND/OR nodes alternate can be set equivalent to a context-free grammar. This involves imposing an ordering on the edges leaving the AND nodes: this ordering is

Author's address: Department of Mathematics, The City University, St. John Street, London, EC1, England.