Christopher Wood
Reflection Essay #4

The case against the `goto` statement seems to be one of perspective. This language construct grew in popularity because of its direct mapping to hardware behavior, but as programming languages and software engineering practices evolved, its usage has slowly withered. Based on the arguments presented by Dijkstra [1] and others, there are many reasons that contribute to the fall of the `goto` statement, most of which center on program correctness proofs through sequencing. However, when considering the use of the `goto` statement outside of this perspective, there were some valid reasons for which it can a useful programming tool. However, historical trends show that these additional perspectives do not carry as much weight as the one promoted by Dijkstra and have, to a large extent, become nonexistent.

At a behavioral level, the `goto` statement enables programmers to introduce arbitrary jumps across local and non-local control flows in a program. In an effort to verify the program's correctness by generating a sequence of statements that are executed to yeild a specific result, these jumps make the analysis stage very difficult (especially when loops and other iterative constructs were used). Not only does it obscure the control flow of a program by branching from the traditional sequential flow of well-structured programs, but it requires the context of a sequence index in a program to be considered in order to make guarantees about its behavior. This inclusion of this context invalidates the use of traditional induction as a technique for proving program correctness. Induction traditionally relies on a well-defined sequence of statements that made up a program, and the induction hypothesis for the behavior of a program is formulated around the sequence index of a program. With multiple contexts for a given sequence index, it is impossible to guarantee the correctness of the program for all such contexts. Therefore, either induction must be abandoned, or it must be modified (perhaps by formulating the correctness using double induction on the sequence index and context).

Other arguments against the `goto` statement are not as substantial. For example, technically speaking, it is not needed in any context. Some may argue that programmer benefits such as the ability to manually construct loops, break out of iterative constructs, and formulate custom conditional statements are lost without the `goto` statement, leading to a Turing Tarpit situation. However, programming languages have undergone significant evolution and introduced more intuitive loop signatures, break or escape statements, and flexible conditional statements to address these issues. Therefore, one might say that the evolution of programming languages invalidates any claim that `goto` statements provide specific implementation benefits.

Furthermore, the primitive programming techniques that came with the `goto` statement do not appear to be sound development practices and are frowned upon in the software engineering community. In fact, one of my former managers at Intel once said, "If you're using a `goto` statement, then you've got bigger issues to deal with than rewriting your function." I believe he and the community are partially correct. There are certainly cases where the use of `goto` statements is a dying practice, and mixing it with modern development techniques only causes developer confusion and makes it much more difficult to maintain code and ensure its correctness. However, consider the case where we have a simple procedure that performs some form of error checking in it. The typical approach to protect the main body of the procedure from running without passing all of the error checks is to used nested `if` statements. In this case though `goto` statements would be perfectly acceptable, because they would enable the developer to jump to a common error handling routine in the event of an error, thus skipping all subsequent error checks and the loop body altogether.

This is an example of where the use of the `goto` statement led to cleaner, smaller, and more efficient code. However, in his paper entitled "Structured Programming with go to Statements," Donald Knuth was able to generate many different examples where the `goto` statement actually increased code size and decreased performance [2]. From this, and the aforementioned error-checking example, we can conclude that careful usage of `goto` statements can be beneficial in certain circumstances and harmful in others.

Collectively, there are several perspectives in which the `goto` statement can be considered harmful. While old development practices might provide adequate reason to support the usage of this statement, the evolution of software engineering practices and programming languages has started to invalidate these claims. To the experienced programmer, the `goto` statement certainly has its benefits. However, as the software development field changes and this statement is phased out in modern languages, it is important that these benefits are abandoned in order to establish a common set of development practices and techniques among all active developers.

## References

[1] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, March 1968.

[2] Donald E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, December 1974.