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. Implementation benefits related to code quality and performance that come from the proper usage of the `goto` statement have been dwarfed by such advancements. In addition, based on the arguments presented by Dijkstra [1] and others, there are many other reasons that contribute to the fall of the `goto` statement. The most compelling of which centers on proofs of program correctness. Therefore, in my opinion, what remains in support for the `goto` statement are few scenarios where skillful usage is needed to yield any benefits.

At a behavioral level, the `goto` statement enables programmers to introduce arbitrary jumps across local and non-local control flows in a program. These jumps make the formal modeling of program behavior 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 programming, but it requires the context of program sequence index to be considered in order to make guarantees about its behavior [1]. The inclusion of this context invalidates the use of mathematical induction as a technique for proving program correctness. Such proofs typically rely on a well-defined sequence of statements that make up a program, and the induction hypothesis for the behavior of a program is formulated around the sequence index of a program. Unfortunately, it is difficult to formalize such a proof with this context variability. 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) if the `goto` statement is permitted.

Other arguments against the `goto` statement are not as theoretically fundamental. For example, some people 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 quality benefits. That is, by and large, `goto`s used for these reasons are no longer needed, so it's best not to include them in modern languages.

Furthermore, the primitive programming techniques that came with the `goto` statement do not appear to be sound and respectable development practices in today's 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 to protect an internal block of code. The typical approach to protect the main body of the procedure from running without passing all of the

error checks is to use nested conditional statements. Depending on the level of nesting that is needed, such code will usually end up being difficult to read and maintain. In this case the `goto` statement would be perfectly acceptable because it would enable the developer to jump to a common error handling block in the event of an error, thus skipping all subsequent error checks and the protected body altogether. In addition, no nesting would be needed with this design. This is just one example of where the use of the `goto` statement can lead 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 (e.g. hash code generation and tree searching) [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.

In conclusion, while old development practices might provide adequate reason to support the usage of this statement, the evolution of software engineering practices (including the area of formal modeling) and programming languages has started to refute these claims. To the experienced programmer, however, the `goto` statement certainly has its benefits in certain scenarios. Unfortunately, it is necessary to abandon these benefits in order to establish a common set of development practices and techniques among all active developers. Similar to the slow transition away from Windows XP in corporate America, such change is needed to enable the efficiencies and higher quality that comes with new technologies (which are, in this case, newer languages and development practices).

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.