BIT 12 (1972), 334–341

# A NOTE ON THE FOR STATEMENT

C. A. R. HOARE

**Abstract.**

This note discusses methods of defining the for statement in high level languages and suggests a proof rule intended to reflect the proper role of a for statement in computer programming. It concludes with a suggestion for possible generalisation.

## 1. Definition by substitution rule.

In most high-level programming languages, the for statement is described as a useful abbreviation for a piece of program which could equally well be substituted for it. For example, the for statement can be defined by stating that the form

"**for** $x := a$ **to** $b$ **do** $Q$"

is equivalent to

"$x := a$;
**while** $x \leqq b$ **do** $\{Q; x := succ(x)\}$"

and

"**for** $x := b$ **down to** $a$ **do** $Q$"

is equivalent to

"$x := b$;
**while** $a \leqq x$ **do** $\{Q; x := pred(x)\}$",

where *succ* and *pred* give the successor and predecessor of their arguments. The notations are borrowed from PASCAL [1].

One of the purposes of singling out a particular common construction and defining an abbreviation for it is to signal a special case which may be susceptible of more efficient machine implementation. For example, in some machines it may be more efficient to implement the for statement

"**for** $x := a$ **to** $b$ **do** $Q$"

Received February 29, 1972.



in a manner more similar to

"$x := pred(a)$;
**while** $x < b$ **do** $\{x := succ(x); Q\}$"

In order to permit this kind of machine-dependent optimisation, ALGOL 60 and FORTRAN state that the value of the counting variable $x$ is undefined on normal exit from the for statement. There is also a restriction against jumping into the middle of a for statement from outside. These riders spoil the simplicity of the original method of defining the for statement.

A remedy to the defect is adopted in ALGOL W [2] and ALGOL 68 [3], which define the counting variable to be local to the for statement, which is accordingly defined as equivalent to:

**begin integer** $x$;
$x := a$;
**while** $x \leqq b$ **do** $\{Q; x := succ(x)\}$
**end**

Then the fact that $x$ is undefined on exit is a natural consequence of the normal scope rules, and so is the prohibition against jumping into the middle. The localisation of $x$ also seems to express more clearly the programmer's intention of using $x$ as a counting variable; and it may make it easier for an implementation to use special registers to store its value and achieve additional efficiency. At least, the implementor would not have to take special action on exit of the loop by a jump, as he does in the case of FORTRAN and ALGOL 60. An inevitable consequence of this view is that the controlled variable must be a simple unsubscripted variable, a restriction which must recommend itself to all implementors of the language, if not to all users.

In ALGOL 60 and many subsequent languages the bounds $a$ and $b$ of the counting variable may be specified by general expressions. If the definition given above is adopted, the expression for $b$ would have to be re-evaluated in order to make the test "$x \leqq b$" in each iteration of the loop. For this reason, ALGOL 68 specifies that the value of the limit shall be computed once and for all before the iteration starts. Thus the effect of a for statement is defined:

**begin integer** $x, b'$;
$x := a$;
$b' := b$;
**while** $x \leqq b'$ **do** $\{Q; x := succ(x)\}$
**end**

As an alternative to this more complicated definition, it is possible to forbid alteration of the limit $b$ from within the loop body $Q$, as is done in PASCAL. This has the additional advantage that if $b$ is a simple variable, the implementor is not forced to take an extra copy of its value.

But unfortunately there still remains a serious conceptual defect, that the counting variable takes on one value more than it should. This defect is clearly illustrated in PASCAL by the case in which a counting variable is expected to range over the whole of a type defined by enumeration. Then the final execution of:

$$x := succ(x)$$

is actually illegal, since it takes the value of $x$ beyond its permitted range. This defect could be remedied by a more complicated definition of the for statement:

**begin integer** $x, b'$;
  $x := a$; $b' := b$;
  **while** $x < b'$ **do** $\{Q; x := succ(x)\}$;
  $Q$
**end**

In PASCAL itself the defect is avoided in a similar fashion. But by this stage the definition by substitution has become rather complicated, and it no longer expresses clearly the intention of the for statement, or the role which it plays in computer programming.

## 2. Definition by proof rule.

Apart from indicating the possibility of efficient implementation, there is a theory that a high-level language feature should also simplify the task of proving the correctness of programs expressed in the language. Even if few programs actually undergo formal proof, this will make it easier to "see" the correctness of a program, with less risk of unexpected conditions arising to invalidate it. So it seems worth while to investigate the simplest possible proof rules for any proposed language feature, and in particular the for statement.

The most obvious simplification in proof methods for a for statement (as compared with a while statement) is that there is no need for an explicit proof of termination. The counting variable can only take on a finite range of values, and when the range is exhausted the loop terminates. But this simplification is valid only if no assignment is made to the counting variable from within the body of the loop. This restriction



seems to be a genuine advantage to the writers and readers of programs, and has been incorporated in ALGOL W, ALGOL 68, and PASCAL.

A second great simplification occurs when the counting variable is specified as a type or subrange of a type which is also the subscript range of an array referenced from within the body of the loop, using the counting variable as a subscript. In this case it is logically impossible that the subscript should be out of range when it is used, and there is no need for the programmer to make an explicit proof of the validity of the reference. Furthermore, the validity can be enforced by a compile-time check, thereby obviating the expense of a run-time subscript check on each iteration of the loop. The most that is required is that the values of $a$ and $b$ be checked before entry to the loop, and often even this is unnecessary. But again, the validity of this simplification depends on the fact that the counting variable is not changed from within the loop. For further elaboration (even over-elaboration) of this idea, see [8].

In order to express the other aspects of correctness proofs of for statements we introduce the following notations for open and closed intervals:

$$\begin{aligned}
[a \ldots b] &= \{i \mid a \leqq i \leqq b\} \text{ (closed)} \\
[a \ldots x) &= \{i \mid a \leqq i < x\} \text{ (open at top)} \\
(x \ldots b] &= \{i \mid x < i \leqq b\} \text{ (open at bottom)} \\
[\,] &= \text{the empty set}
\end{aligned}$$

Note that $[a \ldots a) = (b \ldots b] = [\,]$.

Let $I(s)$ be an assertion about the interval $s$. Let $Q$ be the body of a loop, which has the property that if $I$ is true of an open interval $[a \ldots x)$ before execution it will be true of the closed interval $[a \ldots x]$ on termination.

Now suppose that $I$ is true of the empty set $(= [a \ldots a))$ before embarking on the for statement. After the first iteration it will be true of $[a \ldots a] = [a \ldots succ(a))$. On completion of the second iteration (if any) it will be true of

$$[a \ldots x] = [a \ldots succ(a)] = [a \ldots succ(succ(a)))];$$

and so on until finally it is true of $[a \ldots b]$. This proof rule may be expressed formally using the notation of [4]*:

$$\frac{a \leqq x \leqq b \ \&\ I([a \ldots x))\{Q\}\ I([a \ldots x])}{I([\,])\ \{\textbf{for}\ x := a\ \textbf{to}\ b\ \textbf{do}\ Q\}\ I([a \ldots b])}.$$

* The notation $P\{Q\}R$ may be interpreted as "if $P$ is a true assertion about the values of the program variables before entry to the piece of program $Q$, then $R$ will be a true assertion after exit from $Q$". In a proof rule the formula above the line expresses what has to be proved in order to establish the validity of the formula below the line.

Similar reasoning leads to the proof rule for **down to**:

$$\frac{a \leqq x \leqq b \ \&\ I((x \ldots b])\{Q\}\ I([x \ldots b])}{I([])\ \{\textbf{for}\ x := b\ \textbf{down to}\ a\ \textbf{do}\ Q\}\ I([a \ldots b])}$$

These rules are fairly simple, but their validity depends on the observance of yet further restrictions in the body of the loop, in that not only $x$ but also $a$ and $b$ must remain unchanged throughout the loop. Furthermore, the invariant $I$ must not contain the variable $x$. If these restrictions are violated the proof rule would become more complicated, and eventually would equal in complexity the proof procedures required for the alternative method of achieving the same effect using a **while**.

Even though the rules suggested above are more restrictive than those imposed by any currently fashionable programming language, it is probable that the vast majority of loops expressed in them conform to the restrictions; and those that do not would probably be more clearly expressed as a while loop anyway. It is this clarity of expression which is a far stronger motivation for observing the restrictions than the marginal possibility of better implementation. Surely the restrictions will be felt a burden only by those programmers whose delight it is to use their programming tools for purposes for which they were not originally intended.

### 3. Generalisation.

The for statement as described above deals only with the case where the counting variables range over a set of consecutive integers, or elements of an enumeration. Many languages also permit an arbitrary arithmetic progression. Although these cases are very common, it is well worth while to enquire whether the simplicity of the proof rules for for statements can be extended to more general kinds of progressions, and more general types of counting variables. For example, in normal mathematical notation, a counting variable can range over members of arbitrary sets or sequences; and in a language like PASCAL, which permits sets and sequences as data types, it is very attractive to attempt the same generalisation. In both cases it seems very likely that the implementor can achieve extra efficiency in a machine-dependent way, and the programmer can simplify his proofs.

Consider, for example, a sequence (file) $s$; we can introduce the notation:

**for** $x$ **in** $s$ **do** $Q$,

as a for statement which causes the counting variable $x$ to take in turn



the value of each successive item of the sequence $s$, and perform the action $Q$ upon it. If $s$ is empty, no action takes place.

The proof rule for such a for statement is slightly more complicated:

$$\frac{s = s_1{}^\frown[x]{}^\frown s_2 \ \&\ I(s_1)\{Q\}I(s_1{}^\frown[x])}{I([])\{\textbf{for}\ x\ \textbf{in}\ s\ \textbf{do}\ Q\}I(s)}.$$

where

$^\frown$ is the sign of concatenation
$s_1, s_2$ are arbitrary fresh variables, not appearing except as shown above
$[x]$ is the sequence whose only item is $x$
$[]$ is the empty sequence.

Again, the validity of the rule depends on not changing $s$ or $x$ within the loop body $Q$.

In the case of a set $s$, we can formulate the rule:

$$\frac{s_1 \subset s \ \&\ x \in (s - s_1) \ \&\ I(s_1)\{Q\}I(s_1 \cup \{x\})}{I(\{\})\ \{\textbf{for}\ x\ \textbf{in}\ s\ \textbf{do}\ Q\}I(s)},$$

where

$s_1$ is an arbitrary fresh variable, not appearing except as shown
$\{x\}$ is the unit set of $x$
$\{\}$ is the empty set.

Again, the validity of the rule depends on not changing $x$ or $s$ from within $Q$. But it is most interesting to note that the validity of the rule does not depend on the *ordering* of the set of values taken by $x$; the values may be processed in arbitrary order. Thus if the proof rule given above is taken as the definition of the meaning of the for statement, we have found a method of defining a language which does not force the programmer to express decisions about the order of execution of his program which are perhaps for his current purposes irrelevant. The need for avoiding such irrelevant decisions has been expressed by C. Strachey.

### 4. Conclusion.

It has been suggested [4, 5] that specification of rules for the proof of correctness of programs would throw light on good methods for language design; and this note has attempted to illustrate the suggestion by its application to the for statement. It has been revealed that the observance of certain programming disciplines is of simultaneous benefit to the quality of implementation and to the writers and readers of pro-

grams. The reconciliation of the needs of the implementor and user is the highest goal of the language designer.

In the case of the for statement it can be argued that the benefits are quite small, and do not justify the inclusion of this feature in a high-level programming language. On the other hand, if the feature can be very commonly used, it is justified even if the benefits in each case are only marginal.

Many of the ideas of this paper were propounded in the context of ALGOL 60 in [7]. An example of the use of the proof rule, borrowed from [6], is given below.

## Appendix.

PROBLEM. Find the largest value *max* of the elements between $a$ and $b$ of the array $A$, and assign its subscript value to $m$.

The desired result of this program may be formalised as $I([a \ldots b])$, where

$$I(s) = \textbf{if } a \leqq m \leqq b \ \&\ max = A[m] \ \&\ \forall i(i \in s \supset A[i] \leqq max) .$$

The body $Q$ of the loop is

$$\textbf{if } A[x] > max \textbf{ then } \{max := A[x];\ m := x\} \ldots Q .$$

Define

$$\begin{aligned} S \equiv \textbf{if } \ A[x] > max \textbf{ then } & a \leqq x \leqq b \ \&\ A[x] = A[x] \\ & \&\ \forall i(i \in [a \ldots x] \supset A[i] \leqq A[x]) \\ \textbf{else } & I([a \ldots x]) \end{aligned}$$

It is a mechanical matter to verify that

$$S\{Q\}I([a \ldots x]) .$$

We next need to prove the tedious but trivial lemma:

$$a \leqq x \leqq b \ \&\ I([a \ldots x)) \supset S ,$$

and this gives us (by the rule of consequence):

$$a \leqq x \leqq b \ \&\ I([a \ldots x))\{Q\}I([a \ldots x]) .$$

The proof rule for the for statement enables us to conclude

$$I([\,])\{\textbf{for } x := a \textbf{ to } b \textbf{ do } Q\}I([a \ldots b]) .$$

Thus we have shown that

$$I([\,]) \equiv a \leqq m \leqq b \ \&\ max = A[m]$$



is the precondition which we must ensure before entry to the loop. This may obviously be achieved by the initialisation:

$$m := a;\ max := A[a];$$

provided that $a \leqq b$.

## REFERENCES


1. N. Wirth, *The Programming Language PASCAL*, Acta Informatica 1.1 (1971), 35–63.
2. N. Wirth and C. A. R. Hoare, *A contribution to the development of ALGOL*, C.A.C.M. 9.6 (June 1966).
3. A. van Wijngaarden (ed.), *Report on the Algorithmic Language ALGOL 68*, Numerische Mathematik 14 (1969), 79–218.
4. C. A. R. Hoare, *An Axiomatic Approach to Computer Programming*, C.A.C.M. 12. 10 (Oct. 1969), 576–580, 583.
5. R. W. Floyd, *Assigning Meanings to Programs*, Proc. Amer. Math. Soc. Symposium in Applied Mathematics, Vol. 19, pp. 19–31.
6. P. Naur, *Proof of Algorithms by General Snapshots*, BIT 6 (1966), 310–316.
7. C. A. R. Hoare, *Cleaning up the For Statement*, ALGOL Bulletin 21.3.4 (July 1965), 32–35.
8. C. A. R. Hoare, *Subscript Optimisation and Subscript Checking*, ALGOL Bulletin 29.3.6 (Nov. 1968), 33–44.


DEPARTMENT OF COMPUTER SCIENCE
THE QUEEN'S UNIVERSITY OF BELFAST
BELFAST, NORTHERN IRELAND