HOARE72
Hoare, C.A.R.,; A Note on the For Statement; Bit 12, (1972), pp. 334-341

SYNOPSIS The usefulness of the for statement is analysed and discussed by this paper in three aspects: (1) As a mean for more efficient machine implementation (2) As an aid to program verification (3) Its generalization to solve problems involving elements in a set or file. For statement is not only a useful abbreviation for a piece of program code; in some machines, the for statement:

fox x := a to b do Q

can be implemented more efficiently as:

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

where succ and pred give the successor and predecessor
of their arguemnts.

In order to take advantage of this efficient implementation, some languages have to put restrictions on the counting variable x. In ALGOL 60 and FORTRAN, the counting variable x is undefined on normal exit from the for statement. They also prohibit against jumping into the middle of a for statement from outside. Some languages define the counting variable to be local to the for statement. This localization expresses more clearly the programmer's intention of using it as a counting variable. It makes it easier for an implementation to use index or counting registers to store its value and achieve additional efficiency. Some languages forbid the alteration of the limit b from within the loop body Q. This has the additional advantage that the expression for b needs to be evaluated only once before the iteration starts. Apart from the possibility of efficient implementation, for statement also simplifies the task of proving the correctness of programs. When the value of counting variable is not allowed to vary within the loop body, there is no need of an explicit proof of termination. For statement can also obviate the expense of a run-time subscript check when a counting variable is used as a subscript of an array within the loop. There is no need for the programmer to make an explicit proof of the validity of the reference. Only the values of a and b need to be checked before the entry to the loop. If a and b remain unchanged throughout the loop and the value of x is not changed by the loop body, for statement has a very simple proof rule:- Let I(S) be an assertion about an interval. Then a proof rule for the for statement is:

a <= x <= b & I ([a ... x]) {Q} I ([a ... x])

I ([ ]) { for x := a to b do Q } I ([a ... b])

Generalization is possible to extend for statement to range over variables of non-integer types. In a language like PASCAL, which permits sets and sequences as data types, such a generalization looks appealing. To the implementor, the programs written can gain efficiency and to the programmers, their proofs are simplified. Proof rules are given for both sequencing through a file and a set. For the case of a set S, we can formulate the rule:

S1 S & x (S - S1) & I(S1) {Q}I(S1 {x})

I({}) {for x in S do Q} I(S)

where S1 is an arbitrary fresh variable, not
{x} is the unit set of x,
{{ is the empty set.

SUMMARY This paper illustrates how efficient implementation and the specification of proof rules can throw light on good methods for language design. In the case of the for statement, the benefits may be marginal, but the improvement may be enormous if the concept is applied to other major features of

programming languages.