

Establishing P :

Note that $P \Leftarrow (i=n)$:

```
{Q}
i := n;
{ P : (∀j | i < j ≤ n : A[j] ≠ x) }
do A[i] ≠ x → { P ∧ A[i] ≠ x } S {P} od
{P ∧ A[i] = x}
{R}
```

Now we look for a statement S such that

$$\{P \wedge A[i] \neq x\} S \{P\} \quad (*)$$

In general (for any Q , S , and R),

$$\{Q\} S \{R\} = Q \Rightarrow \text{wp } S R$$

so (*) becomes

$$(P \wedge A[i] \neq x) \Rightarrow \text{wp } S P$$

Since S is this formula's only unknown, we can use the formula to derive S .

The derivation of S :

$$\begin{aligned} & (\forall j \mid i < j \leq n : A[j] \neq x) \wedge A[i] \neq x \\ & \quad \Rightarrow \text{wp } S (\forall j \mid i < j \leq n : A[j] \neq x) \\ = & \quad \{ \text{"unsplit" the range} \} \\ & (\forall j \mid i-1 < j \leq n : A[j] \neq x) \\ & \quad \Rightarrow \text{wp } S (\forall j \mid i < j \leq n : A[j] \neq x) \\ = & \quad \{ \text{comparing predicates, let } S = "i := i-1" \} \\ & (\forall j \mid i-1 < j \leq n : A[j] \neq x) \\ & \quad \Rightarrow \text{wp } "i := i-1" (\forall j \mid i < j \leq n : A[j] \neq x) \\ = & \quad \{ \text{definition of "="} \} \\ & (\forall j \mid i-1 < j \leq n : A[j] \neq x) \\ & \quad \Rightarrow (\forall j \mid i < j \leq n : A[j] \neq x)_{i-1}^i \\ = & \quad \{ \text{substitution} \} \\ & (\forall j \mid i-1 < j \leq n : A[j] \neq x) \\ & \quad \Rightarrow (\forall j \mid i-1 < j \leq n : A[j] \neq x) \\ = & \quad \{ p \Rightarrow p = \text{true} \} \\ & \text{true} \end{aligned}$$

and our program is...

The Linear-Search program

```
{Q}
i := n;
{P: (∀j | i < j ≤ n : A[j] ≠ x)}
do A[i] ≠ x → {P ∧ A[i] ≠ x} i := i-1 {P} od
{P ∧ A[i] = x}
{R}
```

Proved so far: **partial correctness**, i.e.,

if the loop terminates,
then our program establishes R .

Total correctness includes **termination**.

To prove termination: Show that every iteration *makes progress* towards a state in which B is false (and hence the repetition ceases).

More precisely: We define a **bound function** t —a function of the program state—and prove:

- *progress*: Each repetition decreases t ;
 $P \wedge B \wedge t=C \Rightarrow \text{wp } S (t < C)$ for some constant C .
- *boundedness*: While repetition continues, t is bounded from below.
 $P \wedge B \Rightarrow t \geq 0$

For linear search,
we derive t from initial condition $x \in A[0 : n]$;
more precisely,

$$(\exists k \mid 0 \leq k \leq n : A[k] = x).$$

Without knowing k 's value, we can define

$$t = i - k.$$

Skipping formalities:

- each iteration decreases i and hence t ;
- while iteration continues, P and B hold; therefore $i > k$ and hence $t > 0$.

Conclusion: Provided Q holds initially, the program terminates in a state satisfying R .

Concluding remarks on formal development:

- we're seeing more of it in "real-world" software (and especially hardware) development
- invariants and bound functions can be used informally

Structured Programming in conventional HLLs

Excludes **goto** in favor of structured control constructs, bringing to control-flow notation the modularity of expression notation.

Structured control constructs all provide *single entry* and *single exit*:

- $S_0 ; S_1 ; S_2 ; \dots ; S_n$
- **if ... fi**
- **do ... od**
- **if c then** S_0 **else** S_1
- **while c do** S
- ... etc.

Control constructs' essential properties

- recursive *nesting* (as in expressions): a compound statement can be a component of a (larger) compound statement, and the same rules apply at every level
- each part of a compound statement can be considered in *isolation* from the other parts.
- the separation of the parts, and their relation to each other and to the whole, are apparent from their written form.
- the properties (i.e., the state-transformation effects) of a compound statement depend only on the properties of its parts

Sequential Composition

- Pascal: **begin** $S_0 ; S_1 ; S_2 ; \dots ; S_n$ **end**
- C: $\{ S_0 ; S_1 ; S_2 ; \dots ; S_n ; \}$
- GCN: $S_0 ; S_1 ; S_2 ; \dots ; S_n$

A command is a function from states to states, i.e., a state-transformation function.

The state-transformation function of a *sequence* of commands is the *composition* of the commands' state-transformation functions.

Selection

- Pascal: **if b then** S_0 **else** S_1
[if true then S_0 **else** S_1 **]** = S_0
[if false then S_0 **else** S_1 **]** = S_1

C: `if(b) S_0 else S_1`

GCN: **if** $b \rightarrow S_0$ **fi** $\neg b \rightarrow S_1$ **fi**

- Pascal: **if b then** S
[if true then S **]** = S
[if false then S **]** = $\langle \text{empty} \rangle$

C: `if(b) S`

GCN: **if** $b \rightarrow S$ **fi** $\neg b \rightarrow$ **skip fi**

- | | |
|------------------------|--|
| • Modula-2 | GCN |
| IF b_0 THEN S_0 | if $b_0 \rightarrow S_0$ |
| ELSIF b_1 THEN S_1 | $\parallel \neg b_0 \wedge b_1 \rightarrow S_1$ |
| ELSIF b_2 THEN S_2 | $\parallel \neg b_0 \wedge \neg b_1 \wedge b_2 \rightarrow S_2$ |
| ELSE S_3 | $\parallel \neg b_0 \wedge \neg b_1 \wedge \neg b_2 \rightarrow S_3$ |
| END | fi |

- | | |
|------------------|--|
| • Pascal | GCN |
| case e of | if $e = c_0 \rightarrow S_0$ |
| $c_0 : S_0$; | $\parallel e = c_1 \rightarrow S_1$ |
| $c_1 : S_1$; | $\parallel e = c_2 \rightarrow S_2$ |
| $c_2 : S_2$ | fi |
| end | |
| | c_i are constants: $i \neq j \Rightarrow c_i \neq c_j$ |
| | no match \Rightarrow abort |

- | | |
|---------------------------|--|
| • C | GCN |
| switch(e) | if $e = c_0 \rightarrow S_0$ |
| { case $c_0 : S_0$ break; | $\parallel e = c_1 \rightarrow S_1$ |
| case $c_1 : S_1$ break; | $\parallel e = c_2 \rightarrow S_2$ |
| case $c_2 : S_2$ | $\parallel e \notin \{c_0, c_1, c_2\} \rightarrow$ skip |
| } | fi |
| | c_i are constants: $i \neq j \Rightarrow c_i \neq c_j$ |

- | | |
|--------------------|--|
| • C | GCN |
| switch(e) | if $e = c_0 \rightarrow S_0 ; S_1 ; S_2$ |
| { case $c_0 : S_0$ | $\parallel e = c_1 \rightarrow S_1 ; S_2$ |
| case $c_1 : S_1$ | $\parallel e = c_2 \rightarrow S_2$ |
| case $c_2 : S_2$ | $\parallel e \notin \{c_0, c_1, c_2\} \rightarrow$ skip |
| } | fi |
| | c_i are constants: $i \neq j \Rightarrow c_i \neq c_j$ |
| | no match \Rightarrow skip ; |
| | no break \Rightarrow "fall-through" |

- | | |
|--------------------|--|
| • C | GCN |
| switch(e) | if $e = c_0 \rightarrow S_0 ; S_1 ; S_2 ; S_3$ |
| { case $c_0 : S_0$ | $\parallel e = c_1 \rightarrow S_1 ; S_2 ; S_3$ |
| case $c_1 : S_1$ | $\parallel e = c_2 \rightarrow S_2 ; S_3$ |
| case $c_2 : S_2$ | $\parallel e \notin \{c_0, c_1, c_2\} \rightarrow S_3$ |
| default: S_3 | fi |
| } | |

- | | |
|---------------------|--|
| • Modula-2 | GCN |
| CASE e OF | if $e = c_0 \rightarrow S_0$ |
| $c_0 : S_0$ | $\parallel e = c_1 \vee e = c_2 \rightarrow S_{12}$ |
| $c_1, c_2 : S_{12}$ | $\parallel e \in (c_3..c_4) \rightarrow S_{34}$ |
| $c_3..c_4 : S_{34}$ | $\parallel e \notin (\{c_0, c_1, c_2\} \cup (c_3..c_4)) \rightarrow S_5$ |
| ELSE S_5 | fi |
| END | |
| | c_i are constants: $i \neq j \Rightarrow c_i \neq c_j$ |