To get you in the right mind frame, we challenge you to develop a winning strategy for a very simple version of the ancient game of Nim.

You start with an arbitrary number of pencils. (or sticks or stones or objects). Two players take turns removing up to three pencils. The person who takes the last pencil(s) wins.

We found the following version of Nim online at wikimedia.org:

Play it, where you make the first move (otherwise, the computer will win!).

- What is the winning strategy?
• How did you figure this out?
• When did you notice that you were going to win or lose?
• Could you figure this out earlier to help guide your moves?
• Did you find that you used a backward analysis?
• How does this relate to goal-oriented programming?
• Can you find a “winning” invariant?

Another algorithm for summing the elements of an array.  The purpose of the first launch in this section, the game of Nim, was to have you notice that sometimes working linearly from beginning to end is not the best way to approach a problem.  A goal-oriented approach, by which we mean starting from where you want to end up, leads to an easier development of a solution.

This suggests that maybe the systematic development of a correct program segment should start with the goal: what is to be computed. Here we challenge you to revisit summing the elements in an array. This was also the goal of the simple loop from the launch of Week 2 where we summed from the first to the last entries in the array. This time, instead of giving you the program segment to prove correct, we challenge you to try to derive another algorithm that instead sums the values in the array starting at the last element and ending at the first. This builds on what you learned in Week 2, yet turns the process around. Think of how assertions might guide you toward commands. You will notice that the algorithm you will derive is similar yet not the same algorithm as you saw before, in Week 2, since the invariant is different.

It has been suggested that you may find it helpful to re-watch the video in the launch of Week 2 that explains the previous algorithm. Then you may want to look at this homework side by side with the complete annotated code for that previous algorithm (which is shown at 2:24 of the video).

In the below homework, you may want to consider these questions to help guide you:

• In what order do you insert the components?
• What do you insert first? Is it the loop guard? Why?
• What do you need to know before you find the loop guard?
• What do you need to know to find the initialization?
• What do you need to know to find the command in the loop?
• Can you suggest what might be a good flow to create assertions and commands for determining the code segment?
• Does it require starting at the top and working you way down?
• Did you get it right the first time?

• Before you checked, how confident were you that your pseudo code was correct?

**Homework 3.1.1.1** Consider the problem of summing the elements of array \( b \), as outlined in Figure 3.1. Place the following assertions in the correct place (the blank boxes) in the algorithm. Some need to be inserted in multiple places. Some are just there to confuse you (and not be used)

1. \( s = (\sum i \mid k \leq i < n : b(i)) \land 0 \leq k \leq n \)
2. \( k < n \)
3. \( 0 \leq k \)
4. \( 0 < k \)
5. \( s := 0 \)
6. \( k = 0 \)
7. \( k := n \)
8. \( s := s + b(k) \)
9. \( s := s + b(k - 1) \)

• In what order did you insert the components?

• Did you get it right the first time?

• Before you checked, how confident were you that your pseudo code was correct?
Figure 3.1: Partially annotated alternative algorithm for computing the sum of the elements of array $b$. 

\[
\{ 0 \leq n \\} \\
\{ \}
\begin{align*}
\text{while} & \quad \text{do} \\
\{ & \}
\text{k := k - 1} \\
\{ & \}
\text{endwhile} \\
\{ & \}
\{ s = (\sum i | 0 \leq i < n : b(i)) \}
\]
3.1. Opening Remarks \(\rightarrow\) to edX

3.1.1. Launch \(\rightarrow\) to edX
3.1.2. Outline Week 3 \(\rightarrow\) to edX
3.1.3. What you will learn \(\rightarrow\) to edX

3.2. Developing Simple Commands \(\rightarrow\) to edX

3.2.1. The \texttt{skip} command \(\rightarrow\) to edX
3.2.2. Assignment to simple variables \(\rightarrow\) to edX
3.2.3. Careful! \(\rightarrow\) to edX
3.2.4. Assignment to array elements \(\rightarrow\) to edX

3.3. Developing the if Command \(\rightarrow\) to edX

3.3.1. A general strategy \(\rightarrow\) to edX
3.3.2. A commonly encountered case \(\rightarrow\) to edX

3.4. Developing a While Command \(\rightarrow\) to edX

3.4.1. A worksheet for the \texttt{while} command \(\rightarrow\) to edX
3.4.2. Progress towards completion \(\rightarrow\) to edX
3.4.3. \textit{A priori} determination of loop invariants \(\rightarrow\) to edX
3.4.4. Deriving the loop guard and initialization command \(\rightarrow\) to edX
3.4.5. Deriving the loop body \(\rightarrow\) to edX

3.5. Examples \(\rightarrow\) to edX

3.5.1. Evaluating a polynomial \(\rightarrow\) to edX
3.5.2. At last, you write your first code! \(\rightarrow\) to edX

3.6. Enrichment \(\rightarrow\) to edX

3.6.1. A conversation with Prof. David Gries \(\rightarrow\) to edX
3.6.2. Dafny: a language and program verifier for functional correctness

3.7. Wrap Up \(\rightarrow\) to edX

3.7.1. Additional exercises \(\rightarrow\) to edX
3.7.2. Summary \(\rightarrow\) to edX
3.7.3. Why Dijkstra received the ACM Turing Award \(\rightarrow\) to edX
3.1.3 What you will learn to edX

This week begins our journey to develop programs that are correct. Starting with the goal and annotating the precondition and postcondition, we uncover strategies to find loop-based algorithms by first systematically deriving loop invariants which then guide the development of the rest of the commands in the loop.

Upon completion of this week, you should be able to

- Use Hoare triples and weakest precondition to determine appropriate assignment commands.
- Reason and apply goal-oriented programming to develop short program segments involving skip, assignments, and conditional branching.
- Develop loop-based algorithms using goal-oriented programming techniques given invariants.
- Determine various invariants for loops traversing one-dimensional arrays.
- Implement your first program.
3.2 Developing Simple Commands to edX

3.2.1 The skip command to edX

Consider the code segment

\[
\begin{align*}
\{ & Q \\
S & \\
\{ & R
\end{align*}
\]

where the purpose of the game is to determine command \( S \). If \( Q \Rightarrow R \) (in other words, \( R \) is weaker than \( Q \)), then replacing \( S \) with the \texttt{skip} command makes the code segment correct, since then \( Q \Rightarrow \text{wp}(\texttt{skip}, R) \) is equivalent to \( Q \Rightarrow R \), which we assumed evaluates to TRUE.

Consider the following to-be-derived code segment that increases counter \( c \) by one if \( x = 0 \). The precondition indicates that \( x \neq 0 \). (This may be encountered, for example, as part of an \texttt{if} command.)

\[
\begin{align*}
\{ & Q : (x \neq 0) \land (c = \hat{c}) \\
\{ & Q \Rightarrow \text{wp}(\texttt{"S"}, R)? \\
\{ & \text{wp}(\texttt{"S"}, R): \\
S & \\
\{ & R : (x = 0 \land c = \hat{c} + 1) \lor (x \neq 0 \land c = \hat{c})
\end{align*}
\]

How do we systematically arrive upon the fact that \( S \) can be chosen to be the \texttt{skip} command? We check whether \( Q \Rightarrow R \):

\[
\frac{(x \neq 0) \land (c = \hat{c})}{Q} \quad \frac{(x = 0 \land c = \hat{c} + 1) \lor (x \neq 0 \land c = \hat{c})}{R} \\text{?}
\]

By weakening/strengthening (after commuting the disjunction), this implication is TRUE. Hence, \( S \) can be taken to be the \texttt{skip} command:

\[
\begin{align*}
\{ & Q : (x \neq 0) \land (c = \hat{c}) \\
\{ & Q \Rightarrow \text{wp}(\texttt{"S"}, R)? \text{ YES!} \\
\{ & \text{wp}(\texttt{"S"}, R): (x = 0 \land c = \hat{c} + 1) \lor (x \neq 0 \land c = \hat{c}) \\
S & : \texttt{skip} \\
\{ & R : (x = 0 \land c = \hat{c} + 1) \lor (x \neq 0 \land c = \hat{c})
\end{align*}
\]
3.2.2 Assignment to simple variables 

Consider the following to-be-derived code segment that increases counter $c$ by one if $x = 0$. The precondition indicates that $x = 0$. We assume that $x$ cannot change its value. (Again, this may be encountered, for example, as part of an if command.)

\[
\begin{align*}
\{Q : (x = 0) \land (c = \hat{c})\} \\
\{Q \Rightarrow \wp(“S”, R)? \} \\
\{\wp(“S”, R) : \} \\
S : \\
\{R : (x = 0 \land c = \hat{c} + 1) \lor (x \neq 0 \land c = \hat{c})\}
\end{align*}
\]

The question is how to systematically determine command $S$.

We first will want to check whether the precondition implies the postcondition, in which case $S$ can (and should) equal skip. The unfortunate truth is that it doesn’t. We will comment further on this later in this unit.

Next, we notice that the problem specification said that $x$ cannot change its value. By convention, $\hat{c}$ cannot change value either, because it is a “dummy variable” that is introduced to indicate the “original contents of $c$.” Thus, we conclude that an expression, $E$ must be assigned to $c$:

\[
\begin{align*}
\{Q : (x = 0) \land (c = \hat{c})\} \\
\{Q \Rightarrow \wp(“S”, R)? \} \\
\{\wp(“S”, R) : \} \\
S : c := E \\
\{R : (x = 0 \land c = \hat{c} + 1) \lor (x \neq 0 \land c = \hat{c})\}
\end{align*}
\]

where $E$ is the expression that is to be determined. Now, we can further annotate this code segment with $\wp(“c := E”, R)$

\[
\begin{align*}
\{Q : (x = 0) \land (c = \hat{c})\} \\
\{Q \Rightarrow \wp(“S”, R)? \} \\
\wp(“S”, R) : \wp(“c := E”, (x = 0 \land c = \hat{c} + 1) \lor (x \neq 0 \land c = \hat{c})) \\
\iff \text{< definition of := >} \\
(x = 0 \land E = \hat{c} + 1) \lor (x \neq 0 \land E = \hat{c}) \\
S : c := E \\
\{R : (x = 0 \land c = \hat{c} + 1) \lor (x \neq 0 \land c = \hat{c})\}
\end{align*}
\]
and now we notice that $E$ must be chosen so that $Q \Rightarrow \text{wp}("c := E", R)$:

$$(x = 0) \land (c = \hat{c}) \Rightarrow (x = 0 \land E = \hat{c} + 1) \lor (x \neq 0 \land E = \hat{c})$$

This guides us to choose $E$ to equal $c + 1$ since then

$$(x = 0) \land (c = \hat{c}) \Rightarrow (x = 0 \land c + 1 = \hat{c} + 1) \lor (x \neq 0 \land c + 1 = \hat{c})$$

which we notice is true by weakening/strengthening after subtracting +1 on both sides of $(c + 1 = \hat{c} + 1)$. Hence we conclude that the derived-to-be-correct code segment is given by

$$\{Q : (x = 0) \land (c = \hat{c})\}
\{Q \Rightarrow \text{wp}("S", R)? \text{YES!}\}
\{\text{wp}("S", R) : \text{wp}("c := c + 1", (x = 0 \land c = \hat{c} + 1) \lor (x \neq 0 \land c = \hat{c}))\}
\iff \text{definition of := >}
(x = 0 \land c + 1 = \hat{c} + 1) \lor (x \neq 0 \land c + 1 = \hat{c})

S : c := c + 1
\{R : (x = 0 \land c = \hat{c} + 1) \lor (x \neq 0 \land c = \hat{c})\}$$

Notice that the precondition and postcondition for the command $S$ prescribe what command $S$ needs to be.

Let us briefly return to the question of how to determine that the skip command is not a correct choice for command $S$. This is actually the wrong question. The right question is “What if we had tried to derive a command of the form $c := E$ for $S$ in the previous unit?” The answer is that the expression $E$ would have turned out to equal $c$. In other words, $S$ would have been the assignment $c := c$. But instead of assigning $c$ to $c$, we might as well then insert the skip command.

**Homework 3.2.2.1** Systematically derive the expressions $E_0$ and $E_1$ that make the following code segment correct:

$$\{Q : (s = \hat{s}) \land (t = \hat{t})\}$$
$s, t := E_0, E_1$
$$\{R : (s = \hat{t}) \land (t = \hat{s})\}$$

☞ DO EXERCISE ON edX

☞ Watch Video on edX
☞ Watch Video on YouTube
Homework 3.2.2.2 Find the missing assignment to make the following program segment correct. It may be part of a program that sets variable fac equal to $(n-1)! = (n-1) \times (n-2) \times \cdots \times 2 \times 1$. (Check your results!)

\[
\begin{align*}
\{ & Q : 0 < n \\ & i, \text{fac} := n, ? \\
\{ & R : 1 \leq i \leq n \land \text{fac} = (\prod j \mid i \leq j < n : j) \\
\end{align*}
\]

a) $i, \text{fac} := n, 0$

b) $i, \text{fac} := n, 1$

c) $i, \text{fac} := n, n$

d) cannot be made to be correct

DO EXERCISE ON edX

Homework 3.2.2.3 Find the missing assignment to make the following program segment correct. It may be part of a program that converts a binary representation (stored in array $b$) into a decimal number stored in $y$. (Check your results!)

\[
\begin{align*}
\{ & Q : y = (\sum j \mid i \leq j < n : b(j) \times 2^j) \\
y := ? \\
i := i - 1 \\
\{ & R : y = (\sum j \mid i \leq j < n : b(j) \times 2^j) \\
\end{align*}
\]

a) $y := b(i) \times 2^i + y$

b) $y := b(i - 1) \times 2^{i-1} + y$

c) $y := b(i + 1) \times 2^{i+1} + y$

d) cannot be made to be correct

DO EXERCISE ON edX

3.2.3 Careful! to edX

Watch Video on edX
Watch Video on YouTube
Consider the code segment

\[
\begin{array}{l}
\{ T \\
k := k + 1 \\
y := \mathcal{E} \\
\{ y = x(k) \}
\end{array}
\]

where \( \mathcal{E} \) is an expression to be determined so that the Hoare triple holds. Let us annotate:

\[
\begin{array}{l}
\{ T \\
wp("k := k + 1; y := \mathcal{E}", y = x(k)) : \mathcal{E} = x(k + 1) \\
k := k + 1 \\
wp("y := \mathcal{E}", y = x(k)) : \mathcal{E} = x(k) \\
y := \mathcal{E} \\
\{ y = x(k) \}
\end{array}
\]

We would conclude that we should choose expression \( \mathcal{E} \) to equal \( x(k + 1) \) because then

\[
T \Rightarrow x(k + 1) = x(k + 1).
\]

However, if we then insert \( y := x(k + 1) \) we notice that

\[
\begin{array}{l}
\{ T \\
wp("k := k + 1; y := x(k + 1)", y = x(k)) : x(k + 2) = x(k + 1) \\
k := k + 1 \\
wp("y := \mathcal{E}", y = x(k)) : x(k + 1) = x(k) \\
y := x(k + 1) \\
\{ y = x(k) \}
\end{array}
\]

which is NOT a correct code segment because

\[
T \Rightarrow x(k + 2) = x(k + 1)
\]

evaluates to TRUE only if \( x(k + 2) = x(k + 1) \).

What went wrong? The problem is that \( k \) appears in the expression \( \mathcal{E} \), and \( k \) is changed prior to the assignment \( y := x(k + 1) \). The way around this is to capture that \( \mathcal{E} \) is a function of \( k \):

\[
\begin{array}{l}
\{ T \\
k := k + 1 \\
y := \mathcal{E}(k) \\
\{ y = x(k) \}
\end{array}
\]
where \( E(k) \) represents an expression that may depend on \( k \) and that is to be determined so that the Hoare triple holds. Let us annotate:

\[
\begin{align*}
\{ \ T \ \} \\
\{ \ \text{wp}(“k := k + 1; y := E(k)”}, y = x(k) : E(k + 1) = x(k + 1) \} \\
\{ \ k := k + 1 \} \\
\{ \ \text{wp}(“y := E(k)”}, y = x(k) : E(k) = x(k) \} \\
\{ \ y := E(k) \} \\
\{ \ y = x(k) \} \\
\end{align*}
\]

We conclude that we should choose \( E(k + 1) \) to equal \( x(k + 1) \) and hence \( E(k) \) as \( x(k) \). Now let’s check:

\[
\begin{align*}
\{ \ T \ \} \\
\{ \ \text{wp}(“k := k + 1; y := x(k)”}, y = x(k) : x(k + 1) = x(k + 1) \} \\
\{ \ k := k + 1 \} \\
\{ \ \text{wp}(“y := x(k)”}, y = x(k) : x(k) = x(k) \} \\
\{ \ y := x(k) \} \\
\{ \ y = x(k) \} \\
\end{align*}
\]

which is a correct code segment because

\[
T \Rightarrow x(k + 1) = x(k + 1)
\]

evaluates to TRUE.

We conclude that given Hoare triple

\[
\begin{align*}
\{ \ Q \ \} \\
\{ \ k := \cdots \} \\
\{ \ y := ? \} \\
\{ \ R \ \} \\
\end{align*}
\]

where ? is to be determined, one should use \( E(k) \) if in the Hoare triple \( k \) appears on the left of an assignment statement prior to the assignment to \( y \). This generalizes if there are more assignments that precede the assignment to be determined. Each variable that appears on the left of an assignment prior to the assignment to \( y \) should appear as a parameter for expression \( E \).
Homework 3.2.3.1 Find the missing assignment to make the following program segment correct. (Check your results!)

\[
\begin{array}{l}
\{ \quad Q : 0 < i < n \\
i, j := i + 1, ? \\
\{ \quad R : j = n - i \land 0 \leq j < n \\
\end{array}
\]

a) \( i, j := i + 1, n - i \)

b) \( i, j := i + 1, n - i + 1 \)

c) \( i, j := i + 1, n - i - 1 \)

d) cannot be made to be correct

→ DO EXERCISE ON edX

Homework 3.2.3.2 Find the missing assignment to make the following program segment correct. (Check your results!)

\[
\begin{array}{l}
\{ \quad Q : 0 < i < n \\
i := i + 1 \\
\quad j := ? \\
\{ \quad R : j = n - i \land 0 \leq j < n \\
\end{array}
\]

a) \( j := n - i \)

b) \( j := n - i + 1 \)

c) \( j := n - i - 1 \)

d) cannot be made to be correct

→ DO EXERCISE ON edX
Homework 3.2.3.3 Find the missing assignment to make the following program segment correct. (Check your results!)

\[
\begin{align*}
\{ & Q : 0 < i < n \} \\
& j := \? \\
i := i + 1 \\
& \{ R : j = n - i \land 0 \leq j < n \}
\end{align*}
\]

a) \( j := n - i \)
b) \( j := n - i + 1 \)
c) \( j := n - i - 1 \)
d) cannot be made to be correct

DO EXERCISE ON edX

Assignment to array elements can in principle be made similarly systematic. It is the manipulation of the precondition and postcondition that becomes a bit trickier. The reason is that, as we noticed last week when proving code segments that involved assignment to arrays correct, “textual substitution” is greatly simplified if the array element being updated does not appear in a quantifier. For this reason, the range of the quantifier is split to expose that particular element.

Let us consider an example where array \( x \) is added to array \( y \), given in Figure 3.2, where command \( S \) is to be determined. (How to get to this point will be discussed in the later section on how to systematically develop a loop.)

Recognizing that \((0 \leq k \leq n) \land (k < n)\) is equivalent to \((0 \leq k < n)\) we focus on

\[
\begin{align*}
Q_S : ( & \forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i) ) \\
& \land ( \forall i \mid k \leq i < n : y(i) = \hat{y}(i) ) \land (0 \leq k < n)
\end{align*}
\]

\[
\begin{align*}
S : \\
R_S : ( & \forall i \mid 0 \leq i < k + 1 : y(i) = \hat{y}(i) + x(i) ) \\
& \land ( \forall i \mid k + 1 \leq i < n : y(i) = \hat{y}(i) ) \land (0 \leq k + 1 \leq n)
\end{align*}
\]

In reasoning about what \( S \) should be, we notice the difference in ranges in the precondition and postcondition. To reconcile these, we split ranges both in the precondition and the postcondition:
3.2. Developing Simple Commands

Figure 3.2: Add the values in array $x$ to the corresponding values in array $y$. It is implicitly assumed that the values in $x$ do not change. Array $\hat{y}$ is introduced to be able to refer to the original contents of $y$. It is implicitly assumed that the arrays are of size $n$. 

\[
\{ Q : (\forall i \mid 0 \leq i < n : y(i) = \hat{y}(i)) \land (0 \leq n) \} \\
\]

\[
k := 0 \\
\{ P_{\text{inv}} : (\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \land (\forall i \mid k \leq i < n : y(i) = \hat{y}(i)) \land (0 \leq k \leq n) \} \\
\]

\[\text{while } k < n \text{ do} \]

\[
\{ P_{\text{inv}} \land G : (\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \\
\land (\forall i \mid k \leq i < n : y(i) = \hat{y}(i)) \land (0 \leq k \leq n) \land (k < n) \} \\
\]

\[
S:\ \\
\{ \text{wp} \langle "k := k + 1", P_{\text{inv}} \rangle : (\forall i \mid 0 \leq i < k + 1 : y(i) = \hat{y}(i) + x(i)) \\
\land (\forall i \mid k + 1 \leq i < n : y(i) = \hat{y}(i)) \land (0 \leq k + 1 \leq n) \} \\
\]

\[
k := k + 1 \\
\{ P_{\text{inv}} : (\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \land (\forall i \mid k \leq i < n : y(i) = \hat{y}(i)) \land (0 \leq k \leq n) \} \\
\}

\[\text{endwhile} \]

\[
\{ P_{\text{inv}} : (\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \land (\forall i \mid k \leq i < n : y(i) = \hat{y}(i)) \land (0 \leq k \leq n) \land \neg (k < n) \} \\
\}

\[
(\forall i \mid 0 \leq i < n : y(i) = \hat{y}(i) + x(i)) \\
\]
We notice that if we choose $E$ to equal $y(k) + \hat{y}(k)$, then

$$y(k) + x(k) = \hat{y}(k) + x(k)$$

and, after cancelling $x(k)$ on both sides, by weakening/strengthening $Q_S \Rightarrow \text{wp}(\text{“}y(k) := y(k) + x(k)\text{”}, R_S)$ becomes TRUE.
3.3. Developing the if Command  to edX  Monday 7th August, 2017 at 13:32

3.3 Developing the if Command  to edX

3.3.1 A general strategy  to edX

Consider a prototypical if command that has been annotated:

\[
\begin{align*}
Q_s : & \quad (\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \\
& \quad \land (y(k) = \hat{y}(k)) \land (\forall i \mid k + 1 \leq i < n : y(i) = \hat{y}(i)) \land (0 \leq k < n) \\
\end{align*}
\]

\[
\begin{align*}
Q_s \Rightarrow \text{wp}("S", R_s) \text{? YES!} \\
\end{align*}
\]

\[
\begin{align*}
\text{wp}("y(k) := y(k) + x(k)", R_s) : & \quad (\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \\
& \quad \land (y(k) + x(k) = \hat{y}(k) + x(k)) \land (\forall i \mid k + 1 \leq i < n : y(i) = \hat{y}(i)) \\
& \quad \land (1 \leq k < n) \\
\end{align*}
\]

\[
\begin{align*}
S : & \quad y(k) := y(k) + x(k) \\
\end{align*}
\]

\[
\begin{align*}
R_s : & \quad (\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \\
& \quad \land (y(k) = \hat{y}(k) + x(k)) \land (\forall i \mid k + 1 \leq i < n : y(i) = \hat{y}(i)) \land (1 \leq k < n). \\
\end{align*}
\]
What do we know about the guards? Recall from the If Theorem that it must be the case that 

\[ Q \Rightarrow G_0 \lor G_1 \lor \cdots \lor G_{K-1}, \]

where \( K \) equals the number of guarded commands in the `if` command. A straightforward strategy for determining the guards is to keep adding new guards until \( Q \Rightarrow G_0 \lor G_1 \lor \cdots \lor G_{K-1} \). Every time a new guard \( G_i \) is chosen, we focus on the (hopefully) simpler subproblem of determining command \( S_i \) that makes the code segment

\[
\{ G_i \land Q \land Q \Rightarrow \wp(“S_i” , R) \} 
\{ \wp(“S_i” , R) \} 
\{ R \} 
\]
correct.

Let us illustrate this with the following example that sets variable \( z \) to the absolute value of \( x \), where for now we assume you understand the intuitive meaning of the absolute value function, \( \text{abs}() \). Later we will more precisely define it and see how this helps us find guards even more systematically.
Let’s guess a loop guard, \( G_0 : x = -1 \) and place it in the if command recipe. After developing command \( S_0 \), we would arrive at

\[
\begin{align*}
\{ & T \\
\text{if} & \ \\
& x = -1 \rightarrow \\
& \{ Q \land G_0 : T \land x = -1 \} \\
& \{ T \land x = -1 \Rightarrow \wp(“S_0”, R) \text{? YES!} \} \\
& \{ \wp(“S_0”, R) : 1 = \text{abs}(x) \} \\
S_0 : & z = 1 \\
& \{ R : z = \text{abs}(x) \} \\
\text{fi} & \\
\{ & \text{R : } z = \text{abs}(x) \\
\end{align*}
\]

We then check whether \( Q \Rightarrow G_0 \), which instantiates to \( T \Rightarrow x = -1 \) and conclude we need more guards. So, we add the guards \( G_1 : x = 1, G_2 : x = 0 \) to arrive at
\[
\begin{array}{l}
\{ T \} \\
\textbf{if} \\
x = -1 \rightarrow \\
\{ Q \land G_0 : T \land x = -1 \} \\
\{ T \land x = -1 \Rightarrow \wp(“S_0” , R) ? \text{YES!} \} \\
\{ \wp(“S_0” , R) : 1 = \text{abs}(x) \} \\
S_0 : z = 1 \\
\{ R : z = \text{abs}(x) \} \\
\end{array}
\]

\[
\begin{array}{l}
x = 1 \rightarrow \\
\{ Q \land G_1 : T \land x = 1 \} \\
\{ T \land x = 1 \Rightarrow \wp(“S_1” , R) ? \text{YES!} \} \\
\{ \wp(“S_1” , R) : 1 = \text{abs}(x) \} \\
S_1 : z = 1 \\
\{ R : z = \text{abs}(x) \} \\
\end{array}
\]

\[
\begin{array}{l}
x = 0 \rightarrow \\
\{ Q \land G_2 : T \land x = 0 \} \\
\{ T \land x = 0 \Rightarrow \wp(“S_2” , R) ? \text{YES!} \} \\
\{ \wp(“S_2” , R) : 0 = \text{abs}(x) \} \\
S_2 : z = 0 \\
\{ R : z = \text{abs}(x) \} \\
\end{array}
\]

\[
\textbf{fi} \\
\{ R : z = \text{abs}(x) \}
\]

Now, obviously we would have to create an infinite number of guards if we kept going this way. We need to be smarter.

The key is to be more explicit about expressing the postcondition \( R : z = \text{abs}(x) \). Instead, let’s use the definition of the absolute value:

- If \( x \) is greater than or equal to zero, \( \text{abs}(x) = x \).
- If \( x \) is less than or equal to zero, \( \text{abs}(x) = -x \).

This can be expressed as the predicate

\[
R : (x \geq 0 \land z = x) \lor (x \leq 0 \land z = -x).
\]

Notice that it is fine to have the conditions \( x \geq 0 \) and \( x \leq 0 \) overlap, as long as the \textbf{if} command is derived to be correct. We also notice that the guards are easily identified in the postcondition: \( x \geq 0 \) and \( x \leq 0 \).

After deriving \( S_0 \) and \( S_1 \) using the techniques from earlier this week, the \textbf{if} command recipe yields
The details of how to systematicallyally derive $S_i$ for the code segment, given $Q, G_i$, and $R$:

\[
\begin{align*}
\{ G_i \land Q \} \\
\{ G_i \land Q \Rightarrow \wp("S_i", R)? \} \\
\{ \wp("S_i", R) : \} \\
S_i \\
\{ R \}
\end{align*}
\]

build upon what we know about how to derive simple commands. Still, it quickly becomes messy when the postcondition is nontrivial. For this reason, we don’t focus on the general case in this course, treating a simpler (but common) case instead, in the next unit.

One strategy, obviously, is to guess and to then check correctness. This is a perfectly reasonable strategy. Often, the annotations guide one towards an educated guess.
3.3.2 A commonly encountered case

The postcondition for an if command can often be chosen or manipulated into a convenient format that makes the identification of the guards and the derivation of the guarded commands simpler. In this unit we discuss one such case.

**Homework 3.3.2.1** Identify for each of the operations on the left the corresponding predicate that best expresses it on the right.

1. \( z = \text{abs}(x) \)
   - a. \((x \leq 0 \land c = \hat{c} + 1) \lor (x > 0 \land c = \hat{c})\)
2. \( z = \text{min}(x,y) \)
   - b. \((x \leq y \land z = y) \lor (x \geq y \land z = x)\)
3. \( z = \text{max}(x,y) \)
   - c. \((x \leq y \land z = x) \lor (x \geq y \land z = y)\)
4. \( z = \text{abs}(x - y) \)
   - d. \((x \geq 0 \land z = x) \lor (x \leq 0 \land z = -x)\)
5. Increment \( c \) by one if \( x \leq 0 \)
   - e. \((x \geq y \land z = x - y) \lor (y \geq x \land z = y - x)\)

What the examples from the last homework have in common is that the predicate that describes the operation to be computed can be written as a disjunction of conjunctions:

\[(G_0 \land R_0) \lor (G_1 \land R_1) \lor \cdots \lor (G_{K-1} \land R_{K-1}).\]

Now, each component in the disjunction is by itself stronger than \( R \):

\[(G_j \land R_j) \Rightarrow (G_0 \land R_0) \lor (G_1 \land R_1) \lor \cdots \lor (G_{K-1} \land R_{K-1})\]

by weakening/strengthening. What this suggests is the specialized worksheet for this case given in Figure 3.3. Deriving each guarded command \( S_j \) now comes down to deriving the code segments

```latex
\{G_j \land Q : \} \\
\{G_j \land Q \Rightarrow \text{wp("S_j", G_j \land R_j)?} : \} \\
\{\text{wp("S_j", G_j \land R_j):} : \} \\
S_j \\
\{G_j \land R_j : \}
```

in that worksheet.
Figure 3.3: Worksheet for the case where $R$ has the form $(G_0 \land R_0) \lor (G_1 \land R_1) \lor \cdots \lor (G_{K-1} \land R_{K-1})$. 
\[
\begin{align*}
\{ & Q : c = \hat{c} \\
\{ & Q \Rightarrow x \leq 0 \lor x > 0? \text{ YES!} \\
\text{if} \\
\quad x \leq 0 \rightarrow \\
\quad \{ & G_0 \land Q : x \leq 0 \land c = \hat{c} \\
\quad \{ & G_0 \land Q \Rightarrow \text{wp}("S_0", G_0 \land R_0)? \\
\quad \{ & \text{wp}("S_0", G_0 \land R_0) : x \leq 0 \land E_0 = \hat{c} + 1 \\
\quad S_0 : c = E_0 \\
\quad \{ & G_0 \land R_0 : x \leq 0 \land c = \hat{c} + 1 \\
\lbrack & x > 0 \rightarrow \\
\quad \{ & G_1 \land Q : x > 0 \land c = \hat{c} \\
\quad \{ & G_1 \land Q \Rightarrow \text{wp}("S_1", G_1 \land R_1)? \\
\quad \{ & \text{wp}("S_1", G_1 \land R_1) : x > 0 \land E_1 = \hat{c} \\
\quad S_1 : c = E_1 \\
\quad \{ & G_1 \land R_1 : x > 0 \land c = \hat{c} \\
\text{fi} \\
\{ & R : (x \leq 0 \land c = \hat{c} + 1) \lor (x > 0 \land c = \hat{c})
\end{align*}
\]

Figure 3.4: Partially completed worksheet for computing \((x \leq 0 \land c = \hat{c} + 1) \lor (x > 0 \land c = \hat{c})\).
\[
\{ Q : c = \hat{c} \} \\
\{ Q \Rightarrow x \leq 0 \lor x > 0 \? YES! \}
\]

\begin{tabular}{ll}
if & \\
\{ x \leq 0 \rightarrow \} & \\
\{ G_0 \land Q : x \leq 0 \land c = \hat{c} \} & \\
\{ G_0 \land Q \Rightarrow \wp\("S_0", G_0 \land R_0")? YES! \} & \\
\{ \wp\("S_0", G_0 \land R_0") : x \leq 0 \land c + 1 = \hat{c} + 1 \} & \\
\end{tabular}

\begin{tabular}{ll}
\textcolor{red}{S_0 : c = c + 1} & \\
\{ G_0 \land R_0 : x \leq 0 \land c = \hat{c} + 1 \} & \\
\end{tabular}

\begin{tabular}{ll}
\{ \neg x \geq 0 \rightarrow \} & \\
\{ G_1 \land Q : x > 0 \land c = \hat{c} \} & \\
\{ G_1 \land Q \Rightarrow \wp\("S_1", G_1 \land R_1")? YES! \} & \\
\{ \wp\("S_1", G_1 \land R_1") : x > 0 \land c = \hat{c} \} & \\
\end{tabular}

\begin{tabular}{ll}
\textcolor{red}{S_1 : \text{skip}} & \\
\{ G_1 \land R_1 : x > 0 \land c = \hat{c} \} & \\
\end{tabular}

\begin{tabular}{ll}
fi & \\
\{ R : (x \leq 0 \land c = \hat{c} + 1) \lor (x > 0 \land c = \hat{c}) \} & \\
\end{tabular}

Figure 3.5: Completed worksheet for computing \((x \leq 0 \land c = \hat{c} + 1) \lor (x > 0 \land c = \hat{c})\).
Example 3.1 Consider the Hoare triple

\[ \{ Q : c = \hat{c} \} \]

\[ S : \]

\[ \{ R : (x \leq 0 \land c = \hat{c} + 1) \lor (x > 0 \land c = \hat{c}) \} \]

for the command \( S \) that increments \( c \) if \( x \leq 0 \). Here, as usual, \( \hat{c} \) is a “dummy variable” introduced to denote the contents of \( c \) at the beginning of the code segment. We assume that the contents of variable \( x \) will not be changed.

We use the insights in this unit to derive \( S \). We verify that

\[ (x \leq 0 \land c = \hat{c} + 1) \lor (x > 0 \land c = \hat{c}) \]

\[ G_0 \quad R_0 \quad G_1 \quad R_1 \]

Thus, the postcondition has the desired format. Also \( Q \rightarrow G_0 \lor G_1 \) since \( G_0 \lor G_1 \leftrightarrow T \). This allows us to fill out many of the expressions in the worksheet, yielding Figure 3.4. There, we also indicate that \( S_0 \) and \( S_1 \) update \( c \) with expressions \( E_0 \) and \( E_1 \), respectively. From the highlighted fields we deduce that \( E_0 \) should equal \( c + 1 \) (so that \( S_0 \) equals \( c := c + 1 \)) and that \( E_1 \) should equal \( c \) (so that \( S_1 \) should equal \( c := c \) or, equivalently, \( \text{skip} \)). This is summarized in Figure 3.5.

Homework 3.3.2.2 Use Figure 3.6 to develop a code segment that computes \( z = \min(x, y) \):

\[ \{ Q : T \} \]

\[ S \]

\[ \{ R : (x \leq y \land z = x) \lor (x \geq y \land z = y) \} \]

DO EXERCISE ON edX

Homework 3.3.2.3 Use Figure 3.6 to develop a code segment that computes \( z = \abs{x - y} \):

\[ \{ Q : T \} \]

\[ S \]

\[ \{ R : (x \geq y \land z = x - y) \lor (y \geq x \land z = y - x) \} \]

DO EXERCISE ON edX
Figure 3.6: Worksheet for the case where $R$ has the form $(G_0 \land R_0) \lor (G_1 \land R_1)$. 

{ $Q$ : 
  
$Q \Rightarrow G_0 \lor G_1?}$ 

if 

$G_0 \Rightarrow$

{ $G_0 \land Q$: 

$G_0 \land Q \Rightarrow \text{wp}("S_0", G_0 \land R_0)$? 

$\text{wp}("S_0", G_0 \land R_0)$: 

$S_0$: 

$G_0 \land R_0$: 

$\textbf{[}G_1 \Rightarrow$

{ $G_1 \land Q$: 

$G_1 \land Q \Rightarrow \text{wp}("S_1", G_1 \land R_1)$? 

$\text{wp}("S_1", G_1 \land R_1)$: 

$S_1$: 

$G_1 \land R_1$: 

$\textbf{]}$ fi

$R$: 


3.4 Developing a While Command to edX

3.4.1 A worksheet for the while command to edX

In Week 2 Section 2.5.1, we discussed how to prove a while loop correct. Let us now turn to how to derive a typical while loop.

Consider the prototypical loop

\[
\begin{align*}
S_I \\
\text{while } G \text{ do} \\
S \\
\text{endwhile}
\end{align*}
\]

where command \( S_I \) initializes variables (we call this the initialization step), \( G \) is the loop guard, and \( S \) is the loop body.

The key is to turn the proof of correctness for a loop into a worksheet much like we created a worksheet for the if command. This worksheet was hinted at in the launch for this week and is given by

\[
\begin{align*}
\{ & Q : \\
& S_I \\
& \{ & P_{\text{inv}} \\
& \text{while } G \text{ do} \\
& \{ & P_{\text{inv}} \land G \\
& \{ & \text{wp}(“S”, P_{\text{inv}}) \\
& S \\
& \{ & P_{\text{inv}} \\
& \text{endwhile} \\
& \{ & P_{\text{inv}} \land \neg G \\
& \{ & R
\end{align*}
\]

Now, if we can a priori determine what \( P_{\text{inv}} \) is, then everything else in this loop is predetermined:

- We know that \( (P_{\text{inv}} \land \neg G) \Rightarrow R \) must be true. Thus, given \( P_{\text{inv}} \) and \( R \) we must determine a condition \( G \) that makes this true. In other words, to determine \( G \) we focus on
A careful reexamination of how to prove this given $P_{\text{inv}}$, $R$, and $G$ in Week 2 tells us that weakening/strengthening laws are usually provide insight into how to choose $G$.

- Given $Q$ and $P_{\text{inv}}$ we can deploy techniques discussed earlier in this week to derive the initialization command $S_I$. For this, we focus on deriving $S_I$ so that

$$\{ Q : \}
S_I
\{ P_{\text{inv}} \}$$

holds. This is a smaller subproblem and hopefully easier to tackle.

- Similarly, given $P_{\text{inv}}$ and $G$ we can deploy techniques previously discussed in this week to derive command $S$. For this, we focus on deriving $S$ so that

$$\{ P_{\text{inv}} \land G : \}
S
\{ P_{\text{inv}} \}$$

holds. Now, the naive solution would be to choose $S : \text{skip}$ since

$$\{ P_{\text{inv}} \land G : \}
S : \text{skip}
\{ P_{\text{inv}} \}$$

can be easily shown to always hold. However, we need to make progress towards completion since otherwise the loop will never stop. It is this that forces $S$ to be a more complicated command.

### 3.4.2 Progress towards completion

- Watch Video on edX
- Watch Video on YouTube
In this first part of the course, we mainly focus on algorithms that traverse one dimensional arrays in a systematic way: either from the first element to the last or from the last element to the first. Which of these two cases applies can be determined from the loop invariant. Because of this restriction, we can refine the worksheet further. Assuming the variable that keeps track of where in the array we are is given by \( k \), if the algorithm traverses from the first element to the last, the worksheet becomes

\[
\begin{align*}
\{ & Q \\
S_I & \\
\{ & P_{inv} \\
\textbf{while} & G \textbf{ do} \\
\{ & P_{inv} \land G \\
\{ & \wp(\text{"S}_U; k := k + 1", P_{inv}) \\
S_U & \\
\{ & \wp(\text{"k := k + 1", P}_{inv}) \\
k := k + 1 & \\
\{ & P_{inv} \\
\textbf{endwhile} \\
\{ & P_{inv} \land \neg G \\
\{ & R \\
\end{align*}
\]

or

\[
\begin{align*}
\{ & Q \\
S_I & \\
\{ & P_{inv} \\
\textbf{while} & G \textbf{ do} \\
\{ & P_{inv} \land G \\
\{ & \wp(\text{"k := k + 1; S}_U", P_{inv}) \\
k := k + 1 & \\
\{ & \wp(\text{"S}_U", P_{inv}) \\
S_U & \\
\{ & P_{inv} \\
\textbf{endwhile} \\
\{ & P_{inv} \land \neg G \\
\{ & R \\
\end{align*}
\]
Here $S_U$ is the command that updates the pertinent variables. If on the other hand the algorithm traverses the array from last to first (as in the example in the launch), the worksheet is given by

$$
\begin{align*}
\{ & Q \\
S_I & \\
\{ & P_{\text{inv}} \\
\text{while} & G \text{ do} \\
\{ & P_{\text{inv}} \land G \\
\{ & \text{wp}("S_U\;k := k - 1", P_{\text{inv}}) \\
S_U & \\
\{ & \text{wp}("k := k - 1", P_{\text{inv}}) \\
& k := k - 1 \\
\{ & P_{\text{inv}} \\
\text{endwhile} \\
\{ & P_{\text{inv}} \land \neg(G) \\
\{ & R
\end{align*}
$$

or

$$
\begin{align*}
\{ & Q \\
S_I & \\
\{ & P_{\text{inv}} \\
\text{while} & G \text{ do} \\
\{ & P_{\text{inv}} \land G \\
\{ & \text{wp}("k := k - 1; S_U", P_{\text{inv}}) \\
& k := k - 1 \\
\{ & \text{wp}("S_U", P_{\text{inv}}) \\
S_U & \\
\{ & P_{\text{inv}} \\
\text{endwhile} \\
\{ & P_{\text{inv}} \land \neg(G) \\
\{ & R
\end{align*}
$$

Notice that if $k$ is part of the loop guard, it can be chosen to be part of the bound function $t$ that is used to prove complete correctness, which then guarantees completion of the loop. For this reason, we don’t bother with proving complete correctness, since we know loops of the above structure will complete.
3.4.3 A priori determination of loop invariants

The insights in the last two units lead us to the conclusion that we need to systematically derive \( P_{inv} \) from precondition \( Q \) and postcondition \( R \). If we can do that, then we have a systematic way of deriving loop-based algorithms since the other parts of the loop with systematically fall in place.

Let us go back and consider a loop that adds the contents of array \( x \) to those in array \( y \) in Figure 3.2, but let’s pretend we don’t know what the loop should be. Let’s choose to progress through the arrays from first to last element. We will implicitly assume that the arrays are of size \( n \) with \( 0 \leq n \). The precondition then becomes

\[
Q : (\forall i \mid 0 \leq i < n : y(i) = \hat{y}(i)) \land 0 \leq n
\]

while the postcondition is given by

\[
R : (\forall i \mid 0 \leq i < n : y(i) = \hat{y}(i) + x(i)).
\]

It is from these two expressions that we now want to derive possible loop invariants.

In the worksheets discussed in the last unit, \( k \) becomes the index that keeps track of where in the arrays we are. It tells us what parts of the array have been processed. To describe this with a predicate, we take quantifiers in the precondition and postcondition, and split the ranges using this index: the precondition

\[
Q : (\forall i \mid 0 \leq i < n : y(i) = \hat{y}(i)) \land 0 \leq n
\]

is split to yield

\[
(\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i)) \land (\forall i \mid k \leq i < n : y(i) = \hat{y}(i)) \land 0 \leq k \leq n
\]

and the postcondition

\[
R : (\forall i \mid 0 \leq i < n : y(i) = \hat{y}(i) + x(i))
\]

is split to yield

\[
(\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \land (\forall i \mid k \leq i < n : y(i) = \hat{y}(i) + x(i)) \land 0 \leq k \leq n.
\]

This systematic splitting of the ranges is the first step.

Now, as computation proceeds, we should make progress towards the result, which is given by the postcondition. How can we construct a predicate that describes such progress? In other words, how can we extract a loop invariant from the above information? A loop invariant should capture the partial progress towards the final result.

Let us consider possibilities derived from the precondition and postcondition, calling these candidates for loop invariants (Candidates A through D):
3.4. Developing a While Command

<table>
<thead>
<tr>
<th>Candidate</th>
<th>Loop invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td>((\forall i \mid 0 \leq i &lt; k : y(i) = \hat{y}(i)) \land (\forall i \mid k \leq i &lt; n : y(i) = \hat{y}(i)) \land (0 \leq k \leq n))</td>
</tr>
<tr>
<td><strong>B</strong></td>
<td>((\forall i \mid 0 \leq i &lt; k : y(i) = \hat{y}(i) + x(i)) \land (\forall i \mid k \leq i &lt; n : y(i) = \hat{y}(i)) \land (0 \leq k \leq n))</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>((\forall i \mid 0 \leq i &lt; k : y(i) = \hat{y}(i)) \land (\forall i \mid k \leq i &lt; n : y(i) = \hat{y}(i) + x(i)) \land (0 \leq k \leq n))</td>
</tr>
<tr>
<td><strong>D</strong></td>
<td>((\forall i \mid 0 \leq i &lt; k : y(i) = \hat{y}(i) + x(i)) \land (\forall i \mid k \leq i &lt; n : y(i) = \hat{y}(i) + x(i)) \land (0 \leq k \leq n))</td>
</tr>
</tbody>
</table>

Notice:

- Candidate A describes a loop invariant that says that as the loop progresses, \(y\) always contains its original contents. Therefore, upon completing the loop, it will still contain its original contents. Obviously there is no loop guard such that \(P_{\text{inv}} \land \neg G \Rightarrow R\). It is not a valid loop invariant for this computation.

- Candidate B describes a loop invariant that says that as the loop progresses, the first \(k\) elements of \(y\) have been updated with the final result, while the last \(n - k\) elements have not yet been updated. We will later argue that there is a \(G\) such that \(P_{\text{inv}} \land \neg G \Rightarrow R\) and there is an initialization \(S_I\) such that \\(\{Q\}S_I\{P_{\text{inv}}\}\) holds. It will result in a correct algorithm that we will call (algorithmic) Variant 1, corresponding to the given (loop) Invariant 1.

- Candidate C describes a loop invariant that says that as the loop progresses, the last \(n - k\) elements of \(y\) have been updated with the final result, while the first \(k\) elements have not yet been updated. Again, we will later argue that there is a \(G\) such that \(P_{\text{inv}} \land \neg G \Rightarrow R\) and there is an initialization \(S_I\) such that \\(\{Q\}S_I\{P_{\text{inv}}\}\) holds. It will result in a correct algorithm that we will call Variant 2, corresponding to the given Invariant 2.

- Candidate D describes a loop invariant that says that as the loop progresses, \(y\) contains its final contents. Obviously, this means that before the loop starts, \(y\) must already have been initialized with the final contents. Given that we intend for \(S_I\) to only perform simple initializations, it is not a valid loop invariant for this computation.

We have systematically identified two viable loop invariants!
3.4.4 Deriving the loop guard and initialization command to edX

IMPORTANT. In the videos of Units 3.4.4 and 3.4.5, we discuss how to derive the loop for the case where the algorithm marches through the vectors from the last element to the first element. We consider this to be the case corresponding to (Loop) Invariant 2. At the end, in Homework 3.4.5.1, we have an exercise on the edX platform that asks you to derive the algorithm that marches through the vectors from first to last (Invariant 1).

Here in the book in the same units, we discuss how to derive the loop for the case where the algorithm marches through the vectors from the first element to the last element. In other words, Units 3.4.4 and 3.4.5 of the book are a very thorough answer to Homework 3.4.5.1.

We suggest that you watch the videos, do Homework 3.4.5.1 in the online version of Unit 3.4.5, and then go through the materials in the book. Alternatively, go through Units 3.4.4 and 3.4.5 in the book, do Homework 3.4.5.1 in the book, and then go back and watch the videos.

The discussion in the last unit left us with two viable loop invariants for updating an array $y$ by adding the elements of $x$ to each of its elements. Let us pick Invariant 1 and use it to derive what remains of the loop. For now, we only know the precondition $Q$, the postcondition $R$, and the loop invariant $P_{\text{inv}}$, as shown in Figure 3.7. What is left to be determined are the loop guard $G$, the initialization command $S_I$, how to update the loop index $k$, and the update command $S_U$.

We start by systematically determining the loop guard. What we know is that

$$P_{\text{inv}} \land \neg G : (\forall i | 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \land (\forall i | k \leq i < n : y(i) = \hat{y}(i)) \land 0 \leq k \leq n \land \neg(G)$$

must imply

$$R : (\forall i | 0 \leq i < n : y(i) = \hat{y}(i) + x(i)).$$

By examination, we deduce that choosing $G$ as $k < n$ has the desired property because then $P_{\text{inv}} \land \neg G$ becomes

$$(\forall i | 0 \leq i < n : y(i) = \hat{y}(i) + x(i)) \land (\forall i | n \leq i < n : y(i) = \hat{y}(i)) \land k = n,$$

which implies the postcondition by weakening/strengthening.

Next, we systematically derive the initialization, which must make the Hoare triple

$$\{ Q : (\forall i | 0 \leq i < n : y(i) = \hat{y}(i)) \land 0 \leq n \}$$

$$\{ \wp("S_I", P_{\text{inv}}) : \}$$

$$S_I :$$

$$\{ P_{\text{inv}} : (\forall i | 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \land (\forall i | k \leq i < n : y(i) = \hat{y}(i)) \land 0 \leq k \leq n \}$$

hold. Notice that if $S_I$ is chosen to set $k := 0$, then the first quantifier in the loop invariant has an empty range and equals TRUE, meaning that the precondition implies that $P_{\text{inv}}$ holds. More precisely:
3.4. Developing a While Command to edX

Monday 7th August, 2017 at 13:32

| $Q : (\forall i \ | \ 0 \leq i < n : y(i) = \widehat{y}(i)) \land 0 \leq n$ |
|---------------------------------------------------------------|
| $\text{wp(}"S_I",P_{inv})$ : |

$S_I$ :

| $P_{inv} : (\forall i \ | \ 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \land (\forall i \ | \ k \leq i < n : y(i) = \widehat{y}(i)) \land 0 \leq k \leq n$ |
|---------------------------------------------------------------|
| $\text{wp(}"S_U; k := \mathcal{E}(k)"\text{,}P_{inv})$ : |

$S_U$ :

| $\text{wp(}"k := \mathcal{E}(k)"\text{,}P_{inv})$ : |

$k := \mathcal{E}(k)$

| $P_{inv} : (\forall i \ | \ 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \land (\forall i \ | \ k \leq i < n : y(i) = \widehat{y}(i)) \land 0 \leq k \leq n$ |
|---------------------------------------------------------------|
| $\text{endwhile}$ |

| $P_{inv} \land \neg G : (\forall i \ | \ 0 \leq i < k : y(i) = \widehat{y}(i) + x(i)) \land (\forall i \ | \ k \leq i < n : y(i) = \widehat{y}(i)) \land 0 \leq k \leq n \land \neg (G)$ |
|---------------------------------------------------------------|
| $\neg (G)$ |
| $\neg G$ |
| $R : (\forall i \ | \ 0 \leq i < n : y(i) = \widehat{y}(i) + x(i))$ |

Figure 3.7: Recipe for adding vector $x$ to vector $y$ after filling in Invariant 1.
\{ Q : (\forall i \mid 0 \leq i < n : y(i) = \hat{y}(i)) \wedge 0 \leq n \}

\{ \text{wp}("S_I", P_{inv}) : T \}

\{ S_I : k := 0 \}

\{ P_{inv} : (\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \wedge (\forall i \mid k \leq i < n : y(i) = \hat{y}(i)) \wedge 0 \leq k \leq n \}

holds.

Finally, we notice that initially $k = 0$ and eventually $k = n$. Thus, it makes sense to increment $k := k + 1$ in the loop body. With these insights, the worksheet is further filled out as illustrated in Figure 3.8.

### 3.4.5 Deriving the loop body \(\rightarrow\) to edX

**IMPORTANT.** In the videos of Units 3.4.4 and 3.4.5, we discuss how to derive the loop for the case where the algorithm marches through the vectors from the last element to the first element. We consider this to be the case corresponding to (Loop) Invariant 2. At the end, in Homework 3.4.5.1, we have an exercise on the edX platform that asks you to derive the algorithm that marches through the vectors from first to last (Invariant 1).

Here in the book in the same units, we discuss how to derive the loop for the case where the algorithm marches through the vectors from the first element to the last element. In other words, Units 3.4.4 and 3.4.5 of the book are a very thorough answer to Homework 3.4.5.1.

We suggest that you watch the videos, do Homework 3.4.5.1 in the online version of Unit 3.4.5, and then go through the materials in the book. Alternatively, go through Units 3.4.4 and 3.4.5 in the book, do Homework 3.4.5.1 in the book, and then go back and watch the videos.

We now focus on the loop body:

\{ P_{inv} \wedge G : (\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \wedge (\forall i \mid k \leq i < n : y(i) = \hat{y}(i)) \wedge 0 \leq k \leq n \wedge k < n \}

\{ \text{wp}("S_U; k := k + 1", P_{inv}) : \}

\{ S_U : \}

\{ \text{wp}("k := k + 1", P_{inv}) : \}

$K := K + 1$

\{ P_{inv} : (\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \wedge (\forall i \mid k \leq i < n : y(i) = \hat{y}(i)) \wedge 0 \leq k \leq n \}

Watch Video on edX
Watch Video on YouTube
\[
\begin{aligned}
Q: & \quad (\forall i \mid 0 \leq i < n : y(i) = \hat{y}(i)) \land 0 \leq n \\
WP(S_I, P_{inv}): & \quad (\forall i \mid 0 \leq i < 0 : y(i) = \hat{y}(i) + x(i)) \\
& \quad \land (\forall i \mid 0 \leq i < n : y(i) = \hat{y}(i)) \land 0 \leq 0 \leq n \\
S_I: & \quad k := 0 \\
P_{inv}: & \quad (\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \land (\forall i \mid k \leq i < n : y(i) = \hat{y}(i)) \\
& \quad \land 0 \leq k \leq n \\
while \quad k < n \quad do \\
P_{inv} \land G: & \quad (\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \land (\forall i \mid k \leq i < n : y(i) = \hat{y}(i)) \\
& \quad \land k < n \\
WP(S_U; k := k + 1, P_{inv}): & \\
S_U: & \quad wp(“k := k + 1”, P_{inv}): \\
\begin{aligned}
k := k + 1 \\
P_{inv}: & \quad (\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \land (\forall i \mid k \leq i < n : y(i) = \hat{y}(i)) \\
& \quad \land 0 \leq k \leq n \\
endwhile
\end{aligned}
\end{aligned}
\]
The insight we have is that the range of the first quantifier in $P_{\text{inv}}$ expanded by one element in the current iteration, and it is that element that is updated. For this reason, we split the range of that quantifier:

\[
\begin{align*}
\{P_{\text{inv}} \land G : (\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \land (\forall i \mid k \leq i < n : y(i) = \hat{y}(i)) \land 0 \leq k \leq n \land k < n \} \\
\{\text{wp}("S_U; k := k + 1", P_{\text{inv}}) : \\
S_U : \\
\{\text{wp}("k := k + 1", P_{\text{inv}}) : \\
\end{align*}
\]

We can then compute

\[
\text{wp}("k := k + 1", P_{\text{inv}}) = (\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \land y(k) = \hat{y}(k) + x(k) \\
\land (\forall i \mid k + 1 \leq i < n : y(i) = \hat{y}(i)) \land 0 \leq k + 1 \leq n.
\]

We enter this into the loop body:

\[
\begin{align*}
\{P_{\text{inv}} \land G : (\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \land (\forall i \mid k \leq i < n : y(i) = \hat{y}(i)) \land 0 \leq k \leq n \land k < n \} \\
\{\text{wp}("S_U; k := k + 1", P_{\text{inv}}) : \\
S_U : \\
\{\text{wp}("k := k + 1", P_{\text{inv}}) : \\
\end{align*}
\]

Similarly, we can rewrite $P_{\text{inv}} \land G$ to expose the same term in the quantifier:

\[
(\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \land (\forall i \mid k \leq i < n : y(i) = \hat{y}(i)) \land 0 \leq k \leq n \land k < n
\]

becomes

\[
(\forall i \mid 0 \leq i < k : y(i) = \hat{y}(i) + x(i)) \land y(k) = \hat{y}(k) \\
\land (\forall i \mid k + 1 \leq i < n : y(i) = \hat{y}(i)) \land 0 \leq k < n,
\]

which we can enter in the derivation of the loop body:
3.4. Developing a While Command

Homework 3.4.5.1

DO EXERCISE ON edX
Figure 3.9: Completed worksheet for adding vector $x$ to vector $y$ (Variant 1).
\{ Q : (\forall i \mid 0 \leq i < n : y(i) = \tilde{y}(i)) \land 0 \leq n \}

\{ wp(“S_I”, P_{inv}) : \}

\{ P_{inv} : (\forall i \mid 0 \leq i < k : y(i) = \tilde{y}(i)) \land (\forall i \mid k \leq i < n : y(i) = \tilde{y}(i) + x(i)) \land 0 \leq k \leq n \}

while do

\{ P_{inv} \land G : (\forall i \mid 0 \leq i < k : y(i) = \tilde{y}(i)) \land (\forall i \mid k \leq i < n : y(i) = \tilde{y}(i) + x(i)) \land 0 \leq k \leq n \}

wp(“S_U; k := k - 1”, P_{inv}) :

\{ S_U : wp(“k := k - 1”, P_{inv}) : \}

k := k - 1

\{ P_{inv} : (\forall i \mid 0 \leq i < k : y(i) = \tilde{y}(i)) \land (\forall i \mid k \leq i < n : y(i) = \tilde{y}(i) + x(i)) \land 0 \leq k \leq n \}

endwhile

\{ P_{inv} \land \neg G : (\forall i \mid 0 \leq i < k : y(i) = \tilde{y}(i)) \land (\forall i \mid k \leq i < n : y(i) = \tilde{y}(i) + x(i)) \land 0 \leq k \leq n \land \}

\{ \neg ( ) \}

\{ R : (\forall i \mid 0 \leq i < n : y(i) = \tilde{y}(i) + x(i)) \}

Figure 3.10: Blank worksheet for adding vector x to vector y (Variant 2).
In Summary:

- Given a precondition and postcondition, we can derive loop invariants.
- Given a loop invariant and the postcondition, we can derive the loop guard.
- Given the precondition and a loop invariant, we can derive the initialization command.
- Given a loop invariant and an update to the loop index, we can derive the loop body.

All steps are prescribed by the precondition and the postcondition!

3.5 Examples

3.5.1 Evaluating a polynomial

In this section, we look at a really nice example through a sequence of exercises. It may be a trivial example at first, too similar to previous examples and homeworks. But there is an interesting twist!

This exercise prepares you for the first programming exercise, in the next unit. Since we will use MATLAB to implement the algorithm, we switch to assuming that “indexing starts at one”, meaning that the elements of an array $p$ are accessed as $p(1)$, $p(2)$, $\cdots$, $p(n)$.

Now, consider the evaluation of a polynomial of degree $n - 1$ with coefficients stored in array $p$ of size $n$:

$$y = p(1) + p(2)x + p(3)x^2 + \cdots + p(n)x^{n-1} = (\sum i \mid 1 \leq i \leq n : p(i)x^{i-1}).$$

Since there are $n$ coefficients, the last term is $p(n)x^{n-1}$ and a typical term is $p(i)x^{i-1}$.

Now, let’s do what we did before, and partition the quantifier:

$$y = \underbrace{p(1) + p(2)x + \cdots + p(k-1)x^{k-2}}_{(\sum i \mid 1 \leq i < k : p(i)x^{i-1})} + \underbrace{p(k)x^{k-1} + p(k+1)x^k + \cdots + p(n)x^{n-1}}_{(\sum i \mid k \leq i \leq n : p(i)x^{i-1})} \land 1 \leq k \leq n + 1$$

$$= (\sum i \mid 1 \leq i < k : p(i)x^{i-1}) + (\sum i \mid k \leq i \leq n : p(i)x^{i-1}) \land 1 \leq k \leq n + 1$$

From this, we can derive two obvious loop invariants:

Invariant 1: $y = (\sum i \mid 1 \leq i < k : p(i)x^{i-1}) \land 1 \leq k \leq n + 1$,

which leads to an (algorithmic) Variant 1 that updates something like
\[ y := y + p(k) \times x^{k-1} \]
\[ k := k + 1 \]
in the loop body and

**Invariant 2:**  \[ y = (\sum i \mid k \leq i \leq n : p(i)x^{i-1}) \land 1 \leq k \leq n + 1 \]

which leads to Variant 2 that updates something like
\[ k := k - 1 \]
\[ y := y + p(k) \times x^{k-1} \]
in the loop body. (We don’t guarantee this is exactly correct. Remember that Dijkstra may be watching over your shoulder, so you would want to derive the update.)

The problem with the two variants mentioned above is that in each iteration of the loop, \( x^{k-1} \) must be evaluated, which could be expensive. If you have some experience programming, you may observe that it could be good to have a variable, \( z \), that holds \( x^{k-1} \) every time through the loop. But we don’t just guess at such things. We derive loop invariants. How do we systematically derive these loop invariants? Observe that
\[ y = (\sum i \mid 1 \leq i < k : p(i)x^{i-1}) + (\sum i \mid k \leq i \leq n : p(i)x^{i-1}) \land 1 \leq k \leq n + 1 \]
\[ = (\sum i \mid 1 \leq i < k : p(i)x^{i-1}) + (\sum i \mid k \leq i \leq n : p(i)x^{i-k}) \times x^{k-1} \land 1 \leq k \leq n + 1 \]
\[ = (\sum i \mid 1 \leq i < k : p(i)x^{i-1}) + (\sum i \mid k \leq i \leq n : p(i)x^{i-k}) \times x^{k-1} \land 1 \leq k \leq n + 1 \]
which then yields two loop invariants

**Invariant 3:**  \[ y = (\sum i \mid 1 \leq i < k : p(i)x^{i-1}) \land z = x^{k-1} \land 1 \leq k \leq n + 1, \]

which leads to Variant 3 that updates something like
\[ y := y + p(k) \times z \]
\[ k := k + 1 \]
\[ z := z \times x \]
in the loop body and

**Invariant 4:**  \[ y = (\sum i \mid k \leq i \leq n : p(i)x^{i-1}) \land z = x^{k-1} \land 1 \leq k \leq n + 1, \]

which leads to Variant 4 that updates something like
\[ k := k - 1 \]
\[ y := y + p(k) \times z \]
\[ z := z / x \]

Again, we are doing this “quick and dirty”. You will want to derive the details for the algorithms!

Now there is one more invariant hidden in the expression
\[ y = (\sum i \mid 1 \leq i < k : p(i)x^{i-1}) + (\sum i \mid k \leq i \leq n : p(i)x^{i-k}) \times z \land z = x^{k-1} \land 1 \leq k \leq n + 1, \]

namely
Invariant 5: \[ y = \left( \sum_{i} \mid k \leq i \leq n : p(i)x^{i-k} \right) \land 1 \leq k \leq n + 1. \]

It is this loop invariant that we allude at in the video and that we want you to derive as algorithmic Variant 5 and that you will implement as function `EvaluatePolynomialVariant5`, in the next unit.

### 3.5.2 At last, you write your first code! ✨ to edX

“... in order to drive home the message that this introductory programming course is primarily a course in formal mathematics, we see to it that the programming language in question has not been implemented on campus so that students are protected from the temptation to test their programs.”

– Edsger W. Dijkstra ✭ EWD1036

In this course, we have yet to program a single line of real code! The reason is that we don’t believe one should write programs unless one has been equipped to derive a program to be correct. And now you are ready!

The MATLAB Live Script mentioned in the next video can be found in Unit 3.5.2 on the edX platform. Follow the directions in that unit for uploading it to your MATLAB Online account.

In the following exercises, you will develop the loop guard, the initialization, and the loop body of Variant 5 corresponding to Invariant 5

Invariant 5: \[ y = \left( \sum_{i} \mid k \leq i \leq n : p(i)x^{i-k} \right) \land 1 \leq k \leq n + 1. \]

for computing the polynomial

\[
y = p(1) + p(2)x + p(3)x^2 + \cdots + p(n)x^{n-1} = \left( \sum_{i} \mid 1 \leq i \leq n : p(i)x^{i-1} \right).
\]

#### Homework 3.5.2.1

At the end of the last video, you were asked to derive the loop guard \( G \) from

\[
\begin{align*}
\{ & P_{\text{inv}} \land \neg G : y = \left( \sum_{i} \mid k \leq i \leq n : p(i)x^{i-k} \right) \land 1 \leq k \leq n + 1 \land \neg G \\
& R : \left( \sum_{i} \mid 1 \leq i \leq n : p(i)x^{i-1} \right) \}
\end{align*}
\]

Indicate which of the following is a correct loop guard \( G \) (there may be more than one...)

a) \( 1 < k \).

b) \( k < n \).

c) \( k \neq 1 \).

d) \( k \neq n \).

Enter a correct loop guard in the Live Script.

✨ DO EXERCISE ON edX
Homework 3.5.2.2  At the end of the last video, you were asked to derive the initialization command

\[
\begin{align*}
k &= \text{ } \\
y &= \\
\end{align*}
\]

to make

\[
\begin{align*}
\{Q : 0 \leq n \} \\
k := \\
y := \\
\{P_{\text{inv}} : y = (\sum_{i \mid k \leq i \leq n} p(i)x^{i-k}) \land 1 \leq k \leq n + 1 \}
\end{align*}
\]
correct. Indicate which of the following is a correct initialization. (There may be more than one...)

a) \[
\begin{align*}
k &= n \\
y &= p(n)
\end{align*}
\]
b) \[
\begin{align*}
k &= n + 1 \\
y &= 0
\end{align*}
\]
c) \[
\begin{align*}
k &= n \\
y &= 1
\end{align*}
\]
d) \[
\begin{align*}
k &= 0 \\
y &= 0
\end{align*}
\]
Enter a correct initialization command in the Live Script.
**Homework 3.5.2.3** At the end of the last video, you were asked to derive the commands in the loop body

\[
k = k - 1 \\
y =
\]

to make

\[
\{ P^\text{inv} \land G : y = (\sum_{i \mid k \leq i \leq n} p(i)x^{i-k}) \land 1 \leq k \leq n + 1 \land 1 < k \\
  k := k - 1 \\
y := ??? \\
\{ P^\text{inv} : y = (\sum_{i \mid k \leq i \leq n} p(i)x^{i-k}) \land 1 \leq k \leq n + 1 \}
\]
correct. Indicate which of the following is the correct choice for updating \( y \). Hint: derive it systematically!

- a) \( k := k - 1 \)
  \[
y := p(k) \times x^{k-1} + y
\]
- b) \( k := k - 1 \)
  \[
y := p(k) + y \times x
\]
- c) \( k := k - 1 \)
  \[
y := y + p(k - 2) \times x^{k-1}
\]
- d) \( k := k - 1 \)
  \[
y := y + p(k) \times x
\]

Enter a correct update to \( y \) in the Live Script and enjoy getting the right answer the first time!

DO EXERCISE ON edX

With these homeworks you have discovered what is known in the United States as Horner’s rule for evaluating a polynomial. It is what in practice is done to efficiently evaluate a polynomial. You may want
to read the Wikipedia entry for Horner’s method (Horner’s rule) (search for “Horner’s method”).

3.6 Enrichment to edX

3.6.1 A conversation with Prof. David Gries to edX

Watch Video on edX
Watch Video on YouTube

3.6.2 Dafny: a language and program verifier for functional correctness

“Dafny is a programming language with built-in specification constructs. The Dafny static program verifier can be used to verify the functional correctness of programs.”

We believe you will enjoy and appreciate learning about Microsoft’s Dafny language and program verifier. The following more advanced example for that project is closely related to what you learned this Week: The Verification Corner: Loop Invariants. This may lead you to investigate this project more [LINK].

3.7 Wrap Up to edX

3.7.1 Additional exercises to edX

In the second part of the course, we will focus on operations from linear algebra, first with vectors (stored in one dimensional arrays) and later with matrices (stored in two dimensional arrays). To get this started, we now apply what we have learned to the dot product (also known as the inner product) operations.

The dot product of vectors \(x\) and \(y\) of size \(n\) is often written as \(x^T y\), which links the operation to that of computing the matrix-matrix multiplication (product) of a row vector \(x^T\) with a column vector \(y\), viewed as \(1 \times n\) and \(n \times 1\) matrices, respectively. If you are a bit fuzzy on this operation, you may want to visit units 1.4.3 and 1.6.1-1.6.3 of our MOOC titled “Linear Algebra: Foundations to Frontiers” (LAFF) offered on edX.

The operation is defined by

\[
x^T y = (\sum i \mid 1 \leq i \leq n : x(i) \times y(i)),
\]

where we start indexing at one, since we will want to implement it again with MATLAB. (In Week 4, we will briefly revert back to starting our indexing at zero as we introduce a notation that avoids indexing altogether.) If the result is stored in variable \(d\), the postcondition is given by

\[
d = (\sum i \mid 1 \leq i \leq n : x(i) \times y(i)).
\]

Notice that

\[
d = (\sum i \mid 1 \leq i < k : x(i) \times y(i)) + (\sum i \mid k \leq i \leq n : x(i) \times y(i)) \land 1 \leq k \leq n + 1
\]
and also
\[
d = (\sum_{i \mid 1 \leq i \leq k} x(i) \times y(i)) + (\sum_{i \mid k < i \leq n} x(i) \times y(i)) \land 0 \leq k \leq n.
\]

The subtle difference is to which of the two parts the element indexed by \(k\) belongs.

**Homework 3.7.1.1** Identify which of the following are valid loop invariants for deriving a loop that computes the dot product:

a) \(d = (\sum_{i \mid 1 \leq i < k} x(i) \times y(i)) \land 1 \leq k \leq n + 1\).

b) \(d = (\sum_{i \mid k \leq i \leq n} x(i) \times y(i)) \land 1 \leq k \leq n + 1\).

c) \(d = (\sum_{i \mid 1 \leq i \leq k} x(i) \times y(i)) \land 0 \leq k \leq n\).

d) \(d = (\sum_{i \mid k < i \leq n} x(i) \times y(i)) \land 0 \leq k \leq n\).

**Homework 3.7.1.2** For the loop invariant
\[
d = (\sum_{i \mid 1 \leq i < k} x(i) \times y(i)) \land 1 \leq k \leq n + 1
\]
derive a correct program for computing \(x^T y\). You will want to use the worksheet in Figure 3.11 for this exercise.

**Homework 3.7.1.3** Implement the program from the last exercise using the Live Script in

\[
\text{LAFFPPfC} \rightarrow \text{Assignments} \rightarrow \text{Week3} \rightarrow \text{matlab} \rightarrow \text{DotVariant1.mltx}
\]

For additional instructions, see Homework 3.5.3.3 on the edX platform.
Make sure you get the right answer the first time!

There are actually eight algorithms that could result from the loop invariants in Homework 3.7.1.1: For each of the four invariants, you can update (increment or decrement) \(k\) before updating \(d\) or after updating \(d\):

\[
k := ??? \\
d := ???
\]

or

\[
d := ???
\]

\[
k := ???
\]
Figure 3.11: Blank worksheet.
It is actually instructional to derive all eight. For some programs, the update to \( d \) involves \( x(k) \) and \( y(k) \) and for others it involves \( x(k - 1) \) and \( y(k - 1) \) or \( x(k + 1) \) and \( y(k + 1) \). You can actually predict from the loop invariant which of these will happen. In Weeks 4-6 we abstract away from indexing details, and these distinctions disappear.
3.7.2 Summary

Developing an arbitrary command

Choose $S$ to satisfy the following worksheet:

\[
\begin{array}{l}
\{Q : \} \\
\{Q \Rightarrow \text{wp}("S", R) ? \} \\
\{\text{wp}("S", R) : \} \\
S : \\
\{R : \}
\end{array}
\]

Developing the skip command

Check if $Q \Rightarrow R$:

\[
\begin{array}{l}
\{Q : \} \\
\{Q \Rightarrow R ? \} \\
S : \text{skip} \\
\{R : \}
\end{array}
\]

Developing a simple assignment

Choose $E$ to satisfy the following worksheet:

\[
\begin{array}{l}
\{Q : \} \\
\{Q \Rightarrow \text{wp}("S", R) ? \} \\
\{\text{wp}("S", R) : \} \\
S : x := E \\
\{R : \}
\end{array}
\]

Developing a sequence of assignments

Choose $E, E(x), E(x,y)$, etc., to satisfy the following worksheet:
Developing an assignment to an array element

Isolate (by splitting the range) what is different in quantifiers that occur in the precondition and postcondition. Compare and contrast.

Developing an if command

Choose $G_0$, $G_1$, etc., until $Q \Rightarrow G_0 \land G_1 \land \cdots$. For each $G_i$ develop

Complete worksheet:
\[
\begin{align*}
&\{Q:\} \\
&\{Q \Rightarrow G_0 \lor G_1 \lor \cdots \} \\
\textbf{if} \\
&G_0 \rightarrow \\
&\{G_0 \land Q:\} \\
&\{G_0 \land Q \Rightarrow \wp("S_0", R)?:\} \\
&\{\wp("S_0", R):\} \\
&S_0:\ \\
&\{R:\} \\
\textbf{fi} \\
&G_1 \rightarrow \\
&\{G_1 \land Q:\} \\
&\{G_1 \land Q \Rightarrow \wp("S_1", R)?:\} \\
&\{\wp("S_1", R):\} \\
&S_1:\ \\
&\{R:\} \\
\textbf{\cdots} \\
\textbf{fi}
\end{align*}
\]
Developing an if command (special case)

\[
\begin{align*}
&\{ Q : \\
&\quad Q \implies G_0 \lor G_1 ? \\
\text{if} \\
&\quad G_0 \rightarrow \\
&\quad \{ G_0 \land Q : \\
&\quad \{ G_0 \land Q \implies \wp("S_0", G_0 \land R_0) ? \\
&\quad \{ \wp("S_0", G_0 \land R_0) : \\
&\quad S_0 : \\
&\quad \{ G_0 \land R_0 : \\
&\text{[] } G_1 \rightarrow \\
&\quad \{ G_1 \land Q : \\
&\quad \{ G_1 \land Q \implies \wp("S_1", G_1 \land R_1) ? \\
&\quad \{ \wp("S_1", G_1 \land R_1) : \\
&\quad S_1 : \\
&\quad \{ G_1 \land R_1 : \\
\text{fi} \\
&\quad \{ R : (G_0 \land R_0) \lor (G_1 \land R_1) \}
\end{align*}
\]

Developing a while loop

For a loop over an array or multiple arrays, specified by postcondition \( R \) with one or more quantifiers:

- Determine a logical loop index.
- Split the quantifier(s) in \( R \) using the loop index.
- Determine one or more loop invariants from the resulting predicate by answering the question “What constitutes partial progress towards the result.”
- Pick a loop invariant that results.
- Determine the loop guard.
- Determine an initialization step.
• Determine whether to increment or decrement the loop counter in the loop body and whether to do so at the top or bottom of the loop body.

• Derive the remainder of the loop body.

Assuming the loop index is updated at the bottom of the loop body, use the worksheet given on the next page.
\[
\begin{align*}
Q : & \\
wp("S_I", P_{inv}) : & \\
S_I : & \\
P_{inv} : & \\
\text{while do} & \\
P_{inv} \land G : & \\
wp("S_U; k := k \ 1", P_{inv}) : & \\
S_U : & \\
w_{p("k := k \ 1", P_{inv}) :} & \\
k := k \ 1 & \\
P_{inv} : & \\
\text{endwhile} & \\
P_{inv} \land \neg G : & \\
\neg ( & \\
R : & \\
\end{align*}
\]
3.7.3 Why Dijkstra received the ACM Turing Award

Dijkstra received the ACM Turing Award in 1972

“For fundamental contributions to programming as a high, intellectual challenge; for eloquent insistence and practical demonstration that programs should be composed correctly, not just debugged into correctness; for illuminating perception of problems at the foundations of program design.”

What you hopefully have noticed in this week is that the thought process behind developing different parts of a program can be made remarkably systematic. Key is thinking about programs themselves in a structured way (which is why the advent of “structured programming” in the late 1950s was so important). In this course, we then structure goal-oriented programming with “worksheets” for different occasions.