Substitution in Arrays

- We haven't discussed how to understand array assignments yet. If we assign an expression to $b[e]$, the index e can only be evaluated at runtime; which can lead to the following problem.

  Assuming $b[j]$ and $b[k]$ are safe and $j \not\equiv k$, then what is $wp(b[j] := b[j] + 1, b[k] < b[j])$?
  - Some might say it is $b[k] < b[j] + 1$ but what if k and j evaluated to the same integer at runtime? Will we only substitute $b[j]$ with $b[j] + 1$? Or will we substitute both $b[j]$ and $b[k]$ with $b[j] + 1$ (and if so, we will get the weakest precondition being $b[j] + 1 < b[j] + 1 \Leftrightarrow F$)?

- We need to figure out how to understand substitutions in arrays $(e)[e_1 \,/\, b[e_0]]$ first.

1. How to understand $(b[m])[6 \,/\, d[2]]$ where m is a variable or named constant?
   - If b and d are different arrays ($b \not\equiv d$), then this is simple: there will be no expression $d[2]$ in array $b[m]$ then $(b[m])[6 \,/\, d[2]] \equiv b[m]$.
   - If b and d are the same array ($b \equiv d$), then we need to consider whether $m = 2$: if $m = 2$ then $(b[m])[6 \,/\, d[2]] \equiv 6$ or else it$\equiv$ $b[m]$.

2. How to understand $(b[e])[6 \,/\, d[2]]$ where e is an expression?
   - If b and d are different arrays, then $b[e] \not\equiv d[2]$ and we need to look recursively into e since expression $d[2]$ might appear in e; then, $(b[e])[6 \,/\, d[2]] \equiv b\left[e[6 \,/\, d[2]]\right]$.
   - If b and d are the same array, we need to evaluate e. If e is evaluated to 2 at run time, then $(b[e])[6 \,/\, d[2]] \equiv 6$, or else we will look recursively into e like in the above case).

- Here we give the definition of syntactic substitution in arrays:

  $(b[e_2])\ [e_1 \,/\, b[e_0]] \equiv$ **if** $e_2' = e_0$ **then** e_1 **else** $b[e_2']$ **fi**, where $e_2' \equiv (e_2)[e_1 \,/\, b[e_0]]$.

  - This definition covers all cases in Example 1 and 2 while the substitution happens in the same array:
    - When e_2 is a named constant, aka $e_2 \equiv k$, then we get $e_2' \equiv k[e_1 / \, b[e_0]] \equiv k$, and then $(b[k])\ [e_1 \,/\, b[e_0]] \equiv$ **if** $k = e_0$ **then** e_1 **else** $b[k]$ **fi.**

3. Finish the following syntactic substitutions.
   a) $(b[k])[5 \,/\, b[0]] \equiv$ **if** $k = 0$ **then** 5 **else** $b[k]$ **fi**
   b) $(b[k])[e_0 \,/\, b[j]] \equiv$ **if** $k = j$ **then** e_0 **else** $b[k]$ **fi**
   c) $(b[k])[b[j] + 1 \,/\, b[j]] \equiv$ **if** $k = j$ **then** $b[j] + 1$ **else** $b[k]$ **fi**

      Note that, we will keep e_1 (in this case $b[j] + 1$) as it is, even if it involves b.

   d) $(b[k])\ \left[b[j] \,/\, b\left[b[k]\right]\right] \equiv$ **if** $k = b[k]$ **then** $b[j]$ **else** $b[k]$ **fi**

   e) $(b[b[k]])[5 \,/\, b[0]]$

      The inner $b[k]$ need to be taken care first, and $(b[k])[5 \,/\, b[0]] \equiv$ **if** $k = 0$ **then** 5 **else** $b[k]$ **fi**. Then,

      $(b[b[k]])[5 \,/\, b[0]]$ $\qquad \not\equiv b[k]$

      $\equiv$ **if** (**if** $k = 0$ **then** 5 **else** $b[k]$**fi**) $= 0$ **then** 5 **else** $b[$**if** $k = 0$ **then** 5 **else** $b[k]$**fi**$]$ **fi**

      $\mapsto$ **if** $k = 0$ **then** $b[5]$ **else** **if** $b[k] = 0$ **then** 5 **else** $b[b[k]]$ **fi** **fi**

      $\Leftrightarrow$

- In Example 3.e) we "logically simplified" a complicated expression to a "shorter" expression. Formally, we call this operation **optimization**, it means we replace an expression/predicate with a "shorter" expression/predicate that is semantically equal. It is written as $e_1 \mapsto e_2$ ("e_1 optimizes to e_2").
  - We introduce this definition here because syntactic substitutions in arrays usually end up with a long and complicated text (either expression or predicate). It can be useful to shorten it first before execution, similarly to how compilers can optimize code.
  - The optimization is done in a static way: the optimization is done before the code runs.

  - Since to $e_1 \mapsto e_2$ we need $e_1 \Leftrightarrow e_2$, it is okay to just use "$\Leftrightarrow$" to represent optimization. But "$e_1 \Leftrightarrow e_2$" emphasizes that e_1 and e_2 are semantically equal, and "$e_1 \mapsto e_2$" emphasizes that e_1 is (or can be) optimized to e_2.

4. Let's look at a simple example before we introduce the rules. Optimize the following expressions.
   a) $(b[0])[e_1 \,/\, b[2]] \equiv \textbf{if } 0 = 2 \textbf{ then } e_1 \textbf{ else } b[0] \textbf{ fi} \mapsto b[0]$
   b) $(b[1])[e_1 \,/\, b[1]] \equiv \textbf{if } 1 = 1 \textbf{ then } e_1 \textbf{ else } b[1] \textbf{ fi} \mapsto e_1$

<u>Rules for Optimizing Condition Expressions</u>

Let's identify some general rules for optimizing conditional expressions and predicates involving them. This will let us simplify calculation of wlp or wp for array assignments.

- $(\textbf{if } T \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi}) \mapsto e_1$
- $(\textbf{if } F \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi}) \mapsto e_2$
- $(\textbf{if } B \textbf{ then } e \textbf{ else } e \textbf{ fi}) \mapsto e$
- If we know that $B \Rightarrow e_1 = e_2$, then $(\textbf{if } B \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi}) \mapsto e_2$.
  - Since B can imply that $e_1 = e_2$, then no matter whether B is true or not, we always have e_2.
- If we know that $\neg B \Rightarrow e_1 = e_2$, then $(\textbf{if } B \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi}) \mapsto e_1$.

Let op_1 be a unary operator, such as "$\neg$"...; and op_2 be a binary operator such as "$+$", "$<$"...
- $op_1\ (\textbf{if } B \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi}) \mapsto \textbf{if } B \textbf{ then } op_1(e_1) \textbf{ else } op_1(e_2) \textbf{ fi}$
- $(\textbf{if } B \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi})\ op_2\ e_3 \mapsto \textbf{if } B \textbf{ then } e_1\ op_2\ e_3 \textbf{ else } e_2\ op_2\ e_3 \textbf{ fi}$
- $b[\textbf{ if } B \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi }] \mapsto \textbf{if } B \textbf{ then } b[e_1] \textbf{ else } b[e_2] \textbf{ fi}$
- $f(\textbf{if } B \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi}) \mapsto \textbf{if } B \textbf{ then } f(e_1) \textbf{ else } f(e_2) \textbf{ fi}$

Let B, B_1, B_2 be Boolean expression.
- $(\textbf{if } B \textbf{ then } B_1 \textbf{ else } B_2 \textbf{ fi}) \mapsto (B \wedge B_1) \vee (\neg B \wedge B_2)$
- $(\textbf{if } B \textbf{ then } B_1 \textbf{ else } B_2 \textbf{ fi}) \mapsto (B \rightarrow B_1) \wedge (\neg B \rightarrow B_2)$

- $(\textbf{if } B \textbf{ then } B_1 \textbf{ else } F \textbf{ fi}) \mapsto (B \wedge B_1)$
  - $(\textbf{if } B \textbf{ then } B_1 \textbf{ else } F \textbf{ fi}) \Leftrightarrow (B \wedge B_1) \vee (\neg B \wedge F) \Leftrightarrow (B \wedge B_1)$
- $(\textbf{if } B \textbf{ then } F \textbf{ else } B_2 \textbf{ fi}) \mapsto (\neg B \wedge B_2)$

- $(\textbf{if } B \textbf{ then } B_1 \textbf{ else } T \textbf{ fi}) \mapsto (B \rightarrow B_1)$
- $(\textbf{if } B \textbf{ then } B_1 \textbf{ else } T \textbf{ fi}) \mapsto (\neg B \vee B_1)$
  - $(\textbf{if } B \textbf{ then } B_1 \textbf{ else } T \textbf{ fi}) \Leftrightarrow (B \rightarrow B_1) \wedge (\neg B \rightarrow T) \Leftrightarrow (B \rightarrow B_1)$
- $(\textbf{if } B \textbf{ then } T \textbf{ else } B_2 \textbf{ fi}) \mapsto (\neg B \rightarrow B_2)$
- $(\textbf{if } B \textbf{ then } T \textbf{ else } B_2 \textbf{ fi}) \mapsto (B \vee B_2)$

Now, let's go back to the first question of the class.

5. Let j and k be two named constants that are at least 0 and less than $size(b)$. Calculate $wp(b[j] \coloneqq b\,[j] + 1, b[k] < b[j])$, assuming $b[j]$ and $b[k]$ are safe.

   $wp(b[j] \coloneqq b\,[j] + 1, b[k] < b[j])$
   $\equiv (b[k] < b[j])\,[b\,[j] + 1\,/\,b\,[j]]$
   $\equiv (b[k])\,[b\,[j] + 1\,/\,b\,[j]]\; < (b[j])\,[b\,[j] + 1\,/\,b\,[j]]$
   $\equiv\; (\textbf{if } k = j \textbf{ then } b\,[j] + 1 \textbf{ else } b[k] \textbf{ fi}) < (b[j] + 1)$
   $\mapsto \textbf{if } k = j \textbf{ then } (b\,[j] + 1) < (b[j] + 1) \textbf{ else } b[k] < (b[j] + 1) \textbf{ fi}$
   $\mapsto\; \textbf{if } k = j \textbf{ then } F \textbf{ else } b[k] < (b[j] + 1) \textbf{ fi}$
   $\mapsto\; k \neq j \wedge b[k] < (b[j] + 1)$

   - This gives us a valid triple $\{k \neq j \wedge b[k] < (b[j] + 1)\}\; b[j] \coloneqq b\,[j] + 1\; \{b[k] < b[j]\}$

6. Correct a full proof outline of a program that swaps the values of primitive-type variables x and y.
   - To swaps the values of x and y, we need the help of a temporary variable u, then we can create the following minimal proof outline:
     $$\{x = x_0 \wedge y = y_0\}\; u \coloneqq x; x \coloneqq y; y \coloneqq u\; \{x = y_0 \wedge y = x_0\}$$
   - We can keep using backward assignments to create the following full proof outline:
     $\{x = x_0 \wedge y = y_0\}\; u \coloneqq x; \{y = y_0 \wedge u = x_0\}\; x \coloneqq y; \{x = y_0 \wedge u = x_0\}\; y \coloneqq u\; \{x = y_0 \wedge y = x_0\}$

7. Create a full proof outline of a program that swaps $b[m]$ and $b[n]$, assuming that m and n are natural numbers less than $size(b)$.
   - Like question 6, we need to prove the following minimal proof outline:
     $$\{b[m] = c \wedge b[n] = d\}\; u \coloneqq b[m]; b[m] \coloneqq b[n]; b[n] \coloneqq u\; \{b[m] = d \wedge b[n] = c\}$$
   - If we keep using backward assignments, then we can come up with the following full proof outline:
     $$\{b[m] = c \wedge b[n] = d\}\; \{q_0\}\; u \coloneqq b[m]; \{q_1\}\; b[m] \coloneqq b[n]; \{q_2\}\; b[n] \coloneqq u\; \{b[m] = d \wedge b[n] = c\}$$

   Let's calculate q_2, q_1 and q_0. Note that, we also need to prove $b[m] = c \wedge b[n] = d \Rightarrow q_0$.

   $q_2 \;\equiv (b[m] = d \wedge b[n] = c)\,[u\,/\,b[n]]$
   $\equiv (b[m] = d)\,[u\,/\,b[n]] \wedge (b[n] = c)\,[u\,/\,b[n]]$
   $\equiv (b[m])\,[u\,/\,b[n]] = d \wedge (u = c)$
   $\equiv (\textbf{if } m = n \textbf{ then } u \textbf{ else } b[m] \textbf{ fi}) = d \wedge (u = c)$ # Stop here if pure syntactic result is needed

   $q_1 \;\equiv ((\textbf{if } m = n \textbf{ then } u \textbf{ else } b[m] \textbf{ fi}) = d \wedge (u = c))\,[b[n]\,/\,b[m]]$
   $\equiv (\textbf{if } m = n \textbf{ then } u \textbf{ else } b[m] \textbf{ fi})[b[n]\,/\,b[m]] = d \wedge (u = c)$
   $\equiv (\textbf{if } m = n \textbf{ then } u \textbf{ else } b[n] \textbf{ fi}) = d \wedge (u = c)$

   $q_0 \equiv ((\textbf{if } m = n \textbf{ then } u \textbf{ else } b[n] \textbf{ fi}) = d \wedge (u = c))\,[b[m]\,/\,u]$
   $\equiv (\textbf{if } m = n \textbf{ then } b[m] \textbf{ else } b[n] \textbf{ fi}) = d \wedge (b[m] = c)$
   # Let's try to optimize q_0.
   # $m = n$ implies $b[m] = b[n]$
   $\mapsto b[n] = d \wedge (b[m] = c)$

   It is obvious that the precondition $b[m] = c \wedge b[n] = d \Leftrightarrow q_0$ so the proof is done (and we can remove the condition q_0).

<u>Parallel Program Basics</u>

- A parallel program is trying to run all different threads "at the same time". In our language, the syntax of a parallel statement/program with n threads is $S \equiv [S_1 \parallel S_2 \parallel \cdots \parallel S_n]$. We say $[S_1 \parallel S_2 \parallel \cdots \parallel S_n]$ is the **parallel composition** of **threads** $S_1, S_2, \dots, S_n$.
  - Each thread S_i in the composition should be non-parallel and deterministic: it is not legal to wright $S \equiv [S_1 \parallel [S_2 \parallel S_3]]$.

- Before we formally define the semantics of parallel programs, let's use a simple example to see the difference between sequential, parallel, and nondeterministic conditional programs.

8. Find a postcondition for each of the following valid triples.
   a) $\{x = 5\}\ x \coloneqq x + 1; x \coloneqq x * 2\ \{q\}$
   It is quite easy to see that $x = 12$ is a valid postcondition: we will finish two assignments in the given order. It is almost the strongest postcondition, we only omitted the initial value of x compared to $x_0 = 5 \wedge x_1 = x_0 + 1 \wedge x = x_1 * 2$.

   b) $\{x = 5\}\ \mathbf{if}\ T \to x \coloneqq x + 1 \ \Box\ T \to x \coloneqq x * 2\ \mathbf{fi}\ \{q\}$
   Both arms have true guard, so we will execute two branches at the same time with equal probability. Thus, the postcondition is $x = 6 \vee x = 10$. As an aside, the strongest postcondition is $x_0 = 5 \wedge x = x_0 + 1 \vee x_0 = 5 \wedge x = x_0 * 2$.

   c) $\{x = 5\}\ [x \coloneqq x + 1 \parallel x \coloneqq x * 2]\ \{q\}$
   Both threads will be executed "at the same time"; but some thread must be executed faster than the other in real life, and threads will be executed in any possible order. Thus, we might have $x = 12$ if we execute $x \coloneqq x + 1$ first, or we might have $x = 11$ if we execute $x \coloneqq x * 2$ first. Thus, $x = 11 \vee x = 12$ is a valid postcondition here.

- The above example shows the difference between sequential, parallel, and nondeterministic programs.
  - For a sequential statement, we execute each unit statement in the given order.
  - For a nondeterministic $\mathbf{if} - \mathbf{fi}$ statement, we execute each arm at the same time with the same probability.
  - For a parallel statement, all unit statements in the composition will be executed in any possible order. So, parallel statements can be considered as a simulation of nondeterminism: $[x \coloneqq x + 1 \parallel x \coloneqq x * 2]$ can simulate $\mathbf{if}\ T \to x \coloneqq x + 1;\ x \coloneqq x * 2 \ \Box\ T \to x \coloneqq x * 2; x \coloneqq x + 1\ \mathbf{fi}.$