Auxiliary and Logical Variables

CS 536: Science of Programming, Fall 2019

A. Why

- Auxiliary variables help us reason about our programs without adding unnecessary computations.

B. Objectives

At the end of this lecture you should

- Recognize whether or not a set of variables is auxiliary for a program.
- Be able to add auxiliary variables to a program or remove auxiliary variables from a program, consistently.

C. Why Auxiliary Variables?

- We've used logical variables, which only appear in the correctness proof in:
  - The forward assignment rule to name the value a variable had before the assignment statement.
  - Program specifications to name the value a variable had when the program began.
- Since they only appear in proofs, we haven't been calculating the values of logical variables because it's clearly unnecessary to do so.
- Auxiliary variables are an extension of the notion of logical variables. Normally, we calculate the values of all of our program variables; with auxiliary variables, we won't.
- Auxiliary variables added to the program to enable a correctness proof but aren't relevant to the calculation of the values of variables we're actually interested in: Their actual values at runtime, however, don't affect the calculations that we're interested in. It's in that sense that auxiliary variables are unnecessary.
- To illustrate, consider forward assignment: $\{p \wedge \mathtt{x} = \mathtt{x}_0\}\ \mathtt{x} := e\ \{p[\mathtt{x}_0/\mathtt{x}] \wedge \mathtt{x} = e[\mathtt{x}_0/\mathtt{x}]\}$
  - Without introducing $\mathtt{x}_0$, we're kind of stuck for how to describe forward assignment.
  - What if we look at $\{p\}\ \mathtt{x}_0 := \mathtt{x};\ \{p \wedge \mathtt{x} = \mathtt{x}_0\}\ \mathtt{x} := e\ \{p[\mathtt{x}_0/\mathtt{x}] \wedge \mathtt{x} = e[\mathtt{x}_0/\mathtt{x}]\}$ // $\mathtt{x}_0$ fresh
  - The assignment $\mathtt{x}_0 := \mathtt{x}$ sets our "logical" variable but doesn't affect the calculation of $\mathtt{x} := e$.
  - We could calculation $\mathtt{x}_0$ at runtime, but why bother if all we're interested in is $\mathtt{x}$?
  - So we could argue that in some sense the assignment $\mathtt{x}_0 := \mathtt{x}$ doesn't really need to be executed because it doesn't affect $\mathtt{x} := e$.
  - We've had an implicit quantifier over $\mathtt{x}_0$ where the range of the quantifier is both conditions.
    - $\{\exists\, \mathtt{x}_0 . p \wedge \mathtt{x} = \mathtt{x}_0\}\ \mathtt{x} := e\ \{p[\mathtt{x}_0/\mathtt{x}] \wedge \mathtt{x} = e[\mathtt{x}_0/\mathtt{x}]\}$
- Here, $\mathtt{x}_0$ doesn't change once we set it. The extension to auxiliary variables will let us change variables like $\mathtt{x}_0$ as long as those changes don't affect the calculations we're interested in, so we'll still be able to avoid calculating their values.

Example 1

- In the program below, we search through `x`, `f(x)`, `f(f(x))`, `f(f(f(x)))`, … for the first value that meets property `P(x)`. For termination, let's assume that in this sequence, the difference between adjacent values

decreases: $|x - f(x)| > |f(x) - f(f(x))| > |f(f(x)) - f(f(f(x)))| \ldots \geq 0$.

```
x0 := x;     // previous value of x
x := f(x);   // new value of x
delta_x := x – x0;
{inv … } {bd |delta_x|}          // Writing |…| for absolute value
while ¬P(x) do
        … // computations that don't use x0 or delta_x
    // Update old x, current x, and delta x
    x0 := x;
    x := f(x);
    delta_x := x – x0
od
// After the loop, we don't use delta_x or x0.
```

- If `delta_x` isn't used anywhere (except in the bound function), then calculating its actual value doesn't really serve any purpose. Similarly, if x_0 is used only to calculate `delta_x`, then its value doesn't serve any useful purpose. We use `x` in `delta_x := x – x0`, but since we're not calculating `delta_x`, we can ignore the value of `x` here.
- We can't just treat x_0 and `delta_x` as named logical constants because they change over time. We can't just write the program without them, since we need `delta_x` for the bound function and x_0 for `delta_x`.
- `delta_x` and x_0 will be **auxiliary variables**: They're program variables, so we can discuss their logical properties, but they're like logical variables in that we won't rely on computing their values.

D. Auxiliary Variables

- **Definition**: Let *S* be a program and let $V = Vars(S)$. A set of variables $A \subseteq V$ is an **auxiliary set** (for *S*) if:
  - All computations in *S* of values in $V - A$ depend only on variables in $V - A$ and
  - All boolean tests in *S* use only variables from $V - A$.
    - The empty set is trivially auxiliary, and if *S* includes no boolean tests, then *V* is trivially auxiliary.
- **Definition**: The **required variables** (with respect to *A*) are the ones in $V - A$. The presumption is that we're interested in actually calculating the values of required variables, either directly or indirectly. E.g., if we're interested in `x`, then having assignments like `x := y` and `y := z` force us to be interested in `y` and `z` too.
  - We can get away with not actually calculating and storing the values of auxiliary variables because their values can't affect the values of required variables.
- **Definition**: A variable of *S* is a **primary variable** if it is not a member of any auxiliary set of variables for *S*.
  - All variables that appear in tests are primary, as are the variables needed to calculate their values, directly and indirectly. (I.e, if `x` is primary, then `x := y` and `y := z` force `y` and `z` to be primary also.)
- **Notation:** To indicate in a program that we intend a variable to be auxiliary, we'll parenthesize it. In Example 1, we would write `(x0) := x;` and `(delta_x) := x – (x0);` (We can omit parenthesizing them in conditions.)
- **Definition**: An **auxiliary labeling** for a program tells us which program variables are auxiliary vs required.

- **Definition**: An auxiliary labeling is **consistent** if
  - No auxiliary variable appears in a **`if`** or **`while`** test and
  - For every assignment statement v **`:=`** e, if v is required then all the variables of e are also required.
    - Contrapositively, if any variable in e is auxiliary, then v must be auxiliary.
- A case analysis shows us which usages of auxiliary variables are allowed and which are disallowed. Here, a and a' are auxiliary and r and r' are required.
- **Allowed**:
  - (a) **`:=`** … r … (a') … — If lhs is auxiliary, rhs can include auxiliary and required variables. (We say that a' **forces a dependency** (from a') to a.)
  - r **`:=`** … r' … — If lhs is required, rhs can include required variables
  - **`if`** / **`while`** … r … — Required variables can appear in tests
- **Disallowed**:
  - r **`:=`** … (a) … — If lhs is required, rhs cannot include auxiliary variables
  - **`if`** / **`while`** … (a) … — Auxiliary variables cannot appear in tests
- **Expanding an auxiliary labeling.** Let's call a labeling **fully expanded** if it includes all labels of the form (a) **`:=`** … (a') … (i.e., in v **`:=`** e, if there's an auxiliary variable in e, then v is labeled auxiliary. There's a simple algorithm for fully expanding a starting set of variables: As long as the the program contains an assignment notated y **`:=`** … (x) …., mark all occurrences of y as (y). A fully-expanded labeling is consistent iff it has no occurrences of auxiliary variables in **`if`** or **`while`** tests.

Example 2

- Let's expand the initial labeling {`v`} for the following program. We start with

  `x := y; y := (v)+w; if w ≥ 0 then x := x+1; w := w−1 fi`
- Because of `y := (v)`…, mark `y`:

  `x := (y); (y) := (v)+w; if w ≥ 0 then x := x+1; w := w−1 fi`
- Because of `x := (y)`…, mark `x`:

  `(x) := (y); (y) := (v)+w; if w ≥ 0 then (x) := (x)+1; w := w−1 fi`
- No more variables need to be marked as auxiliary, and there are no disallowed uses of auxiliary variables, so {`v`, `x`, `y`} is a consistent set of auxiliary variables.
- More generally for this program, the assignments `x := y` and `y := v+w` generate the following dependencies: `y` (being auxiliary) forces `x` (to be auxiliary), and `v` forces `y`. (The assignment `x := x+1` makes `x` force `x`, which is trivial, and since `w` appears in the test, it's primary, so it doesn't matter that `y := v+w` makes `w` force `y`.) Altogether, there are three consistent labelings.
  - `(x) := y;  y := v+w;  if w ≥ 0 then (x) := (x)+1; w := w−1 fi` // {`x`} auxiliary
  - `(x) := (y); (y) := v+w;  if w ≥ 0 then (x) := (x)+1; w := w−1 fi` // {`x`, `y`} auxiliary
  - `(x) := (y); (y) := (v)+w; if w ≥ 0 then (x) := (x)+1; w := w−1 fi` // {`v`, `x`, `y`} auxiliary
- In the other direction, since three of the $2^4 - 1 = 15$ nontrivial labelings are consistent, the other twelve are inconsistent:

- Since **if** (w) … appears, it is primary and all eight labelings that include w are inconsistent.
- Since x := (y) is inconsistent, {y} and {v, y} are inconsistent.
- Since y := (v)+w is inconsistent, {v} and {v, x} are inconsistent.

Example 3

- Consider the program y := r; **while** t > 1 **do** y := y*t; t := t–k **od** .
- The consistent labelings are
  - (y) := r; **while** t > 1 **do** (y) := (y)*t; t := t–k **od** // {y} auxiliary
  - (y) := (r); **while** t > 1 **do** (y) := (y)*t; t := t–k **od** // {r, y} auxiliary
- For inconsistent labelings
  - From **while** t …, we know that no labeling can include t.
  - From t := t–(k) and (t) := (t)–(k), we know that no labeling can include k.
    - (Since no labeling with t is consistent, (t) := (t)–(k) is inconsistent.)
  - From y := (r), we know that r without y is inconsistent.

Example 4

- Let's go back to the x_0 and delta_x program from Example 1. (To save space, I've compressed it and removed the **inv** and **bd** headers.)

  x_0 := x; x := f(x); delta_x := x – x_0;
  while ¬P(x) **do** x_0 := x; x := f(x); delta_x := x – x_0 **od**

- Since x appears in the **while** test, it must be primary. The assignment delta_x := x – x_0 forces a dependency from x_0 to delta_x, but delta_x is used by any assignment, so it forces no dependencies.
- There are two consistent labelings. One is {delta_x} and {delta_x, x_0}.

  x_0 := x; x := f(x); (delta_x) := x – x_0;
  while ¬P(x) **do** x_0 := x; x := f(x); (delta_x) := x – x_0 **od**

- The other consistent labeling is {delta_x, x_0}.

  (x_0) := x; x := f(x); (delta_x) := x – (x_0);
  while ¬P(x) **do** (x_0) := x; x := f(x); (delta_x) := x – (x_0) **od**

Example 5:

- As a general example of using auxiliary variables, let's consider the following disjoint parallel program without disjoint conditions. (Note x and y do not in appear e_2 and e_1 respectively.)

  $\{p_1(x, y) \wedge q_1(x, y)\}$
  $[\ \{p_1(x, y)\}\ x := e_1(x)\ \{p_2(x, y)\}$
  $\|\ \{q_1(x, y)\}\ y := e_2(y)\ \{q_2(x, y)\}$
  $]\ \{p_2(x, y) \wedge q_2(x, y)\}$

  To prove this using disjoint parallelism, we have to make the programs have disjoint conditions. To do this, we'll introduce auxiliary variable X to discuss the value of x in thread 2's conditions and Y to discuss y in thread 1's conditions.

$\{p_1(\mathtt{x}, \mathtt{y}) \wedge q_1(\mathtt{x}, \mathtt{y})\}$
$\mathtt{X := x;\ Y := y;}$
$\{p_1(\mathtt{x}, \mathtt{Y}) \wedge q_1(\mathtt{X}, \mathtt{y})\}$
$[\ \{p_1(\mathtt{x}, \mathtt{Y})\}\ x := e_1(x)\ \{p_2(\mathtt{x}, \mathtt{Y})\}$
$\|\ \{q_1(\mathtt{X}, \mathtt{y})\}\ y := e_2(y)\ \{q_2(\mathtt{X}, \mathtt{y})\}$
$]\ \{p_2(\mathtt{x}, \mathtt{Y}) \wedge q_2(\mathtt{X}, \mathtt{y})\}$

- Note the conclusion has been changed from $p_2(\mathtt{x}, \mathtt{y}) \wedge q_2(\mathtt{x}, \mathtt{y})$ to $p_2(\mathtt{x}, \mathtt{Y}) \wedge q_2(\mathtt{X}, \mathtt{y})$. Getting back the original $p_2(\mathtt{x}, \mathtt{y}) \wedge q_2(\mathtt{x}, \mathtt{y})$ will depend on the particular definitions of the conditions and expressions.

Example 6

- Let's look at a concrete instance of Example 5, starting with

  $\{\mathtt{x-y = d}\}\ [\mathtt{x := x+1} \| \mathtt{y := y+1}]\ \{\mathtt{x-y = d}\}$

- Neither thread itself maintains $\mathtt{x-y = d}$ by itself; it's only the combination that does, so we cannot just make $\mathtt{x-y = d}$ the preconditions and postconditions of the threads.
- We can start following the pattern of Example 5:

  $\{\mathtt{x-y = d}\}\ \mathtt{X := x;\ Y := y}\ \{\mathtt{x-Y = d} \wedge \mathtt{x-Y = d}\}$
  $[\ \{\mathtt{x-Y = d}\}\ \mathtt{x := x+1}\ \{\ ???\ \}$
  $\|\ \{\mathtt{x-Y = d}\}\ \mathtt{y := y+1}\ \{\ ???\ \}$
  $]\ \{??? \wedge ???\}\ \{\mathtt{x-y = d}\}$

- We need to figure out the missing conditions. If we use *sp* on each thread, we get
  - $\{\mathtt{X = x} \wedge \mathtt{X-Y = d}\}\ \mathtt{x := x+1}\ \{\mathtt{x = X+1} \wedge \mathtt{X-Y = d}\}$
  - $\{\mathtt{Y = y} \wedge \mathtt{X-Y = d}\}\ \mathtt{y := y+1}\ \{\mathtt{y = Y+1} \wedge \mathtt{X-Y = d}\}$
- Since $(\mathtt{x = X+1} \wedge \mathtt{X-Y = d})$ and $(\mathtt{y = Y+1} \wedge \mathtt{X-Y = d})$ implies $(\mathtt{x - y = (X+1) - (Y+1) = X-Y = d})$, we can combine the two threads and get

  $\{\mathtt{x-y = d}\}\ \mathtt{X := x;\ Y := y}\ \{\mathtt{X = x} \wedge \mathtt{X-Y = d} \wedge \mathtt{Y = y} \wedge \mathtt{X-Y = d}\}$
  $[\ \{\mathtt{X = x} \wedge \mathtt{X-Y = d}\}\ \mathtt{x := x+1}\ \{\mathtt{x = X+1} \wedge \mathtt{X-Y = d}\}$
  $\|\ \{\mathtt{Y = y} \wedge \mathtt{X-Y = d}\}\ \mathtt{y := y+1}\ \{\mathtt{y = Y+1} \wedge \mathtt{X-Y = d}\}$
  $]\ \{\mathtt{x = X+1} \wedge \mathtt{X-Y = d} \wedge \mathtt{y = Y+1} \wedge \mathtt{X-Y = d}\}$
  $\{\mathtt{x-y = d}\}$

- We use X and Y here in the conditions of the threads but not the code, so they can be seen as logical variables:

  $\{\mathtt{X = x} \wedge \mathtt{Y = y} \wedge \mathtt{x-y = d}\}$ … *program*… $\{\mathtt{x-y = d}\}$

E. Removing Auxiliary Variables

- We need to connect the behavior of programs with and without auxiliary variables. It turns out to be easier to discuss the behavior of removing auxiliary variables instead of adding them, so we'll do it that way.
- **Definition**: Let *S* be a program and *A* be a set of auxiliary variables. Then *S* - *A* ("*S with A removed*") is *S* where where each assignment to a variable in *A* has been replaced by a **`skip`** statement.

- If desired, we can optimize S - A by changing **skip;** S and S**; skip** by just S, repeating until this can't be done. There's also the optimization of changing **if** B **then skip else skip fi** to **skip** (if B cannot cause a runtime error).

Example 7

- Going back to the program and labelings of Examples 1 and 4, we had two consistent labelings: {delta_x} gave

  x_0 := x; x := f(x); (delta_x) := x – x_0;
  while ¬P(x) **do** x_0 := x; x := f(x); (delta_x) := x – x_0 **od**

  Removal gives

  x_0 := x; x := f(x); **skip;**
  while ¬P(x) **do** x_0 := x; x := f(x); **skip od**

  which optimizes to

  x_0 := x; x := f(x); **while** ¬P(x) **do** x_0 := x; x := f(x) **od**

- The other consistent labeling was {delta_x, x_0}:

  (x_0) := x; x := f(x); (delta_x) := x – (x_0);
  while ¬P(x) **do** (x_0) := x; x := f(x); (delta_x) := x – (x_0) **od**

  Removal gives

  skip; x := f(x); **skip;**
  while ¬P(x) **do skip;** x := f(x); **skip od**

  which optimizes to

  x := f(x); **while** ¬P(x) **do** x := f(x) **od**

Example 8

- Let's go back to Example 2, where we had a program with three auxiliary labelings.
- First was the labeling {x}. Marking, removing, and optimizing gives
  - $S \equiv$ (x) := y; y := v+w; **if** w ≥ 0 **then** (x) := (x)+1; w := w–1 **fi**
  - S – {x} ≡ **skip;** y := v+w; **if** w ≥ 0 **then skip;** w := w–1 **fi**
  - S – {x} after optimization: y := v+w; **if** w ≥ 0 **then** w := w–1 **fi**
- For {x, y} we get
  - $S \equiv$ (x) := (y); (y) := v+w; **if** w ≥ 0 **then** (x) := (x)+1; w := w–1 **fi**
  - S – {x} ≡ **skip; skip; if** w ≥ 0 **then skip;** w := w–1 **fi**
  - S – {x} after optimization: **if** w ≥ 0 **then** w := w–1 **fi**
- For {v, x, y}, we get a different marking from {x, y} but the same results after removal and optimization:
  - $S \equiv$ (x) := (y); (y) := (v)+w; **if** w ≥ 0 **then** (x) := (x)+1; w := w–1 **fi**
  - S – {x} ≡ **skip; skip; if** w ≥ 0 **then skip;** w := w–1 **fi**
  - S – {x} after optimization: **if** w ≥ 0 **then** w := w–1 **fi**

F. Execution of Programs With Auxiliary Variables — Skip

- The goal is to argue that removing auxiliary variables from a program does not change how the program works on required variables. To phrase this, it helps to start with a lemma about a single operational semantics step (→), which makes it easy to go to overall operational semantics (→*).
- **Lemma** (**Preservation of Required Computation**): Let S be a program with auxiliary and required variables A and R. Let $\sigma \cup \tau$ be a state for S where σ covers R and τ covers A. (I.e., their domains are A and R respectively) Then if $\langle S, \sigma \cup \tau\rangle \rightarrow \langle S', \sigma' \cup \tau'\rangle$, then $\langle S - A, \sigma\rangle \rightarrow^* \langle S' - A, \sigma'\rangle$
- **Proof**: For $S - A$, it's easier to look at the version before optimization (since optimization clearly doesn't change the semantics of $S - A$). Since S and $S - A$ differ only in $S - A$ having **`skip`** where S has assignments to auxiliary variables, this is the important case; **`if`** and **`while`** also have to be discussed; **`skip`** is trivial and omitted.

  Say S includes v **`:=`** e, then $\langle v$ **`:=`** $e, \sigma \cup \tau\rangle \rightarrow \langle E, (\sigma \cup \tau)[v \mapsto \alpha]\rangle$ where $\alpha = (\sigma \cup \tau)(e)$. If v is auxiliary, the update to $\sigma \cup \tau$ can only affect σ, so we have $\langle v$ **`:=`** $e, \sigma \cup \tau\rangle \rightarrow \langle E, \sigma \cup \tau[v \mapsto \alpha]\rangle$. The corresponding execution in $S - A$ is $\langle$**`skip`**$, \sigma\rangle \rightarrow \langle E, \sigma\rangle$, so the programs behave the same way on R.

  For **`if`** and **`while`** statements, removing A does not change the tests in **`if`** B and **`while`** B, so S jumps depending on $(\sigma \cup \tau)(B)$ and $S - A$ jumps depending on $\sigma(B)$. But B contains only required variables, so $(\sigma \cup \tau)(B) = \sigma(B)$, so the behavior in S and $S - A$ are the same on R. //
- **Theorem** (**Preservation of Required Computations**): Let A and R be auxiliary and required variables for S. Let σ cover R, and let τ cover A. Then if $\langle S, \sigma \cup \tau\rangle \rightarrow^* \langle S', \sigma' \cup \tau'\rangle$, then $\langle S - A, \sigma\rangle \rightarrow^* \langle S' - A, \sigma'\rangle$.

  Proof: Simply iterate the lemma above to cover the number of execution steps for S.

G. Proof Rules Involving Auxiliary Variables — Just worry about proof rule

- Now that we understand the semantics of adding and removing auxiliary variables, we can formalize these as sound proof rules.
- **Theorem (Preservation of Validity)**: Let $\models \{p\}\ S'\ \{q\}$ with auxiliary and required variables A and R. If no variables of A are free in p and q, then $\models \{p\}\ S' - A\ \{q\}$.

  Proof: Let $\sigma \models p$ be a state that covers R, and let τ cover A so that $\sigma \cup \tau$ is a state for S. Say $\langle S, \sigma \cup \tau\rangle \rightarrow^* \langle E, \sigma' \cup \tau'\rangle$ where σ' and τ' cover A and R. By the theorem on preservation, $\langle S - A, \sigma\rangle \rightarrow^* \langle E - A, \sigma'\rangle$. For satisfaction of q, validity of $\{p\}\ S\ \{q\}$ implies $\sigma' \cup \tau' \models q$. Since q depends only on R, this implies $\sigma' \models q$. So $\sigma \models \{p\}\ S - A\ \{q\}$.

Auxiliary Variable Removal

1. $\{p\}\ S\ \{q\}$
2. $\{p\}\ S - A\ \{q\}$ Auxiliary variable removal, 1, A

where $S - A$ is the A-auxiliary contraction of S and no free variables of p and q appear in A.

Auxiliary and Logical Variables

CS 536: Science of Programming

A. Why

- Auxiliary variables help us reason about our programs in a way that lets us recognize and remove unnecessary computations.

B. Objectives

At the end of this activity you should be able to

- Recognize whether or not a set of variables is auxiliary for a program.
- Remove a set of auxiliary variables from a program.

C. Problems

1. √Compare auxiliary variables to program variables and logical variables. Make your discussion brief (say, at most one sentence for each of the three kinds of variables).
2. If A_1 and A_2 are both sets of auxiliary variables, is $A_1 \cup A_2$ also auxiliary?
3. For each type of variable below, say whether they can always / never / sometimes be used as auxiliary variables. Explain briefly.
   a. A **write-only variable** is one that is assigned to but its value is never used.
   b. A **read-only variable** already has a value when the program starts; the value may be looked at but not changed.
4. Consider the following program

```
x := y+z; u := v; w := u; if w > 0 then w := w–1; y := y*z fi
```

   a. √Which of `u`, `v`, `w`, `x`, `y`, `z` are primary (cannot be auxiliary under any consistent labeling)?
   b. √What auxiliary labelings do you get if you start with the following sets of possible auxiliary variables?

{x}, {y}, {z}, {x, y}, {x, z}, {y, z}, {x, y, z}

   c. √Which of the labelings in (b) are consistent? (Those are the legal sets of auxiliary variables).
   d. For each of the sets of auxiliary variables from part (c), what do you get if you remove the set from our program?
5. Consider the following program. (Note `n` is a constant.)

```
x := 1; y := 0;
while x < n do
      y := y+1; z := x*y; x := x+x
od
```

   a. Which of `x`, `y`, `z` are primary?
   b. What are the possible sets of auxiliary variables for this program?
   c. For each possible set of auxiliary variables, what do you get if you remove the set from our program?

Solution to Activity 24 (Auxiliary Variables)

1. Program variables appear in the program and possibly also the outline conditions. Logical variables appear only in the conditions, so they never need to be stored in memory. Auxiliary variables are program variables whose values we don't want to store in memory.

2. Let $R_1 = V - A_1$ and $R_2 = V - A_2$. Also, let $A = A_1 \cup A_2$ and $R = V - A$. For A to be auxiliary, we need the variables of R to depend only on variables in R. Since A_1 and A_2 are auxiliary, variables in R_1 depend only on variables in R_1 and similarly for R_2. Since $R = V - A = V - A_1 \cup A_2 = (V - A_1) \cap (V - A_2) = R_1 \cap R_2$, variables in R depends only on variables in R_1 and R_2, so A is auxiliary..

3. (Write-only and read-only variables)
   a. Write-only variables can always be auxiliary: They don't appear in **`if`**/**`while`** tests and they daren't used on the right hand side of assignments to any variables, much less required variables.
   b. Read-only variables (a.k.a. constants) may or may not be auxiliary. They can appear in **`if`**/**`while`** tests and on the r.h.s. of assignments to any kind of variable.

4. (Program `x := y+z; u := v; w := u; if w > 0 then y := y*z fi`)
   a. The **`if`** `w > 0` tells us `w` is primary. The assignments `w := (u)` and `u := (v)` tell us that `u` and `v` can't be auxiliary. So `u`, `v`, `w` is the answer.
   b. {`x`}, {`x`, `y`}, and {`x`, `y`, `z`} are the only auxiliary sets:
      - `x := y+z` tells us that if `y` or `z` are auxiliary, so is `x`. (If one or both of `y` and `z` are marked auxiliary, we have the labeling `x := (y) + z` or `y + (z)` or `(y) + (z)`.)
      - `y := y*z` tells us that if `z` is auxiliary, so is `y`. (We would have the labeling `(y) := (y)*(z)`.)
   c. We start with `x := y+z; u := v; w := u; if w > 0 then w := w-1; y := y*z fi`.
      Removing {`x`} yields `u := v; w := u; if w > 0 then w := w-1; y := y*z fi`
      Removing {`x`, `y`} or {`x`, `y`, `z`} yields `u := v; w := u; if w > 0 then w := w-1 fi`

5. (Program `x := 1; y := 0; while x < n do y := y+1; z := x*y; x := x+x od`)
   a. From **`while`** `x < n`, we know `x` and `n` are primary. The only interesting assignment is `z := x * y` because it tells us that if `y` is auxiliary, `z` must be auxiliary too. (The labeling would be `(z) := x * (y)`.)
   b. The sets of auxiliary variables are {`z`} and {`y`, `z`}.
   c. Removing `z` yields `x := 1; y := 0; while x < n do y := y+1; x := x+x od`
      Removing `y` and `z` yields `x := 1; while x < n do x := x+x od`