Types, Expressions, and States
CS 536: Science of Programming, Fall 2018

8/27: solved

A. Why?

- Expressions represent values in programming languages, relative to a state.
- Types describe common properties of sets of values.
- States describe memory; an expression has a value relative to a state.

B. Outcomes

At the end of this lecture, you should

- Understand what expressions we'll be using in our language.
- Understand what a state is, how we're representing them, and how expressions have values relative to states.

C. Types and Expressions

- Let’s start looking at programming language we’ll be using.
- The datatypes will be pretty simple (no records or function types, for example).
  - Primitive types: int (integers) and bool (boolean). We can add other types like characters, strings, and floating-point numbers, but for what we’re doing, integers and Booleans are enough.
  - Composite types: Multi-dimensional arrays of primitive types of values, with integer indexes.
- Expressions are built from
  - Constants: Integers (0, 1, -1, …) and Boolean constants (T, F).
  - Simple variables of primitive types.
  - Structured variables: We have arrays. (No records or pointers.) For a 1-dimensional array reference, the syntax is the usual b[e] where e is an integer expression. Arrays are zero-origin and fixed-size. (You can look up the size using size(b).) There are some limitations on the use of arrays (see below).
  - On integers: +, -, *, /, min, max, %, =, ≠, <, ≤, >, ≥, divides
  - On booleans: ¬, ∧, ∨, →, ↔, = (note = and ↔ mean the same thing).
  - On arrays: size
- Conditional expressions: B ? e₁ : e₂ as in C or Java etc., where B is a boolean expression and e₁ and e₂ have the same simple type. (Can’t be arrays or functions, e.g.) You can also write a conditional expression as if B then e₁ else e₂ fi.
  - The two expressions e₁ and e₂ must have the same type so that the whole conditional has a consistent type. (Sometimes this is called “balancing”.)
- We don’t have: Assignment expressions, pointers, records, arrays as values. Also, we don’t explicitly declare variables; we’ll assume we know or can infer their types. (E.g., x must be an integer in x+2.) The default datatype for a variable is integer.
Array Limitations:

- We only have arrays of primitive types of values (no arrays of functions, for example). Arrays indexes are zero-origin; \texttt{size(b)} gives the length of an array. The size can be zero. At runtime, an illegal index causes a runtime error.

- We don’t have arrays as values, so we can’t assign an array to a variable, and we don’t have expressions that yield arrays. You can pass an array as a function argument or parameter; e.g., \texttt{sort(b)} might be a function that sorts \texttt{b} in place.

- Multi-dimensional arrays are allowed, but you can’t take a slice of an array. E.g., if \texttt{b} has 2 dimensions, you can use \texttt{b[1][3]} as an expression but not just \texttt{b[1]} (since it would yield a 1-dimensional array). To get the length along each dimension, we can use \texttt{size1(b), size2(b), etc.}.

- You can use an array in two contexts: \texttt{b[some index]} and as a function argument or parameter (including a predicate function). We don’t have array variables or array assignments or expressions of type array.

**Example 1:** \[(x < 0 \ ? \ x+y : x*y)+z\] means “If \(x < 0\) evaluates to true, then we evaluate \(x+y\) and add the result to \(z\), otherwise evaluate \(x*y\) and add the result to \(z\).” (Types: \(x, y, \) and \(z\) must all be integers.)

**Example 2:** \[(i < 0 \ ? \ 0 : a[i])\] yields 0 if \(i\) is negative, otherwise it yields \(a[i]\). (Types: \(i\) is an integer and \(a\) is an array of integers.) Note we avoid \(i < 0\) but not \(i \geq \text{array size}\).

**Example 3:** \[(i < 0 \ ? \ b[0] : i \geq \text{size(b)} \ ? \ b[	ext{size(b)}-1] : b[i])\] yields \(b[i]\) if \(i\) is in range; if \(i\) is negative, it yields \(b[0]\); if \(i\) is too large, it yields the last element of \(b\).

**Example 4:** \(b[i < 0 \ ? \ 0 : i \geq \text{size(b)} \ ? \text{size(b)}-1 : i]\) yields the same value as Example 3, but it does this by calculating the index first.

**Example 5:** A (conditional) expression can’t yield a function, so

- **Example 5a:** \((x > 1 \ ? \ \min(t, u) : \max(t, u))\) is legal

- **Example 5b:** \((x > 1 \ ? \ \min : \max)(t, u)\) is illegal

**Example 6:** We can’t have array-valued expressions, so

- **Example 6a:** \((x \ ? \ a[0] : b[0])\) is legal assuming \(a\) and \(b\) are one-dimensional arrays.

- **Example 6b:** \((x \ ? \ a : b)[0]\) is illegal

**Notation:** \(c\) and \(d\) are constants; \(e\) and \(s\) are general expressions; \(B\) and \(C\) are boolean expressions, \(a\) and \(b\) are array names, and \(u, v, \) etc. are variables. Greek letters like \(\alpha\) and \(\beta\) stand for semantic values.

**Syntactic Values and Semantic Values**

- In this course, we’ll sometimes be dealing with syntactic objects like expressions, programs, and predicates; other times we’ll deal with their meanings. Often, the context of a discussion tells us what kind of object we’re talking about. (E.g., \(x\) and \(p\) have to be syntactic items in “Does \(x\) occur in the predicate \(p\)?”)

- Unfortunately, some symbols are used as syntactic items and also to indicate semantic values: Does “2+2 = 4” indicate an expression within a program? Or is it a statement about how mathematical values work?

- I’ll try to use \texttt{this fixed-width font} when writing out basic syntactic items like expressions and programs. E.g., \texttt{sqrt(2)} or “2+2 = 4”.

---

© James Sasaki, 2018
The everyday font is for semantic values. Sometimes I’ll italicize them or write them out in words to really emphasize that they’re semantic values. So “two plus two equals four” must be a statement about semantics. Often, I’ll use greek letters to stand for semantic values.

On your homeworks and exams, you don’t have to make this distinction. We’ll have to rely on the context to say whether something is syntactic or semantic. I won’t make this distinction on the blackboard, either.

D. States of Memory

Expressions and predicates are syntactic objects; their meanings are semantic values (like the integers etc).

The value of an expression or the truth of a predicate can depend on the values of the variables.

A (memory) state (typically $\sigma$, $\tau$) specifies values of variables.

Technically, a state is a function from variables (considered as pieces of text) to semantic values. A function is a set of ordered pairs of the form $(x, val)$. This ordered pair is a binding of the variable $x$ to the value $val$. If $\sigma$ binds $x$ to the value $val$ (i.e., $\sigma$ includes $(x, val)$), then $\sigma(x) = val$.

Only one binding can appear for any variable, so we can’t have $\{(x, 1), (x, 3)\}$ as a state.

A function is a legal state if it binds variables to values and each variable is bound only once.

On the blackboard, we typically draw memory states with the identifiers next to boxes that contain values. E.g., let $\sigma$ be the following state:

![Memory State Diagram]

Using the set of ordered pairs notation, we write $\sigma = \{(x, 1), (y, F)\}$ for this memory state.

**Notation:** Within a memory state, we can use $x = val$ as an abbreviation for the binding $(x, val)$. So we can abbreviate $\sigma = \{(x, 1), (y, F)\}$ to $\sigma = \{x = 1, y = F\}$.

**Note:** Since $\sigma(x)$ is a semantic value, it isn’t right to write things like $\sigma(x) = y+z$ because $y+z$ is a syntactic value, not a semantic one. Since an expression only has a value relative to a state, you can’t just say “the value of $y+z$” unless you mention a state: “the value of $y+z$ in state $\sigma$” for example. So writing something like $\sigma(x) = \sigma(y+z)$ does make sense. Similarly, if $\alpha$ and $\beta$ stand for the values of $x$ and $y$ in some state (i.e., $\sigma(y) = \alpha$ and $\sigma(z) = \beta$), then $\sigma(x) = \alpha + \beta$ makes sense. (Note we could have written “$\sigma(x) = \alpha$ plus $\beta$” — that tells you that the plus operation is a semantic value (the mathematical function of addition).

Arrays are more complicated: The value of an array is a function from indexes to values. E.g., in the state below, the array $b$ is mapped to a 3-index function:

![Array Diagram]
So in state \( \tau \), the variable \( b \) is mapped to a function \( \{ (0, 3), (1, 5), (2, 9) \} \), so we can write
\[
\tau = \{ b = \{ (0, 3), (1, 5), (2, 9) \}, x = 5 \}.
\]

Often, life is easier if we give names to intermediate values, so we might write
\[
\tau = \{ b = \beta, x = 5 \}
\]

where function \( \beta = \{ (0, 3), (1, 5), (2, 9) \} \)

Since \( \beta \) is a mathematical function, we could also write
\[
\tau = \{ b = \beta, x = 5 \}
\]

where \( \beta(0) = 3, \beta(1) = 5, \text{ and } \beta(2) = 9 \)

Most of the time, it’ll be up to you to decide which one of these to use.

- **Notation**: Let’s abbreviate the function value from \( \{ (0, 3), (1, 5), (2, 9) \} \) to \( (3, 5, 9) \), so we can write
\[
\tau = \{ b = (3, 5, 9), x = 5 \}.
\]

In general, as the value of a variable, \( (\alpha_0, \alpha_1, \ldots, \alpha_n) \) will stand for the function \( \{ (0, \alpha_0), (1, \alpha_1), \ldots, (n, \alpha_n) \} \). Note that the binding \( b = (2) \) indicates that \( b[0] = 2 \), but just \( b = 2 \) indicates that (the value of) \( b \) is an integer.

- **Notation**: For flexibility, let’s also allow ourselves to write bindings for \( b[\text{constant}] \), so we can write \( \tau = \{ b[0] = 3, b[1] = 5, b[2] = 9, x = 5 \} \).

- **Side note**: Don’t worry about multi-dimensional arrays.

- **Definition**: A state is **well-formed** or **proper for an expression** if it binds type-correct values to all the variables of the expression. Similarly, we’ll say a state is **proper for a predicate** if it binds type-correct values to all the free variables of the predicate.

  - E.g., the state \( \{ x = 0, y = 1 \} \) is proper for the expression \( x + y \) (assuming \( x \) and \( y \) are integer variables)
  
  - But the state \( \{ x = (2, 4) \} \) is improper for \( x + y \) for two reasons: It gives \( x \) an array value, and it doesn’t define a value for \( y \).

  - It doesn’t matter if a predicate doesn’t actually need the value of a variable; the state must still define a type-correct value. E.g., \( \{ x = 0, y = 1 \} \) is proper for \( x = (T \ ? \ 17 : y) \) but just \( \{ x = 0 \} \) is not.

  - In general, when we talk about some collection of states, we **won’t include the improper ones**. E.g., if we say “Let \( \Sigma_0 \) be the set of states in which \( x = y+1 \) is false”, we don’t include improper states like \( \{ x = 5, y = \text{“abc”} \} \).

**Common Mistakes Writing out States**

- A state is a set of bindings, with each binding connecting a program variable (something that could be part of a program) to a semantic value. Our semantic values will typically be integers and functions from indexes to integers, but it’s easy to add other primitive types of values, like characters.

- The main mistake people make is forgetting the difference between textual and semantic items.

- In \( \{ x = \text{expression} \} \), the expression has to stand for a semantic value, so things like “2”, “two”, “3+5”, are fine. But more complicated expressions can occur: “\( \text{truncate}(f(3)) \)” where \( \text{truncate} \) and \( f \) are functions on values. It can be helpful to define a semantic variable (a name for a semantic object). I’m typically using a Greek letter for these variables; for example, “\( \{ x = \alpha \} \) where \( \alpha = \text{truncate}(f(3)) \)”. But they really can be any symbol: “\( \{ x = a \} \) where \( a = \text{truncate}(f(3)) \)”.

- So one mistake is to use something that’s obviously a textual value where we need a semantic one. For example, “\( \{ x = \text{“string”} \} \)” (unless we’ve added strings to our repertoire of values).
More common is to use something inconsistently: as a syntactic value in one place and a semantic value in another.

E.g., “\( \{ x = 5, \ y = x - 3 \} \)” is illegal: To be legal, “\( x - 3 \)” has to have a value, so “\( x \)” has to have a value (it has to be the name of a mathematical object like 5). But the binding \( x = 5 \) tells us “\( x \)” is a variable, so “\( x \)” is a syntactic object. It can’t be syntactic and semantic at the same time. We could say \( \sigma = \{ x = 5, \ y = \sigma(x) \} \) minus three).

In this nicely word-processed document, “\( x \)” is presented in this font, so we know it’s supposed to be a syntactic object. So if “\( \{ y = x+3 \} \)” appears in the notes, it’s a bug, because \( x \) is supposed to be a textual item. But “\( \{ y = x+3 \} \)” is okay so long as we consistently use \( x \) as the name of a semantic value.

The problem is exacerbated because on the blackboard and in your homeworks, you don’t have to differentiate between “\( x \)” and “\( x \)”. Even so, if someone wrote \( \{ x = 5, \ y = x - 3 \} \) on the blackboard, it would have to be illegal because of using \( X \) in two incompatible ways.

E. Values of Expressions

In general, expressions have values relative to a state. E.g., relative to \( \{ x = 1, \ b = (2, 0, 4) \} \), the expression \( b[b[|x|]] \) has the value 2.

**Notation:** \( \sigma(e) \) is the value of expression \( e \) in state \( \sigma \).

- We’re extending the notion of \( \sigma(v) \) where \( v \) stands for a variable.
- Note \( \sigma(e) \) is semantic value, not a syntactic value.

We calculate \( \sigma(e) \) by following the structure of \( e \): Recursively evaluate subexpressions and combine their values using operators or functions. The base cases are variables (we get their values from the state) and constants (which have values independent of any state).

The value of \( \sigma(e) \) depends on what kind of expression \( e \) is:

- \( \sigma(x) = \) the value that \( \sigma \) binds variable \( x \) to
- \( \sigma(c) = \) the value of the constant \( c \)
- \( \sigma(e_1 + e_2) = \sigma(e_1) + \sigma(e_2) \) [and similar for -, *, etc.]
- \( \sigma(e_1 < e_2) = \text{T} \) if \( \sigma(e_1) \) is less than \( \sigma(e_2) \) [similar for \( \leq, =, \text{etc}. \)]
- \( \sigma(e_1 \& e_2) = \text{T} \) if \( \sigma(e_1) \) and \( \sigma(e_2) \) are both \( \text{T} \) [similar for \( \lor, \text{etc}. \)]
- \( \sigma(B \ ? e_1 : e_2) = \sigma(e_1) \) if \( \sigma(B) = \text{T} \)
- \( \sigma(B \ ? e_1 : e_2) = \sigma(e_2) \) if \( \sigma(B) = \text{F} \)
- \( \sigma(b[e_1]) = \gamma(\beta) \) where \( \gamma = \sigma(b) \) and \( \beta = \sigma(e_1) \) [8/27 rewrote a bit]

Here, \( b \) is an array name, so it stands for a function from index values to array values; \( e_1 \) is the index expression, so we have to evaluate to find the element of \( b \) we want.

You can also write \( \sigma(b[e_1]) = \sigma(b)(\sigma(e_1)) \) if you don’t want to define \( \gamma \) and \( \beta \). Function application is left-associative, so \( \sigma(b)(\sigma(e_1)) = (\sigma(b))(\sigma(e_1)) \). I.e., \( \sigma(b) \) is a function we’re applying to \( \sigma(e_1) \).

* Using a felt marker, presumably :-)
Here, \( b \) is an array name, so it stands for a function from index values to array values; \( e_1 \) is the index expression, so we have to evaluate to find the element of \( b \) we want. So \( \sigma(b) \) is a function and \( \sigma(e_1) \) is the value we apply it to.

- **Example 7:** Let \( \sigma = \{ x = 1, b = \alpha \} \) where \( \alpha = (2, 0, 4) \). Then
  
  - \( \sigma(x) = 1 \)
  - \( \sigma(x+1) = \sigma(x) + \sigma(1) = 1 + 1 = 2 \) \[8/27\]
  - \( \sigma(b) = \alpha \)
  - \( \sigma(b[x+1]) = (\sigma(b))(\sigma(x+1)) = \alpha(2) = 4 \)
  
  If we don't want to write out the intermediate steps first, we could write
  
  - \( \sigma(b[x+1]) = (\sigma(b))(\sigma(x+1)) = \alpha(\sigma(x)+1) = \alpha(1+1) = \alpha(2) = 4. \)

- **Example 8:** Using the same \( \sigma \) as in Example 7, let \( \tau = \sigma \cup \{ y = 1 \} \) (\( \sigma \) as in example 7) and \( e \equiv (x = (y > 0 \ ? 17 : y)) \), then \( \tau(e) = F \):
  
  - \( \tau(e) = T \) iff \( \tau(x) = \tau(y > 0 \ ? 17 : y) \)
  - For the left side, \( \tau(x) = \sigma(x) = 1 \)
  - For the right side, \( \tau(y > 0) = T \) iff \( \tau(y) > \tau(0) \) iff one is greater than zero, which is true
  - So \( \tau(y > 0 \ ? 17 : y) = \tau(17) = \text{seventeen because } \tau(y > 0) = T \)
  - So \( \tau(e) = T \) iff one equals seventeen, which is false
  - So \( \tau(e) = F \)

- **Example 9:** Let \( \sigma = \{ x = 1, b = \alpha \} \) where \( \alpha = (2, 0, 4) \), then
  
  - \( \sigma(b[x+1]–2) = \sigma(b[x+1]) – \sigma(2) = (\sigma(b))(\sigma(x+1))–2 \)
    
    \[
    = (\sigma(b))(\sigma(x)+1)–2 \\
    = \alpha(1+1)–2 \\
    = \alpha(2)–2 = 4–2 = 2.
    \]

- **The empty state:** Since a state is a set of bindings, the empty set \( \emptyset \) is a state (the empty state). It’s proper for any expression or predicate that doesn’t include variables. E.g., In state \( \emptyset \), the expression \( 2+2 \) evaluates to four. (In fact, since we don’t care about bindings for variables that don’t appear in an expression, we can say that in any state \( \sigma \), \( 2+2 \) evaluates to 4.

- **Example 10:** Let \( \sigma = \emptyset \) (the empty state) then
  
  - \( \sigma(2+2 = 4) = \sigma(2+2) \) equals \( \sigma(4) = … = \text{four equals four} = T. \)

- **The value of an expression has to be a semantic value.** So \( \sigma(v+w) = \sigma(v)+\sigma(w) \) is okay.
  
  - We infer that \( v \) and \( w \) are syntactic variables, the “+” on the left side of the “=“ is the addition operator in our language; the “+” on the right side stands for for the mathematical function of addition. \( \sigma(v) \) and \( \sigma(w) \) are semantic values because the meaning of a variable is a value, not an expression.
  
  - But \( \sigma(v+w) = v \) plus \( w \) is an error; we can’t use \( v \) and \( w \) as variables that appear in the expression \( x+y \) while also using them as semantic variables whose values we’re summing.
Types, Expressions, and States
CS 536: Science of Programming, Fall 2018

A. Why
- Expressions represent values in programming languages, relative to a state.
- Types describe common properties of sets of values.
- States describe memory; an expression has a value relative to a state.

B. Outcomes
At the end of today, you should
- Be able to recognize expressions we'll be using in our language.
- Be able to recognize a state is and know what it means for a state to be proper.
- Be able to evaluate an expression relative to a state.

C. Questions
1. Which of the following expressions are legal or illegal according to the syntax we’re using? Assume x, y, z are integer variables and b is an array name.
   a. \((x > y \ ? \ x : y)\) /* do you need assumptions as to the types of x and y */ [8/27]
   b. \((x < y \ ? -1 : (x = y \ ? 0 : 1))\)
   c. \((y = 0 \ ? f : g)(17)\)
   d. \(b[0][1]\) [8/27] /* What type must b have for this to be legal? */
   e. \(b\) [8/27] /* Remember we're given that b is an array */
   f. \(f(b, b[0]) < 3\) /* Also, if this is legal, what is the type of f? */
   g. \((x < 3 \ ? x : F)\)

2. Which of the following are legal ways to write out a state? (And if not, why not?)
   a. \{x = 5, y = 2\}
   b. \{x = five, y = one plus one\}
   c. \{x = 5, y = x minus 3\}
   d. \{x = 5, y = \alpha - 3\} where \(\alpha = 5\)
   e. \{x = 5, y = (the value of x in this environment minus 3)\}
   f. \{\}

3. Consider the state \(\sigma_2\) described graphically below.

   \[
   \begin{array}{c|c|c|c|c}
   \sigma_2 & 0 & 1 & 2 & 3 \\
   \hline
   x & 12 & & & \\
   y & T & & & \\
   b & 2 & 4 & 3 & 8 \\
   \end{array}
   \]
a. Write a definition for $\sigma_2 = \{ \ldots \}$ using the standard set of bindings notation.

b. Calculate $\sigma_2(e)$ where $e \equiv y \land x > b[x/5]$. Assume integer division truncates.

4. Let $\sigma_3 \equiv \{ z = 4, b[0] = 1, b[1] = 5, b[2] = 8 \}$.
   a. Abbreviate this using tuple notation for the value of $b$ (i.e., $b = (\ldots)$).
   b. Write out the graphical representation of $\sigma_3$ (a memory diagram as in Problem 2).
   c. Calculate $\sigma_3(e)$ where $e \equiv b[b[z-4]] > z \ ? \ z+2 \ : \ z-2$. (Hint: Give names to parts of $e$ and calculate the values of those parts first.)

5. Let $e_4 \equiv x = y + 1 \land y = z^2 - 3 \land z = 6$. Write out the textual definition of a state $\sigma_4$ in which $e_4$ evaluates to true. Use only bindings that map variables to constants. $\sigma_4 = \{ x = 34, y = 33, z = 6 \}$

6. Which of the following states are legal and also proper for the expression $b[i] + 0 * y$? ([8/27] But might cause a runtime error.) (If illegal, why?)
   a. $\{i = 0, b = (3, 4, 8), y = 3, z = 5\}$
   b. $\{i = 0, b = (6), y = 5\}$
   c. $\{i = 0, b = 6, y = 5\}$
   d. $\{i = 1, b = (3, 4, 8)\}$
   e. $\{i = 1, i = 2, y = 0, b = (2, 6)\}$
   f. $\{i = 5, b = (1, 2), y = 4\}$
**Solution to Activity 3 (Types, Expressions, and States)**

1. (Legal and illegal expressions)
   a. \((x > y \ ? \ x \ : \ y)\) is legal
   b. \((x < y \ ? \ -1 \ : \ (x = y \ ? \ 0 \ : \ 1))\) is legal
   c. \((\gamma = 0 \ ? \ f \ : \ g)(17)\) is illegal because the conditional expression can't yield a function
   d. \(b[0][1]\) is legal (\(b\) must be a 2-dimensional array)
   e. \(b\) (all by itself) is illegal, since \(b\) we've assumed is an array
   f. \(f(b, b[0]) < 3\) is legal (the name \(b\) is being used as an argument to a function). We infer that \(f\) has type \((\text{int array}) \times \text{int} \rightarrow \text{int}\).
   g. \((x < 3 \ ? \ x \ : \ F)\) is illegal because \(x\) and \(F\) have different types. (I.e., the expression doesn’t have a fixed type because the types of its arms don’t match.)

2. (Legal ways to represent states)
   a. \(\{x = 5, \ y = 2\}\) is legal
   b. \(\{x = \text{five}, \ y = \text{one plus one}\}\) is legal because “five” and “one” etc. refer to semantic objects.
   c. \(\{x = 5, \ y = x \text{ minus 3}\}\) is illegal: To be legal, “\(x\) minus 3” has to be a value, so “\(x\)” has to be a value (it has to be the name of a mathematical object like 5). But the binding \(x = 5\) tells us “\(x\)” is a variable that can appear in an expression, so “\(x\)” is a syntactic object. It can’t be syntactic and semantic at the same time.
   
   Also, in this nicely word-processed document, “\(x\)” is presented in this font, so we know it’s supposed to be a syntactic object. On paper, you see the difference between “\(x\)” and “\(x\)” etc. refer to semantic objects. Even so, if someone wrote \(\{x = 5, \ y = x \text{ minus 1}\}\) on the blackboard, it would have to be illegal because of using \(X\) in two incompatible ways.
   d. \(\{x = 5, \ y = \alpha - 3\}\) where \(\alpha = 5\) — is legal. We infer that symbols \(x\) and \(y\) are syntactic objects and \(\alpha\) is the name of the semantic object 5.
   e. \(\{x = 5, \ y = \text{(the value of } x \text{ in this environment, minus 3)\}\) is legal. Since “the value of \(x\) in this environment” is just another name (albeit complicated) for the mathematical object 5, it’s legal to use here.
   f. \(\{\}\) is legal, since it’s just another way to write \(\emptyset\), the empty state.

3. (Graphically defined state)
   a. \(\sigma_2 = \{x = 12, \ y = \text{T}, \ b = (2, 4, 3, 8)\}\). Also correct is \(\sigma_2 = \{x = 12, \ y = \text{T}, \ b[0] = 2, \ b[1] = 4, \ b[2] = 3, \ b[3] = 8\}\)
   b. You can write out these kinds of calculations to different levels of detail, but a brief answer is that \(\sigma_2(e) = \sigma_2(y \land x > b[x/5]) = \text{T} \land 12 > 3 = \text{T}\) You can certainly show intermediate steps:

‡ Using a felt marker, apparently :-(
\[ \sigma_2(e) = \sigma_2(y) \land \sigma_2(x) > \sigma_2(b)(\sigma_2(x/5)) \]
\[ = T \land 12 > \sigma_2(b)(12/5) \]
\[ = T \land 12 > \sigma_2(b)(2) = T \land 12 > 3 = T \]

4. (Alternative ways to represent a state with an array value)
   a. \( \sigma_3 = \{ z = 4, b = (1, 5, 8) \} \)
   b. 
   
   \[
   \begin{array}{c|ccc}
   & 0 & 1 & 2 \\
   \hline
   z & 4 & & \\
   b & 1 & 5 & 8
   \end{array}
   
   c. We have \( e \equiv b[b[z-4]] > z ? z+2 : z-2 \). To make this easier to deal with, let’s break it down. Let \( e \equiv e_1 > z ? z+2 : z-2 \) where \( e_1 \equiv b[e_2] \) and \( e_2 \equiv b[z-4] \).
   - First, \( \sigma_3(e_2) = \sigma_3(b[z-4]) = (\sigma_3(b))(\sigma_3(z-4)) = (\sigma_3(b))(4-4) = (\sigma_3(b))(0) = 1 \)
   - So \( \sigma_3(e_1) = \sigma_3(b[e_2]) = (\sigma_3(b))(\sigma_3(e_2)) = (\sigma_3(b))(1) = 5 \)
   - Then \( \sigma_3(e_1 > z) = (\sigma_3(e_1) > \sigma_3(z)) = 5 > 4 = F \)
   - So \( \sigma_3(e) = \sigma_3(e_1 > z ? z+2 : z-2) = \sigma_3(z+2) \)
   - So finally, \( \sigma_3(e) = \sigma_3(z+2) = \sigma_3(z) + \sigma_3(2) = 4+2 = 6 \)

5. \( \sigma_4 = \{ z = 6, y = 33, x = 34 \} \)

6. (Proper states)
   a. Proper: The extra binding for \( z \) isn't a problem
   b. Proper: The value of \( b \) is an array of length 0.
   c. Improper: The value of \( b \) can't be an integer.
   d. Improper: We need a binding for \( y \) even though we're multiplying it by zero.
      [So our semantics uses eager evaluation, not lazy evaluation.]
   e. Improper: We have two bindings for \( i \).
   f. Proper but causes a runtime error, since \( b \) has size 2. [8/27]