

Class 03: Expression Values; State Updates; Simple Programs

A. Why?

To understand how our programs work, we must understand how expressions are evaluated to values (given a memory state), how memory states can be changed, and the basic syntax and semantics of our simple deterministic programming language.

B. Outcomes

By the end of the class you should

- Understand what the textbook means by “the value of an expression (given some memory state.”
- Be able to update a state and read/write the textbook’s notation for state updates.
- Know the basic syntax and intuitive semantics of our simple deterministic programming language.

C. Semantics of Expressions (page 30)

- Textbook has two notations for “the value/meaning/semantics of expression s in memory state σ .”
 - They’ll usually use $\sigma(s)$ [extension of *state(variable)* notation]
 - They start out using $\mathcal{S} \llbracket s \rrbracket (\sigma)$. (The \mathcal{S} is for “semantics.”)
 - The hollow brackets are hints that the argument is a syntactic object.
 - The notation is for precise communication.
 - ***It’s important to practice translating from notation to English!***
 - Other direction good but less critical.
- Definition of “the value of expression s in σ ” uses syntactic structure of s .
- Base cases: s has no subexpressions
 - s is a constant or variable
- Inductive cases: s does have subexpressions
 - I.e., *unaryOp* s_1 and $a[s_1]$ [have 1 subexpression each], s_1 *binaryOp* s_2 [has 2 subexpressions], **if** B **then** s_1 **else** s_2 **fi** [has 3 subexpressions].

Cases of definition of $\sigma(s)$

1. If s is an integer or boolean constant c , then its meaning is itself.
 - $\sigma(c) = I(c) = c$.
 - $I(c)$ means “The standard interpretation of the constant c ”
2. If s is a simple variable x , its value is specified by the state.
 - $\sigma(x)$ [as a function on expressions] = $\sigma(x)$ [as a function on variables]

3. If s is $s_1 \text{ op } s_2$, then its meaning is the result of running the function denoted by the operator on the values of s_1 and s_2 .
 - $\sigma(s_1 \text{ op } s_2) = \alpha(\sigma(s_1), \sigma(s_2))$ where α is the function $I(\text{op})$.
 - $I(+)$ is addition on integers, $I(=)$ is equality on integers (or booleans),
4. (Unary operations like $-x$ are similar.)
5. If s is an array indexing expression $a[s_1, \dots, s_n]$ then its meaning is the result of running the function denoted by the array on the values of the indexes.
 - $\sigma(a[s_1, \dots, s_n]) = \alpha(\beta_1, \dots, \beta_n)$ where $\alpha = \sigma(a)$ and each $\beta_i = \sigma(s_i)$.
 - Equivalent: $\sigma(a)(\sigma(s_1), \dots, \sigma(s_n))$.
6. If s is a conditional expression **if** B **then** s_1 **else** s_2 **fi**, then its value is either the value of s_1 or the value of s_2 depending on whether the value of B is true or false.
 - If $s \equiv \text{if } B \text{ then } s_1 \text{ else } s_2 \text{ fi}$, then $\sigma(s) = \sigma(s_1)$ if $\sigma(B) = \mathbf{true}$ and $\sigma(s) = \sigma(s_2)$ if $\sigma(B) = \mathbf{false}$.
 - Examples of the meaning/value of an expression:
 - Let σ_0 be the state that maps m to 5, $a[0]$ to 6, $a[1]$ to 10, and $a[2]$ to 8.
 - Let $s_0 \equiv m \geq 0 \wedge m < 3$, then $\sigma_0(m) = \mathbf{true}$
 - Let $s_1 \equiv \text{if } s_0 \text{ then } m \text{ else } 0 \text{ fi}$, then $\sigma_0(s_1) = \sigma_0(m)$ [because $\sigma_0(s_0) = \mathbf{true}$], and $\sigma_0(m) = 5$.
 - Let $s_2 \equiv a[s_1 - 3]$, then $\sigma_0(s_2) = \alpha(5-3) = \alpha(2) = 8$.
 - Let $s_3 \equiv a[s_2/6]$, then $\sigma_0(s_3) = \alpha(8/6) = \alpha(1) = 10$. [Assume division truncates.]

D. State Updates (page 32)

- Our programming language will be very simple: assignment statements, conditionals, and while loops.
- The meaning of a statement or program is that it takes an initial state of memory and produces a final state of memory.
 - Example: if $\sigma(x) = 17$, then the assignment statement that increments x by 1 takes σ and produces a new state (let's call it τ) where $\tau(x) = 18$.
 - τ is σ updated so that at x , it yields 18.
 - Textbook writes this $\tau = \sigma[x := 18]$.
- General notation is $state[var := value]$
 - Can chain these: If in state τ we assign 0 to y , we get $\tau[y := 0]$, which equals $\sigma[x := 18][y := 0]$. (In English: The state σ updated to map x to 18 and then updated to map y to 0.)
 - Let $\tau' = \tau[y := 0]$, then $\tau'(y) = 0$ regardless of whether $\sigma(y)$ was defined or not: $\tau'(y) = (\tau[y := 0])(y) = 0$.

- In general, $(state[var_1 := val])(var_1) = val$; if var_1 and var_2 aren't \equiv , then $(state[var_1 := val])(var_2) = state(var_2)$. E.g. $\tau'(x) = (\tau[y := 0])(x) = \tau(x) = (\sigma[x := 18])(x) = 18$.
- In English: The value of x in state τ' = the value of x in the state τ updated to map y to 0 = the value of x in state τ = the value of x in σ updated at x to 18 = 18.
- Note: There's also a notion of updating an array variable; we'll worry about it some other day.
- In $state[var := val]$, the value can be complicated.
- Example: if $\sigma(x) = 5$, then $\sigma[x := \sigma(x+1)] = \sigma[x := 6]$.
 - In English: If σ maps x to 5, the the update of σ at x with the value of $x+1$ (in state σ) is the update of σ at x with 6.
- This is essentially what the assignment x gets $x+1$ does: It changes the current state at x with the value of $x+1$ (in the current state).
- It's tempting to write $\sigma[x := x+1]$, but technically, the r.h.s. (right hand side) of the $:=$ is supposed to be a value, not an expression.
- Note on notation: In the textbook, states are written as $state_var[var_1 := value_1] \dots [var_n := value_n]$ where $state_var$ is σ or τ (possibly decorated with primes or subscripts), $n \geq 0$ is some number of iterated updates, each var_i is a program variable (something that could appear in a program), and each $value_i$ is a a value that can be stored in memory. A *value* is written as a constant (0, 1, ..., **true**, **false**, maybe 'a', 'b', ...) or as $state(expr)$, the value of an expression.
- Complicated example: $\sigma[x := 1][y := 2][z := \sigma[x := 1][y := 2](x+y)]$.
- To avoid repetition, it's often easier to define intermediate items: $\tau[z := \tau(x+y)]$ where $\tau = \sigma[x := 1][y := 2]$.

A. Simple Programs

- Our programming language has five kinds of statements and no declarations
- **skip** — does nothing
- Assignment: $var := expr$
 - We'll always work with type-correct programs, and we'll assume we know the types of variables without bothering to write declarations. E.g., in $x := 17$, x must be an integer variable.
- Sequence: $S_1 ; S_2$ where S_1 and S_2 are statements. Semicolon is a separator.
 - Longer sequences are read left-to-right: $S_1 ; S_2 ; S_3$, for example.
- Conditional
 - **if** B **then** S_1 **else** S_2 **fi** where B is a boolean expression and S_1 and S_2 are statements. (Can be nested.)

- **if** B **then** S_1 **fi** is an abbreviation for **if** B **then** S_1 **else skip fi**.
- Loop
 - **while** B **do** S_1 **od** where B is a boolean expression and S_1 is a statement
- Example:
 - **if** $n < 0$ **then** $y := 1$ **else** $x := 0; y := 1$; **while** $x \leq n$ **do** $x := x+1; y := y+y$ **od fi**
 - This statement contains 9 sub-statements (10 if you count itself):
 1. $y := 1$
 2. $x := 0$
 3. $y := 1$
 4. $x := x+1$
 5. $y := y+y$
 6. $x := x+1; y := y+y$
 7. **while** $x \leq n$ **do** $x := x+1; y := y+y$ **od**
 8. $y := 1$; **while** $x \leq n$ **do** $x := x+1; y := y+y$ **od**
 9. $x := 0; y := 1$; **while** $x \leq n$ **do** $x := x+1; y := y+y$ **od**
 - (By the way, what does this program do?)
- Converting between a typical language like C or C++ or Java and our programming language is pretty straightforward; the only problem comes up with statements with side effects (our language doesn't have them).
 - **while** ($--x \geq 0$) $z^*=2$; /* In C++ */
 - $x := x-1$; **while** $x \geq 0$ **do** $z := z^*2; x := x-1$ **od** /* Our equivalent */
 - (The decrement of x before the loop is there because we might skip the loop.)
 - **while** ($x++ \geq 0$) $z^*=2$; /* In C++ */
 - **while** $x > 0$ **do** $x := x+1; z := z^*2$ **od**; $x := x+1$ /* Our equivalent */
 - (The increment of x after the loop is there to handle the post-decrement during the test that stops the loop.)

B. Programs are State Transformers

- We evaluate programs in order to manipulate the memory state.
 - **skip** does the null transformation
 - $var := expr$ does a state update; if σ is the current state, the assignment changes the state to $\sigma[var := \sigma(expr)]$.
 - Sequences do a series of transformations, one after another.
 - Conditional and loop statements don't alter state themselves, they alter control.
- Examples:
 - If we execute $x := 0$ in state σ , the result is $\sigma[x := 0]$. If we execute $y := x+1$ in state $\sigma[x := 0]$, the result is $\sigma[x := 0][y := 1]$, since the value $1 = \sigma[x := 0](x+1)$. We get the same result if we execute $x := 0; y := x+1$ starting in state σ .