Denotational Semantics; Runtime Errors; Nondeterminism pt 1

CS 536: Science of Programming, Fall 2020

A. Why
• Our simple programming language is a model for the kind of constructs seen in actual languages.
• Execution of an entire programs can be viewed as a state transformers.
• Infinite loops and runtime errors cause failure of normal program execution.
• Nondeterminism can help us avoid unnecessary determinism.
• Nondeterminism can help us develop programs without worrying about overlapping cases.

B. Outcomes
At the end of today, you should know how to
• Use denotational semantics to describe overall execution of programs in our language
• Determine that evaluation of an expression or program fails due to a runtime error.
• Make sequential nondeterministic choices

C. Denotational Semantics Definition and Rules
• In addition to the small step-by-step operational semantics for our programs, we'll also introduce a version of semantics that concentrates only on the beginning and end of the evaluation process (hence he name "large-step" semantics).

  **Definition:** The *denotational semantics* of $S$ in $\sigma$ is $\tau$ if in state $\sigma$, program $S$ terminates in $\tau$. (I.e., $\langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle$.) Symbolically, we write $M(S, \sigma) = \{\tau\}$.
  - The reason we have a singleton set containing $\tau$ instead of just $\tau$ is that later, we'll look at non-deterministic computations, which can have more than one possible final state.

  **Notation:** If you slip up and write $M(S, \sigma) = \tau$ instead of $\{\tau\}$, it’s not a big deal.

  **Example 1:** Let $\sigma$ be a state and let $S = x := 1; y := 2$. Since $\langle x := 1; y := 2, \sigma \rangle \rightarrow \langle y := 2, \sigma[x \mapsto 1] \rangle \rightarrow \langle E, \sigma[x \mapsto 1][y \mapsto 2] \rangle$, we know $M(S, \sigma) = \{\sigma[x \mapsto 1][y \mapsto 2]\}$.

  **Notation:** In the literature, some people write hollow square brackets around arguments that are syntactic to emphasize that they are indeed syntactic. Other notations for $M(S, \sigma)$ include $M[S](\sigma)$ and $M[S] \sigma$ and $M(S)(\sigma)$. In the last two cases, $M[S]$ and $M(S)$ are viewed as functions that transform memory states, so $M[S](\sigma) = \tau$ means function $M[S]$ maps $\sigma$ to $\tau$. Our notation $\sigma(e)$ would be written $\sigma[e]$.
**Denotational Semantics Rules**

- Since $M(S, \sigma) = \tau$ means $\langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle$, we can give specific rules for $M(S, \sigma)$ depending on the kind of $S$.
- **Skip and Assignment:** These statements complete in only one step, so the operational semantics rules give the denotational semantics immediately.
  - $M(\text{skip}, \sigma) = \{\sigma\}$
  - $M(v := e, \sigma) = \{\sigma[v \mapsto \sigma(e)]\}$
  - $M(b[e_1] := e, \sigma) = \{\sigma[b[\alpha] \mapsto \beta]\}$ where $\alpha = \sigma(e_1)$ and $\beta = \sigma(e)$.
- **Composition:** $M(S_1; S_2, \sigma) = M(S_2, \tau)$ where $\{\tau\} = M(S_1, \sigma)$. To justify this, say we have $\langle S_1; S_2, \sigma \rangle \rightarrow^* \langle S_2, \tau \rangle \rightarrow^* \langle E, \tau' \rangle$. Since $M(S_1, \sigma) = \{\tau\}$, we run $S_2$ starting in state $\tau$, so $M(S_1; S_2, \sigma) = M(S_2, \tau) = M(S_2, M(S_1, \sigma))$.
- **Notation:** We’ll bend the notation a bit and write $M(S_2, M(S_1, \sigma))$ to mean $M(S_2, \tau)$ where $\{\tau\} = M(S_1, \sigma)$.
  - Note the subscripts in $S_1; S_2$ are 1 then 2 but the subscripts in $M(S_2, M(S_1, \sigma))$ are 2 then 1.
- **Conditional:** The meaning of an if-else statement is either the meaning of the true branch or the meaning of the false branch.
  - If $\sigma(B) = T$, then $M(\text{if } B \text{ then } S_1 \text{ else } S_2, \sigma) = M(S_1, \sigma)$
  - If $\sigma(B) = F$, then $M(\text{if } B \text{ then } S_1 \text{ else } S_2, \sigma) = M(S_2, \sigma)$
- **Example 2:** Let $S = \text{if } y \text{ then } x := x+1 \text{ else } z := x+2 \text{ fi}$, then
  - If $\sigma(y) = T$, then $M(S, \sigma) = \{\sigma[x \mapsto \sigma(x)+1]\}$
  - If $\sigma(y) = F$, then $M(S, \sigma) = \{\sigma[z \mapsto \sigma(x)+2]\}$
- **Iterative:** One way to definite the meaning of $W = \text{while } B \text{ do } S \text{ od}$ is recursively:
  - If $\sigma(B) = F$ then $M(W, \sigma) = \{\sigma\}$
  - If $\sigma(B) = T$ then $M(W, \sigma) = M(S; W, \sigma) = M(W, M(S, \sigma))$.
  - Unfortunately, this definition is not well-formed if $W$ leads to an infinite loop.
- Another way to characterize $M(W, \sigma)$ involves looking at the series of states in which we evaluate the test.
  - Let $\sigma_0 = \sigma$, and for for $k = 0, 1, \ldots$, let $\{\sigma_{k+1}\} = M(S, \sigma_k)$. Then $\sigma_0, \sigma_1, \sigma_2, \ldots$ is the sequence of states seen at successive while loop tests: $\sigma_k$ is the state in effect the $k$'th time we evaluate the loop test.
  - Then $M(W, \sigma)$ is the (set containing the) first state in this sequence that satisfies $\neg B$, assuming there is such a state. (If there isn't, we have an infinite loop.)
- **Example 3:** Let $W = \text{while } x < n \text{ do } S \text{ od}$, where the loop body $S = x := x+1; y := y+x$. The general case for the behavior of $S$ is (for any $\tau$), $M(S, \tau[x \mapsto \alpha][y \mapsto \beta]) = \{\tau[x \mapsto \alpha+1][y \mapsto 2 \beta]\}$. Say we start execution of $W$ in state $\sigma = \{x = 0, n = 3, y = 1\}$. Our sequence of states is
  - $\sigma_0 = \sigma = \{x = 0, n = 3, y = 1\}$
  - $M(S, \sigma_0) = \{\sigma_1\}$ where $\sigma_1 = \{x = 1, n = 3, y = 2\}$
• $M(S, \sigma_1) = \{\sigma_2\}$ where $\sigma_2 = \{x = 2, n = 3, y = 4\}$, and
• $M(S, \sigma_2) = \{\sigma_3\}$ where $\sigma_3 = \{x = 3, n = 3, y = 8\}$.
• Of this sequence, $\sigma_3$ is the first state that satisfies $x \geq n$, so $M(W, \sigma) = \{\sigma_3\} = \{(x = 3, n = 3, y = 8)\}$.

D. Convergence and Divergence of Loops

• Not all loops terminate. Evaluation of an infinite loop yields an unending path of $\rightarrow$ steps: Either an infinite sequence of different configurations or a finite-length cycle of configurations. More generally in computer science we can also also have infinite recursion, which we won't study in detail but is treated similarly to infinite iteration.

• **Definition:** Execution of $S$ starting in $\sigma$ **diverges** if it doesn't converge; i.e., $\langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle$ for no $\tau$.

• **Notation:** $M(S, \sigma) = \{\bot_d\}$ ("bottom sub-$d$") means $S$ diverges in $\sigma$. Note that although we're writing it in a place where you'd expect a memory state, $\bot_d$ is not an actual memory state; we'll call it a *pseudo-state* as opposed to an actual or real memory state like $\sigma$ and $\tau$.
  
  • **Note:** Divergence is one way in which a program doesn't successfully terminate. We'll introduce other flavors of $\bot$ as we look at other ways to not get successful termination.

• **Notation:** $\langle S, \sigma \rangle \rightarrow^* \langle E, \bot_d \rangle$ means that $S$ starting in $\sigma$ diverges. Again, we're not using $\bot_d$ as an actual memory state here, but since $M(S, \sigma) = \{$$\}$ means $\langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle$, if we're going to write $M(S, \sigma) = \{\bot_d\}$ to say that $S$ diverges, it's consistent to write $\langle S, \sigma \rangle \rightarrow^* \langle E, \bot_d \rangle$.

• To determine when $M(W, \sigma) = \{\bot_d\}$, recall that in the previous section we looked at the series of states $\sigma_0, \sigma_1, \sigma_2, \ldots$ in which we evaluate the loop test. For this sequence, $\sigma_0 = \sigma$, and $\sigma_{k+1} = M(S, \sigma_k)$ for $k \geq 0$. For terminating loops, $M(W, \sigma)$ is the first state in the sequence that satisfies $\neg B$. We can now write $M(W, \sigma) = \{\bot_d\}$ to indicate that no state in the sequence satisfies $\neg B$.

• **Example 4:** Let $W = \text{while } T \text{ do skip od}$ and $\sigma$ be any state. Then $\langle W, \sigma \rangle \rightarrow \langle \text{skip } ; W, \sigma \rangle$ but $\langle \text{skip } ; W, \sigma \rangle \rightarrow \langle W, \sigma \rangle$. (As a directed graph, this is a two-node cycle, $\langle W, \sigma \rangle \Rightarrow \langle \text{skip } ; W, \sigma \rangle$.) Hence $M(W, \sigma) = \{\bot_d\}$.

• **Example 5:** Let $W = \text{while } x \neq n \text{ do } x := x - 1 \text{ od}$ and let $\sigma = \{x = -1, n = 0\}$.
  
  • Let $\sigma_0 = \sigma = \{x = -1, n = 0\}$
  • Let $\{\sigma_1\} = M(x := x - 1, \sigma_0) = \{\sigma_0[x \leftarrow -2]\} = \{x = -2, n = 0\}$
  • Let $\{\sigma_2\} = M(x := x - 1, \sigma_1) = \{\sigma_1[x \leftarrow -3]\} = \{x = -3, n = 0\}$
  • In general, let $\{\sigma_k\} = M(x := x - 1, \sigma_{k-1}) = \{x = -k - 1, n = 0\}$
  • Since every $\sigma_k = x \neq n$, we have $M(W, \sigma) = \{\bot_d\}$.

E. Expressions With Runtime Errors

• Using $\bot_d$ lets us talk about a program not successfully terminating because it simply doesn't terminate at all.
• Runtime errors cause a program to terminate, but unsuccessfully. E.g., in σ, the assignment \( z := x/y \) fails if \( \sigma(y) = 0 \) because evaluation of \( \sigma(x/y) \) fails. There are two notions of failure here: The expression fails, and this causes the statement to fail.

• **Definition:** \( \sigma(e) = \bot_e \) means evaluation of \( e \) in state \( \sigma \) causes a runtime error.
  
  - Here, \( \bot_e \) is used as a pseudo-value of an expression, to indicate an error. It's not a value; we're writing it in place of an actual value.
  
  - If \( e \) can fail at runtime, then instead of \( \sigma(e) \in V \) for some set of values \( V \), we now have \( \sigma(e) \in V \cup \{ \bot_e \} \). Of course, some expressions never fail: \( \sigma(2+2) \in \mathbb{Z} \cup \{ \bot_e \} \) but more specifically, \( \sigma(2+2) \in \mathbb{Z} \).

• **Primary errors:** The primitive values and operations being supported determines what basic runtime errors can occur. For us, let's include:
  
  - *Array index out of bounds:* \( \sigma(b[e]) = \bot_e \) if \( \sigma(e) < 0 \) or \( \sigma(e) \geq \sigma(size(b)) \); similar for multiple dimensions.
  
  - *Division by zero:* \( \sigma(e_1/e_2) = \sigma(e_1 \% e_2) = \bot_e \) if \( \sigma(e_2) = 0 \).
  
  - *Square root of negative number:* \( \sigma(sqrt(e)) = \bot_e \) if \( \sigma(e) < 0 \).

• **Example 6:** \( b[1] \), \( n/0 \), and \( sqrt(-1) \) fail for all \( \sigma \). \( b[k] \) fails in state \( \{ b = (2, 3, 5, 8) \} \) but not in state \( \{ b = (6) \} \).

• **Hereditary Failure:** If evaluating a subexpression fails, then the overall expression fails.
  
  - If \( op \) is a unary operator, then \( \sigma(op \ e) = \bot_e \) if \( \sigma(e) = \bot_e \).
  
  - If \( op \) is a binary operator, then \( \sigma(e_1 \ op \ e_2) = \bot_e \) if \( \sigma(e_1) = \bot_e \) or \( \sigma(e_2) = \bot_e \).
  
  - For a conditional expression, \( \langle if \ B \ then \ e_1 \ else \ e_2 \ fi \rangle = \bot_e \) if one of the following three situations occurs: (1) \( \sigma(B) = \bot_e \) (2) \( \sigma(B) = T \) and \( \sigma(e_1) = \bot_e \) or (3) \( \sigma(B) = F \) and \( \sigma(e_2) = \bot_e \). We don't worry about a hypothetical failure of the branch we don't evaluate.

• **Example 7:** \( \sigma(x/y) = \bot_e \) when \( \sigma(y) = 0 \), but \( \sigma(y = 0 \ ? \ 0 : x/y) \) never = \( \bot_e \).

**F. Statements With Runtime Errors**

• An expression that causes a runtime error causes the statement it appears in to terminate unsuccessfully. We'll write \( \langle S, \sigma \rightarrow \langle E, \bot_e \rangle \) for the operational semantics of such a statement. This use of \( \bot_e \) as a (pseudo)-state is different from its use as a pseudo-value (\( \sigma(e) = \bot_e \)).

• **Definition** (Statements with expressions with runtime errors) If a statement evaluates an expression that causes a runtime error, then the statement terminates unsuccessfully. To the operational semantics, we add:
  
  - If \( \sigma(e) = \bot_e \), then \( \langle v := e, \sigma \rightarrow \langle E, \bot_e \rangle \).
  
  - If \( \sigma(e_1) \) or \( \sigma(e_2) = \bot_e \), then \( \langle b[e_1] := e_2, \sigma \rightarrow \langle E, \bot_e \rangle \).
  
  - If \( \sigma(B) = \bot_e \), then \( \langle while \ B \ do \ S \ od, \sigma \rightarrow \langle E, \bot_e \rangle \).
  
  - If \( \sigma(B) = \bot_e \), then \( \langle if \ B \ then \ S_1 \ else \ S_2 \ fi, \sigma \rightarrow \langle E, \bot_e \rangle \).
  
  - If \( \langle S_1, \sigma \rightarrow \langle E, \bot_e \rangle \) then \( \langle S_1 ; S_2, \sigma \rightarrow \langle E, \bot_e \rangle \) where \( T_1 \) either is a statement or \( E \).
• The pseudo-states \( \perp_d \) and \( \perp_e \) share some properties, so it’s helpful to have a more general notation for “error”.

**Notation:** \( \perp \) refers generically to \( \perp_d \) and/or \( \perp_e \). For example, \( \langle S, \sigma \rangle \rightarrow^* \langle E, \perp \rangle \) means either \( \langle S, \sigma \rangle \rightarrow^* \langle E, \perp_d \rangle \) (evaluation of \( S \) causes a runtime error) or \( \langle S, \sigma \rangle \rightarrow^* \langle E, \perp_e \rangle \) (evaluation of \( S \) diverges).

**Notation:** \( \perp \in M(S, \sigma) \) means \( \langle S, \sigma \rangle \rightarrow^* \langle E, \perp \rangle \). (Here, \( \perp \) can be \( \perp_d \) or \( \perp_e \).)

**Trying to use** \( \perp \): Since we are writing \( \perp \) in some of the places where an actual memory state would appear, it’s good to be thorough, look at the other places states appear, and extend those notions or notations.

• When we say “for all states…” or “for some state…”, we don’t include \( \perp \).
• We can’t add a binding to \( \perp : \{ v \mapsto a \} = \perp \).
• We can’t bind a variable to \( \perp : \sigma(v) \neq \perp \) and \( \sigma[v \mapsto \perp] = \perp \).
• We can’t take the value of a variable or expression in \( \perp \): If \( \sigma = \perp \) then \( \sigma(v) = \sigma(e) = \perp \). (More succinctly, \( \perp(v) = \perp(e) = \perp \).)
• Operationally, execution halts as soon we generate \( \perp \) as a “state”: \( \langle S, \perp \rangle \rightarrow^0 \langle E, \perp \rangle \).
• Denotationally, we can’t run a program in \( \perp : M(S, \perp) = \{ \perp \} \)

• From the properties above, it follows that we can’t evaluate something after generating \( \perp \).
• If \( \langle S_1, \sigma \rangle \rightarrow \langle E, \perp \rangle \), then \( \langle S_1' ; S_2, \sigma \rangle \rightarrow \langle E, \perp \rangle \).
• If \( M(S_1, \sigma) = \{ \perp \} \), then \( M(S_1', S_2, \sigma) = M(S_2, M(S_1, \sigma)) = M(S_2, \perp) = \{ \perp \} \).
• If \( W = \text{while } B \text{ do } S_1 \text{ od} \) and \( \sigma(B) = T \) but \( M(S_1, \sigma) = \{ \perp \} \), then \( M(W, \sigma) = \{ \perp \} \).
  • (In detail, \( M(W, \sigma) = M(S_1 ; W, \sigma) = M(W, M(S_1, \sigma)) = M(W, \perp) = \{ \perp \} \).

**Satisfaction and Validity and** \( \perp \): \( \perp \) never satisfies a predicate: \( \perp \not\models p \) for all \( p \), even if \( p = \) the constant \( T \). In general, we now have three possibilities: \( \sigma \models p \), \( \sigma \models \neg p \), or \( \sigma = \perp \). So \( \sigma \models p \) is now equivalent to \( (\sigma \models \neg p \text{ or } \sigma = \perp) \), not just \( \sigma \models \neg p \). We can also have \( \sigma \models p \) and \( \sigma \models \neg p \) simultaneously (when \( \sigma = \perp \)).

**Logical negation and** \( \perp \): Since \( \sigma \models \neg p \) is no longer equivalent to \( \sigma \not\models p \), we need a better notion of what \( \neg p \) means. The solution is to treat \( \neg p \) as shorthand for \( p \rightarrow F \) where \( F \) is the predicate “false”.
  • Just a quick note: For the meaning of \( T \) and \( F \), we have \( \sigma \models T \) and \( \sigma \not\models F \) for all \( \sigma \). (We can also derive \( F \) by defining \( F = T \not\models T \).) For all \( \sigma \) (i.e., unless \( \sigma = \perp \)), \( \sigma \models F \rightarrow F \), so \( \sigma \not\models F \).

**Generating** \( \perp \) **while testing for satisfaction**: Another problem to worry about is what to do if evaluation of a predicate causes an error? Clearly, we can’t allow things like \( \{ y = 0 \} \models y/y = 1 \). To handle this, we’ll add \( \perp \) to the semantics of basic operations and tests:

• For any relation (like less than, etc), we have \( (\alpha \text{ relation } \beta) \) yields \( \perp \) if \( \alpha \) or \( \beta = \perp \).
• For any binary operation (like addition, etc), we have \( (\alpha \text{ operation } \beta) \) yields \( \perp \) if \( \alpha \) or \( \beta = \perp \).
• Similarly for a unary operation, we have \( (\text{operation } \perp) \) yields \( \perp \).

* I’m using "yields" here for “semantically evaluates to”. “Equals” or “≡” are okay as long as we remember we mean semantic equality.
• Some of the implications of this are reasonably intuitive: \( \bot + 1 \) yields \( \bot \).
• But some implications are less intuitive: Semantic operations and tests like \( \bot \neq 2 \), \( \bot < \bot \), \( \bot = \bot \), and \( \bot \neq \bot \) all yield \( \bot \) (not T or F).
• Returning to \( y/y = 1 \), we still have \( \sigma \models y/y = 1 \) iff \( \sigma(y/y) = \sigma(1) \) iff \( \sigma(y) \text{ divided by } \sigma(y) = \sigma(1) \) iff \( (\bot = \bot) \) iff \( \bot \).

G. Sequential Nondeterministic Programs, part 1

H. Avoiding Unnecessary Design Choices

• When writing programs, it’s hard enough concentrating on the decisions we have to make at any given time, so it’s helpful to avoid making decisions we don’t have to make.
• Example 8: A very simple example is a statement that sets \( \text{max} \) to the max of \( x \) and \( y \). It doesn’t really matter which of the following two we use. They’re written differently but behave the same:
  • \( \text{if } x \geq y \text{ then } \text{max} := x \text{ else } \text{max} := y \text{ fi} \)
  • \( \text{if } y \geq x \text{ then } \text{max} := y \text{ else } \text{max} := x \text{ fi} \)
• The difference is when \( x = y \), the first statement sets \( \text{max} := x \); the second sets \( \text{max} := y \). It doesn’t matter which one of these we choose, we just have to pick one.
• Our standard \textit{if-else} statement is \textit{deterministic}: It can only behave one way. A nondeterministic \textit{if-fi} will specify that one of \( \text{max} := x \) and \( \text{max} := y \) has to be run, but it won’t say how we choose which one.
  • We don’t plan to execute our programs nondeterministically; we design programs using nondeterminism in order to delay making unnecessary decisions about the order in which our code makes choices.
  • When we make the code more concrete by rewriting it using everyday deterministic code, then we’ll decide which way to write it.

I. Nondeterministic if-fi

• Syntax: \( \text{if } B_1 \rightarrow S_1 \; \boxdot \; B_2 \rightarrow S_2 \; \boxdot \; \ldots \; \boxdot \; B_n \rightarrow S_n \text{ fi} \)
  • The box symbols separate the different clauses, like commas in an ordered \( n \)-tuple.
  • Don’t confuse these right arrows with ones in other contexts (implication operator and single-step execution).
• Definition: In \( \text{if } B_1 \rightarrow S_1 \; \boxdot \; B_2 \rightarrow S_2 \; \boxdot \; \ldots \; \boxdot \; B_n \rightarrow S_n \text{ fi} \), each \( B_i \rightarrow S_i \) clause is a \textit{guarded command}.
  • The \textit{guard} \( B_i \) tells us when it’s okay to run \( S_i \).
• Informal semantics
• If exactly one of the guard tests $B_1, B_2, \ldots, B_n$ is true, then execute its corresponding statement.
• If more than one test is true, then nondeterministically select a corresponding statement and execute it.
• If no guard is true, abort with a runtime error.

• **Example 9**: The max-setting example can be written using $\text{if } x \geq y \rightarrow \text{max} := x \land y \geq x \rightarrow \text{max} := y \text{fi}$
  • If only one of $x \geq y$ and $y \geq x$ is true, we execute its corresponding assignment.
  • If both are true, then we choose one of the tests and execute its assignment.

• In the max example, we really don’t care which arm is executed because they set max to the same value.
  • In more general examples, the different arms might behave differently but as long as each gets us to where we’re going, we don’t care which one gets chosen.
  • E.g., say we have an if-fi with two arms; one arm sets a variable $z := 0$; the other arm sets $z := 1$. This is okay if after the if-fi all we need to ensure is, say, $z \geq 0$. (If we needed, e.g., \texttt{even}(z), then we’d have a bug.)

• Of course, not all code written with nondeterministic if-fi has to behave nondeterministically.
  • **Example 10**: The statement $\text{if } B \rightarrow S_1 \square \neg B \rightarrow S_2 \text{fi}$ behaves like our usual deterministic if-else.

### J. Nondeterministic Choices are Unpredictable

• For us, “nondeterministic” means “unpredictable”.

• Let $\text{flip()}$ be a function that returns 0 or 1, so the assignment $x := \text{flip()}$ behaves like a coin flip. Using $\text{if } T \rightarrow x := 0 \land T \rightarrow x := 1 \text{fi}$ models an nondeterministic $x := \text{flip()}$.

• Note $\text{flip()}$ is **nondeterministic**, so its behavior is completely unpredictable. A thousand coin flips might give us anything: all 0’s, all 1’s, some pattern, random 500 heads and 500 tails, etc.

• If $\text{flip()}$ modeled a random coin flip with a probability attached to the result, then we could talk about distributions and fairness — after a thousand coin flips, we’d expect roughly the same number of 0’s and 1’s.

• **Unpredictability shouldn’t matter for purposes of correctness**: The idea with nondeterministic code is that it makes choices where we don’t care about which outcome is chosen. So, no matter how we later rewrite the code deterministically, the result should still be correct.
  • If we need a program with fair random choices, we’ll have to code that in at some point, but before then we can concentrate on getting the right results once the choices are made. (E.g., make sure our code works whether we get heads or tails, and then later worry about how to toss the coin.)
  • Other programs (like the max program) are written nondeterministically because they make overlapping choices. Here, converting from nondeterministic code to deterministic code is when we’d have to decide (e.g.) in what order to do a list of if-else if tests.