Denotational Semantics; Runtime Errors; Nondeterminism pt 1

CS 536: Science of Programming, Fall 2019

9/10: v2

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 σ is τ if in state σ, program S terminates in τ. (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 τ instead of just τ 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 σ be a state and let $S \equiv \texttt{x} := 1 \texttt{;}\ \texttt{y} := 2$. Since $\langle \texttt{x} := 1 \texttt{;}\ \texttt{y} := 2, \sigma\rangle \rightarrow \langle \texttt{y} := 2, \sigma[\texttt{x} \mapsto 1]\rangle \rightarrow \langle E, \sigma[\texttt{x} \mapsto 1][\texttt{y} \mapsto 2]\rangle$, we know $M(S, \sigma) = \{\sigma[\texttt{x} \mapsto 1][\texttt{y} \mapsto 2]\}$.
- **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 state, so $M[\![S]\!](\sigma) = \tau$ means $M[\![S]\!]$ maps σ to τ.

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.
  - $M(\textbf{skip}, \sigma) = \{\sigma\}$
  - $M(v := e, \sigma) = \{\sigma[v \mapsto \sigma(e)]\}$
  - $M(\mathtt{b}[e_1] := e, \sigma) = \{\sigma[\mathtt{b}[\alpha] \mapsto \beta]\}$ where $\alpha = \sigma(e_1)$ and $\beta = \sigma(e)$.
- **Composition**: $M(S_1\ \textbf{;}\ S_2, \sigma) = M(S_2, \tau)$ where $\{\tau\} = M(S_1, \sigma)$. To justify this, say we have $\langle S_1\ \textbf{;}\ S_2, \sigma\rangle \rightarrow^* \langle S_2, \tau\rangle \rightarrow^* \langle E, \tau'\rangle$. Since $M(S_1, \sigma) = \{\tau\}$, we run S_2 starting in state τ, so $M(S_1\ \textbf{;}\ S_2, \sigma) = M(S_2, \tau) = M(S_2, M(S_1, \sigma))$.
- **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 $S_1\ \textbf{;}\ S_2$ 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 $\sigma(B) = \text{T}$, then $M(\textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}, \sigma) = M(S_1, \sigma)$
  - If $\sigma(B) = \text{F}$, then $M(\textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}, \sigma) = M(S_2, \sigma)$
- **Example 2**: Let $S \equiv$ **if** y **then** x := x+1 **else** z := x+2 **fi**, then
  - If $\sigma(\mathtt{y}) = \text{T}$, then $M(S, \sigma) = \{\sigma[\mathtt{x} \mapsto \sigma(\mathtt{x})+1]\}$
  - If $\sigma(\mathtt{y}) = \text{F}$, then $M(S, \sigma) = \{\sigma[\mathtt{z} \mapsto \sigma(\mathtt{x})+2]\}$
- **Iterative**: One way to definite the meaning of $W \equiv$ **while** B **do** S **od** is recursively:
  - If $\sigma(B) = \text{F}$ then $M(W, \sigma) = \{\sigma\}$
  - If $\sigma(B) = \text{T}$ then $M(W, \sigma) = M(S\ \textbf{;}\ W, \sigma) = M(W, M(S, \sigma))$.
  - 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: σ_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 $W \equiv$ **while** x < n **do** S **od**, where the loop body $S \equiv$ x := x+1 **;** y := y+y. The general case for the behavior of S is (for any τ), $M(S, \tau[\mathtt{x} \mapsto \alpha][\mathtt{y} \mapsto \beta]) = \{\tau[\mathtt{x} \mapsto \alpha+1][\mathtt{y} \mapsto 2\beta]\}$. Say we start execution of W in state $\sigma = \{\mathtt{x} = 0, \mathtt{n} = 3, \mathtt{y} = 1\}$. Our sequence of states is
  - $\sigma_0 = \sigma = \{\mathtt{x} = 0, \mathtt{n} = 3, \mathtt{y} = 1\}$
  - $M(S, \sigma_0) = \{\sigma_1\}$ where $\sigma_1 = \{\mathtt{x} = 1, \mathtt{n} = 3, \mathtt{y} = 2\}$
  - $M(S, \sigma_1) = \{\sigma_2\}$ where $\sigma_2 = \{\mathtt{x} = 2, \mathtt{n} = 3, \mathtt{y} = 4\}$, and

- $M(S, \sigma_2) = \{\sigma_3\}$ where $\sigma_3 = \{\mathtt{x} = 3, \mathtt{n} = 3, \mathtt{y} = 8\}$.
- Of this sequence, σ_3 is the first state that satisfies $\mathtt{x} \geq \mathtt{n}$, so $M(W, \sigma) = \{\sigma_3\} = \{\{\mathtt{x} = 3, \mathtt{n} = 3, \mathtt{y} = 8\}\}$.

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 σ **diverges** if it doesn't converge; i.e., $\langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle$ for no τ.
- **Notation**: $M(S, \sigma) = \{\perp_d\}$ ("**bottom sub-d**") means S diverges in σ. Note that although we're writing it in a place where you'd expect a memory state, $\perp_d$ is not an actual memory state; we'll call it a **pseudo-state** as apposed to an **actual** or **real** memory state like σ and τ.
  - **Note**: Divergence is one way in which a program doesn't successfully terminate. We'll introduce other flavors of $\perp$ as we look at other ways to not get successful termination.
- **Notation**: $\langle S, \sigma\rangle \rightarrow^* \langle E, \perp_d\rangle$ means that S starting in σ diverges. Again, we're not using $\perp_d$ as an actual memory state here, but since $M(S, \sigma) = \{\tau\}$ means $\langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle$, if we're going to write $M(S, \sigma) = \{\perp_d\}$ to say that S diverges, it's consistent to write $\langle S, \sigma\rangle \rightarrow^* \langle E, \perp_d\rangle$.
- To determine when $M(W, \sigma) = \{\perp_d\}$, 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 $M(W, \sigma) = \{\perp_d\}$ to indicate that no state in the sequence satisfies $\neg B$.
- **Example 4**: Let $W \equiv$ **while** $\mathtt{T}$ **do skip od** and σ be any state. Then $\langle W, \sigma\rangle \rightarrow \langle$**skip ;** $W, \sigma\rangle$ but $\langle$**skip ;** $W, \sigma\rangle \rightarrow \langle W, \sigma\rangle$. (As a directed graph, this is a two-node cycle, $\langle W, \sigma\rangle \rightleftarrows \langle$**skip ;** $W, \sigma\rangle$.) Hence $M(W, \sigma) = \{\perp_d\}$.
- **Example 5**: Let $W \equiv$ **while** $\mathtt{x} \neq \mathtt{n}$ **do** $\mathtt{x} := \mathtt{x} - 1$ **od** and let $\sigma = \{\mathtt{x} = -1, \mathtt{n} = 0\}$.
  - Let $\sigma_0 = \sigma = \{\mathtt{x} = -1, \mathtt{n} = 0\}$
  - Let $\{\sigma_1\} = M(\mathtt{x} := \mathtt{x} - 1, \sigma_0) = \{\sigma_0[\mathtt{x} \mapsto -2]\} = \{\{\mathtt{x} = -2, \mathtt{n} = 0\}\}$
  - Let $\{\sigma_2\} = M(\mathtt{x} := \mathtt{x} - 1, \sigma_1) = \{\sigma_1[\mathtt{x} \mapsto -3]\} = \{\{\mathtt{x} = -3, \mathtt{n} = 0\}\}$
  - In general, let $\{\sigma_k\} = M(\mathtt{x} := \mathtt{x} - 1, \sigma_{k-1}) = \{\{\mathtt{x} = -k - 1, \mathtt{n} = 0\}\}$
  - Since every $\sigma_k \vDash \mathtt{x} \neq \mathtt{n}$, we have $M(W, \sigma) = \{\perp_d\}$.

E. Expressions With Runtime Errors

- Using $\perp_d$ 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 σ(`y`) = 0 because evaluation of σ(`x/y`) fails. There are two notions of failure here: The expression fails, and this causes the statement to fail.
- **Definition**: $\sigma(e) = \perp_e$ means evaluation of e in state σ causes a runtime error.
  - Here, $\perp_e$ 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 $\sigma(e) \in V \cup \{\perp_e\}$. Of course, some expressions never fail: $\sigma(2+2) \in \mathbb{Z} \cup \{\perp_e\}$ 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*: σ(`b[`e`]`) = $\perp_e$ if $\sigma(e) < 0$ or $\geq \sigma$(`size(b)`); similar for multiple dimensions.
  - *Division by zero*: $\sigma(e_1 / e_2) = \sigma(e_1 \,\%\, e_2) = \perp_e$ if $\sigma(e_2) = 0$.
  - *Square root of negative number*: σ(`sqrt(`e`)`) = $\perp_e$ if $\sigma(e) < 0$.
- **Example 6**: `b[-1]`, `n/0`, and `sqrt(-1)` fail for all σ. `b[k]` fails in state {`b` = (2, 3, 5, 8), `k` = 4} but not in state {`b` = (6), `k` = 0}
- **Hereditary Failure**: If evaluating a subexpression fails, then the overall expression fails.
  - If *op* is a unary operator, then $\sigma(op\ e) = \perp_e$ if $\sigma(e) = \perp_e$.
  - If *op* is a binary operator, then $\sigma(e_1\ op\ e_2) = \perp_e$ if $\sigma(e_1)$ or $\sigma(e_2) = \perp_e$.
  - For a conditional expression, σ(**`if`** B **`then`** e_1 **`else`** e_2 **`fi`**) = $\perp_e$ if one of the following three situations occurs: (1) $\sigma(B) = \perp_e$ (2) $\sigma(B)$ = T and $\sigma(e_1) = \perp_e$ or (3) $\sigma(B)$ = F and $\sigma(e_2) = \perp_e$. We don't worry about a hypothetical failure of the branch we don't evaluate.
- **Example 7**: σ(`x/y`) = $\perp_e$ when σ(`y`) = 0, but σ(`y = 0 ? 0 : x/y`) never = $\perp_e$.

F. Statements With Runtime Errors

- An expression that causes a runtime error causes the statement it appears in terminate unsuccessfully. We'll write $\langle S, \sigma\rangle \rightarrow \langle E, \perp_e\rangle$ for the operational semantics of such a statement. This use of $\perp_e$ as a (pseudo)-state is different from its use as a pseudo-value ($\sigma(e) = \perp_e$).
- **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 $\sigma(e) = \perp_e$, then $\langle v := e, \sigma\rangle \to \langle E, \perp_e\rangle$.
- If $\sigma(e_1)$ or $\sigma(e_2) = \perp_e$, then $\langle \mathtt{b}[e_1] := e_2, \sigma\rangle \to \langle E, \perp_e\rangle$
- If $\sigma(B) = \perp_e$, then $\langle \textbf{while } B \textbf{ do } S \textbf{ od}, \sigma\rangle \to \langle E, \perp_e\rangle$
- If $\sigma(B) = \perp_e$, then $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma\rangle \to \langle E, \perp_e\rangle$
- If $\langle S_1, \sigma\rangle \to \langle T_1, \perp_e\rangle$ then $\langle S_1 \textbf{ ; } S_2, \sigma\rangle \to \langle E, \perp_e\rangle$ 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, $\langle S, \sigma\rangle \to^* \langle E, \perp\rangle$ means either $\langle S, \sigma\rangle \to^* \langle E, \perp_e\rangle$ (evaluation of S causes a runtime error) or $\langle S, \sigma\rangle \to^* \langle E, \perp_d\rangle$ (evaluation of S diverges).
- **Notation**: $\perp \in M(S, \sigma)$ means $\langle S, \sigma\rangle \to^* \langle E, \perp\rangle$. (Here, $\perp$ can be $\perp_d$ or $\perp_e$.)
  - 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.
- Errors are not actual memory states or actual values, so we define
  - $M(S, \perp) = \{\perp\}$
  - $\sigma[v \mapsto \perp] = \perp$. Also, $\perp[v \mapsto \alpha] = \perp$.
  - If $\sigma = \perp$, then $\sigma(e) = \perp$
  - If $\langle S_1, \sigma\rangle \to \langle E, \perp\rangle$, then $\langle S_1 \textbf{; } S_2, \sigma\rangle \to \langle E, \perp\rangle$ [as we saw above]
- From this last definition, it follows that
  - If $M(S_1, \sigma) = \{\perp\}$, then $M(S_1 \textbf{; } S_2, \sigma) = M(S_2, M(S_1, \sigma)) = M(S_2, \perp) = \{\perp\}$
  - Also, if $M(S_1, \sigma) = \{\perp\}$, then if $W \equiv \textbf{while } B \textbf{ do } S_1 \textbf{ od}$ and $\sigma(B) = \mathtt{T}$, then $M(W, \sigma) = M(S_1 \textbf{ ; } W, \sigma) = M(W, M(S_1, \sigma)) = M(W, \perp) = \{\perp\}$.
- **Errors and Satisfaction / Validity of predicates**: $\perp$ never satisfies a predicate: $\perp \not\models p$ for all p, even if $p \equiv$ the constant $\mathtt{T}$. In general, we now have three possibilities: $\sigma \models p$, $\sigma \models \neg p$, or $\sigma = \perp$. So $\sigma \not\models p$ is now equivalent to ($\sigma \models \neg p$ or $\sigma = \perp$), not just $\sigma \models \neg p$. We can also have $\sigma \not\models p$ and $\sigma \not\models \neg p$ simultaneously (when $\sigma = \perp$).
  - 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 $p \to \mathtt{F}$ where $\mathtt{F}$ is the predicate "false".
    - We can define the meaning of F by saying that $\sigma \not\models \mathtt{F}$ for all σ. Defining $\mathtt{F} \equiv 0 \neq 0$ is another approach.
    - It's straightforward to show properties like $\sigma \models \neg \mathtt{F}$ iff $\sigma \neq \perp$.

- The other problem to worry about is what to do if evaluation of a predicate causes an error?
  - Clearly, we can't allow things like $\{y=0\} \vDash y/y = 1$.
- To handle this, we'll add $\perp$ to the semantics of basic operations and tests:
  - For any *relation* (like less than, etc), we have (α *relation* β) yields* $\perp$ if α or $\beta = \perp$.
  - For any binary *operation* (like addition, etc), we have (α *operation* β) yields $\perp$ if α or $\beta = \perp$.
  - Similarly for a unary operation *op*, we have (*op* α) yields $\perp$ if $\alpha = \perp$.
- Some of the implications of this are reasonably intuitive: ($\perp$ plus one) yields $\perp$.
- But some implications are less intuitive: $\perp$ is also the result of $\perp \neq 2$ (e.g), $\perp < \perp$, $\perp = \perp$, and $\perp \neq \perp$.
- Returning to $y/y = 1$, we still have $\sigma \vDash y/y = 1$ iff $\sigma(y/y) = \sigma(1)$ iff ($\sigma(y)$ *divided by* $\sigma(y)$) = *one*, so
  - If $\sigma(y)$ = some $\alpha \neq 0$, then $\sigma \vDash y/y = 1$ iff (α *divided by* α = *one*) iff (*one* = *one*) iff *true*
  - But if $\sigma(y) = 0$, then $\sigma \vDash y/y = 1$ iff (0 *divided by* 0 = *one*) iff ($\perp$ = *one*) iff $\perp$.
  - Thus $\sigma \nvDash y/y = 1$ and similarly (since ($\perp \neq$ *one*) yields $\perp$), $\sigma \nvDash y/y \neq 1$.

Sequential Nondeterministic Programs, part 1 [added 9/10]

G. 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 1**: A very simple example is a statement that sets `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:
  - **`if`** `x≥y` **`then`** `max:=x` **`else`** `max:=y` **`fi`**
  - **`if`** `y≥x` **`then`** `max:=y` **`else`** `max:=x` **`fi`**
- The difference is when `x = y`, the first statement sets `max:=x`; the second sets `max:=y`. It doesn't matter which one of these we choose, we just have to pick one.
- Our standard **`if-else`** statement is **deterministic**: It can only behave one way. A nondeterministic **`if-fi`** will specify that one of `max:=x` and `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.

* I'm using "yields" here for "semantically evaluates to". Using "equals" or "=" might get confused with "semantically equal to".

- When we make the code more concrete by rewriting it using everyday deterministic code, then we'll decide which way to write it.

H. Nondeterministic if-fi

- **Syntax**: **`if`** $B_1 \rightarrow S_1 \; \square \; B_2 \rightarrow S_2 \; \square \; \ldots \; \square \; B_n \rightarrow S_n$ **`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 **`if`** $B_1 \rightarrow S_1 \; \square \; B_2 \rightarrow S_2 \; \square \; \ldots \; \square \; B_n \rightarrow S_n$ **`fi`**, each $B_i \rightarrow S_i$ clause is a **guarded command**.
  - The **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, …, 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 2**: The `max`-setting example can be written using **`if`** `x ≥ y → max := x  y ≥ x → max := y` **`fi`**
  - If only one of `x ≥ y` and `y ≥ 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 ≥ 0`. (If we needed, e.g., `even(z)`, then we'd have a bug.)
- Of course, not all code written with nondeterministic **`if-fi`** has to behave nondeterministically.
  - **Example 3**: The statement **`if`** $B \rightarrow S_1 \; \square \; \neg B \rightarrow S_2$ **`fi`** behaves like our usual deterministic **`if-else`**.

I. Nondeterministic Choices are Unpredictable

- For us, "nondeterministic" means "unpredictable".
- Let `flip()` be a function that returns 0 or 1, so the assignment `x := flip()` behaves like a coin flip.
- If `flip()` models a **random** coin flip with a probability attached to the result, then we can talk about distributions and fairness — after a thousand coin flips, we'd expect roughly the same number of 0's and 1's.
- If `flip()` is **nondeterministic**, then its behavior is completely unpredictable. A thousand coin flips might give us anything: Random results, all 0's, all 1's, some pattern, etc.

  - **Example 4**: Using **`if`** `T → x := 0 ▯ T → x := 1` **`fi`** models an nondeterministic `x := flip()`
- **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.

---------- ended 2019-09-09

Denotational Semantics; Runtime Errors; Nondeterminism pt 1

CS 536: Science of Programming, Fall 2019

A. Why

- Our simple programming language is a model for the kind of constructs seen in actual languages.
- Our programs stand for state transformers.
- 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 be able to

- Give the denotational semantics of a program in a state.
- Say when and how evaluation of an expression or program fails due to a runtime error.
- Evaluate a nondeterministic conditional statement (**`if-fi`**)

C. Problems

Denotational Semantics

Problems 1 – 4 are the denotational versions of the similar questions from Activity 5

1. What is
   a. $M(\texttt{x := x+1}, \{\texttt{x} = 5\})$?
   b. $M(\texttt{x := x+1}, \sigma)$? (Your answer will be symbolic.)
   c. $\langle \texttt{x := x+1; y := 2*x}, \{\texttt{x} = 5\} \rangle$?

2. Let $S \equiv$ **`if`** `x > 0` **`then`** `x := x+1` **`else`** `y := 2*x` **`fi`**.
   a. Let $\sigma(\texttt{x}) = 8$. What is $M(S, \sigma)$?
   b. Repeat, if $\sigma(\texttt{x}) = 0$.
   c. Repeat, if we don't know what $\sigma(\texttt{x})$ is. (Your answer will be symbolic.)

3. Let $S \equiv$ **`if`** `x > 0` **`then`** `x := x/z` **`fi`**.
   a. What is $M(S, \sigma)$ if $\sigma = \{\texttt{x} = 8, \texttt{z} = 3\}$? (Don't forget, integer division truncates)
   b. What is $M(S, \{\texttt{x} = -2, \texttt{z} = 3\})$?

4. Let $W \equiv$ **`while`** `x < 3` **`do`** S **`od`** where $S \equiv$ `x := x+1; y := y*x`.
   a. Evaluate the body S in an arbitrary state τ and give $M(S, \tau)$.

b. What is $M(W,\sigma)$ if $\sigma \vDash \texttt{x} = 4 \wedge \texttt{y} = 1$?

c. What is $M(W,\sigma)$ if where $\sigma \vDash \texttt{x} = 1 \wedge \texttt{y} = 1$?

Runtime Errors

5. Let $S \equiv \texttt{x} := \texttt{y/b[x]}$ and let $\sigma = \{\texttt{b} = (3, 0, -2, 4), \texttt{x} = \alpha, \texttt{y} = 13\}$. Find all α such that $M(S, \sigma) = \{\perp_e\}$.

   (Remember, integer division truncates.)

6. Repeat the previous problem on $S \equiv \texttt{y} := \texttt{y / sqrt(b[x])}$ and $\sigma = \{\texttt{b} = (-1, 9, 12, 0), \texttt{x} = \alpha, \texttt{y} = 8\}$. Treat `sqrt` as returning the truncated integer square root of its argument. (I.e., `sqrt`(0) = 0, `sqrt` of 1, 2, and 3 all = 1, `sqrt` of 4 through 8 = 2, etc.)

Nondeterminism, part 1

7. Let $IF \equiv \textbf{if}\ B_1 \rightarrow S_1 \square B_2 \rightarrow S_2 \square \ldots \square B_n \rightarrow S_n\ \textbf{fi}$ and $BB \equiv B_1 \vee B_2 \vee \ldots B_n$.

   a. What property does *BB* have to have for us to avoid a runtime error when executing *IF*?

   b. Does it matter if we reorder the guarded commands? (E.g., if we swap $B_1 \rightarrow S_1$ and $B_2 \rightarrow S_2$.)

8. Let $T_1 \equiv \textbf{if}\ B_1 \rightarrow S_1 \square B_2 \rightarrow S_2\ \textbf{fi}$ and $T_2 \equiv \textbf{if}\ B_1\ \textbf{then}\ S_1\ \textbf{else if}\ B_2\ \textbf{then}\ S_2\ \textbf{fi fi}$.

   a. Fill in the table below to describe what happens for each combination of B_1 and B_2 being true or false.

   b. For what kinds of states σ can these two statements behave differently?

If $\sigma \vDash \ldots$	T_1	T_2
$B_1 \wedge B_2$	*Executes* S_1 or S_2	
$B_1 \wedge \neg B_2$		
$\neg B_1 \wedge B_2$		
$\neg B_1 \wedge \neg B_2$.		

Solution to Activity 6 (Denotational Semantics; Runtime Errors, Nondeterminism pt 1)

Denotational Semantics

1. (Calculate meanings of programs)

   a. $M(\mathtt{x := x+1}, \{\mathtt{x}=5\}) = \{\{\mathtt{x}=5\}[\mathtt{x} \mapsto \{\mathtt{x}=5\}(\mathtt{x}+1)]\} = \{\{\mathtt{x}=6\}\}$

   b. $M(\mathtt{x := x+1}, \sigma) = \{\sigma[\mathtt{x} \mapsto \sigma(\mathtt{x}+1)]\} = \{\sigma[\mathtt{x} \mapsto \sigma(\mathtt{x})+1]\}$

   c. $M(\mathtt{x := x+1;\ y := 2*x}, \{\mathtt{x}=5\})$
   $= M(\mathtt{y := 2*x}, M(\mathtt{x := x+1}, \{\mathtt{x}=5\})$
   $= M(\mathtt{y := 2*x}, \{\mathtt{x}=6\})$ [from part (a)]
   $= \{\{\mathtt{x}=6\}[\mathtt{y} \mapsto \beta]\}$ where $\beta = \{\mathtt{x}=6\}(\mathtt{2*x}) = 12$
   $= \{\{\mathtt{x}=6, \mathtt{y}=12\}\}$

2. Let $S \equiv$ **if** $\mathtt{x}>0$ **then** $\mathtt{x := x+1}$ **else** $\mathtt{y := 2*x}$ **fi**.

   a. If $\sigma(\mathtt{x}) = 8$, then $\sigma(\mathtt{x}>0) = \mathrm{T}$, so $M(S, \sigma) = M(\mathtt{x := x+1}, \sigma) = \{\sigma[\mathtt{x} \mapsto \sigma(\mathtt{x}+1)]\} = \{\sigma[\mathtt{x} \mapsto 9]\}$

   b. If $\sigma(\mathtt{x}) = 0$, then $\sigma(\mathtt{x}>0) = \mathrm{F}$, so $M(S, \sigma) = M(\mathtt{y := 2*x}, \sigma) = \{\sigma[\mathtt{y} \mapsto \sigma(\mathtt{2*x})]\} = \{\sigma[\mathtt{y} \mapsto 0]\}$

   c. If $\sigma(\mathtt{x}) > 0$ then $M(S, \sigma) = M(\mathtt{x := x+1}, \sigma) = \{\sigma[\mathtt{x} \mapsto \sigma(\mathtt{x})+1]\}$
   If $\sigma(\mathtt{x}) \le 0$ then $M(S, \sigma) = M(\mathtt{y := 2*x}, \sigma) = \{\sigma[\mathtt{y} \mapsto 2 \times \sigma(\mathtt{x})]\}$

3. Let $S \equiv$ **if** $\mathtt{x}>0$ **then** $\mathtt{x := x/z}$ **fi** $\equiv$ **if** $\mathtt{x}>0$ **then** $\mathtt{x := x/z}$ **else skip fi**

   a. If $\sigma = \{\mathtt{x}=8, \mathtt{z}=3\}$, then $\sigma(\mathtt{x}>0) = \mathrm{T}$, so $M(S, \sigma) = M(\mathtt{x := x/z}, \sigma) = \{\sigma[\mathtt{x} \mapsto \alpha]\}$ where $\alpha = \sigma(\mathtt{x/z}) = \sigma[\mathtt{x} \mapsto 8/3] = \sigma[\mathtt{x} \mapsto 2]$, since integer division truncates.

   b. If $\sigma = \{\mathtt{x}=-2, \mathtt{z}=3\}$ then $\sigma(\mathtt{x}>0) = \mathrm{F}$, so $M(S, \sigma) =$ so $M($**skip**$, \sigma) = \{\sigma\}$.

4. Let $W \equiv$ **while** $\mathtt{x}<3$ **do** S **od** where $S \equiv \mathtt{x := x+1;\ y := y*x}$.

   a. For arbitrary τ,
   $M(S, \tau) = M(\mathtt{x := x+1;\ y := y*x}, \tau)$
   $= M(\mathtt{y := y*x}, \tau[\mathtt{x} \mapsto \tau(\mathtt{x})+1])$
   $= \{\tau[\mathtt{x} \mapsto \tau(\mathtt{x})+1][\mathtt{y} \mapsto \alpha]\}$ where $\alpha = \tau[\mathtt{x} \mapsto \tau(\mathtt{x})+1](\mathtt{y*x}) = \tau(\mathtt{y}) \times (\tau(\mathtt{x})+1)$

   b. If $\sigma \vDash \mathtt{x}=4 \wedge \mathtt{y}=1$, then $\sigma(\mathtt{x}<3) = \mathrm{F}$ so $M(W, \sigma) = \{\sigma\}$.

   c. If $\sigma \vDash \mathtt{x}=1 \wedge \mathtt{y}=1$, then $\sigma(\mathtt{x}<3) = \mathrm{T}$ so we have at least one iteration to do. Let $\sigma_0 = \sigma$, let $\sigma_1 = M(S, \sigma_0) = \sigma_0(\mathtt{y}) \times (\sigma_0(\mathtt{x})+1)$, and let $\sigma_2 = M(S, \sigma_1) = \sigma_1(\mathtt{y}) \times (\sigma_1(\mathtt{x})+1)$. Then
   $\sigma_0 = \sigma[\mathtt{x} \mapsto 1][\mathtt{y} \mapsto 1]$
   $\sigma_1 = M(S, \sigma_0) = \sigma_0[\mathtt{x} \mapsto \sigma_0(\mathtt{x})+1][\mathtt{y} \mapsto \sigma_0(\mathtt{y}) \times (\sigma_0(\mathtt{x})+1)] = \sigma[\mathtt{x} \mapsto 2][\mathtt{y} \mapsto 2]$
   $\sigma_2 = M(S, \sigma_1) = \sigma_1[\mathtt{x} \mapsto 2+1][\mathtt{y} \mapsto 2 \times (2+1)] = \sigma[\mathtt{x} \mapsto 3][\mathtt{y} \mapsto 6]$
   Since σ_0 and $\sigma_1 \vDash \mathtt{x} < 3$ but $\sigma_2 \vDash \mathtt{x} \ge 3$, we have $M(W, \sigma) = \{\sigma_2\} = \{\sigma[\mathtt{x} \mapsto 3][\mathtt{y} \mapsto 6]\}$.

Runtime Errors

5. $M(S, \sigma) = M(\texttt{x := y/b[x]}, \sigma) = \{\sigma[\texttt{x} \mapsto \gamma]\}$ where $\gamma = \sigma(\texttt{y/b[x]}) = 13 \,/\, \sigma(\texttt{b})(\alpha) = \bot_e$

   iff $\sigma(\texttt{b})(\alpha) = \bot_e$ or $\sigma(\texttt{b})(\alpha) = 0$

   iff (α is out of range for $\sigma(\texttt{b})$) or ($\sigma(\texttt{b})(\alpha) = 0$) (b[x] fails if x is out of range)

   iff ($\alpha < 0$ or $\alpha \geq 4$) or ($\sigma(\texttt{b})(\alpha) = 0$) ($\sigma$(b) has size 4)

   iff ($\alpha < 0$ or $\alpha \geq 4$) or ($\alpha = 1$) (b[1] is the only element = 0)

   iff $\neg$(α = 0, 2, or 3)

6. $M(S, \sigma) = M(\texttt{y := y / sqrt(b[x])}, \sigma) = \{\sigma[\texttt{y} \mapsto \beta]\}$ where $\beta = (\sigma(\texttt{y}) \,/\, \mathit{sqrt}(\gamma)) = (8 \,/\, \mathit{sqrt}(\gamma))$ and $\gamma = \sigma(\texttt{b})(\sigma(\texttt{x})) = \sigma(\texttt{b})(\alpha)$

   So $\beta = \bot_e$ (and thus $M(S, \sigma) = \{\sigma[\texttt{y} \mapsto \bot_e]\} = \{\bot_e\}$)

   iff $\gamma = \bot_e$ or $\gamma < 0$ or $\mathit{sqrt}(\gamma) = 0$ (i.e., b[x] fails, b[x] < 0, or sqrt(b[x]]) = 0)

   iff (α out of range for σ(b)] or $\gamma < 0$ or $\mathit{sqrt}(\gamma) = 0$ ($\gamma = \bot_e$ iff b[x] has a bad index)

   iff ($\alpha < 0$ or $\alpha \geq 4$) or $\gamma = \sigma(\texttt{b})(\alpha) < 0$ or $\mathit{sqrt}(\gamma) = 0$ (σ(b) is of size 4)

   iff ($\alpha < 0$ or $\alpha \geq 4$) or ($\alpha = 0$) or $\mathit{sqrt}(\gamma) = 0$ (only b[0] < 0)

   iff ($\alpha < 0$ or $\alpha \geq 4$) or ($\alpha = 0$) or ($\alpha = 3$) (only sqrt(b[3]) = sqrt(0) = 0)

   iff ($\alpha \leq 0$ or ≥ 3) (combining terms)

Nondeterminism, part 1

7. (Basic properties of nondeterministic if)
   a. We need $\sigma \vDash BB$, because if $\sigma \vDash \neg BB$, then $M(IF, \sigma) = \{\bot_e\}$. (In English: At least one guard must be true; if none of them are true, we get a runtime error.)
   b. The order of the guarded commands doesn't matter: If more than one guard is true, we nondeterministically choose one element from the set of corresponding statements, and in a set, the elements aren't ordered.

2. (Deterministic vs nondeterministic conditionals) Recall $T_1 \equiv$ **if** $B_1 \rightarrow S_1 \,\square\, B_2 \rightarrow S_2$ **fi** and $T_2 \equiv$ **if** B_1 **then** S_1 **else if** B_2 **then** S_2 **fi**.

a. Execution of T_1 and T_2:

If $\sigma \vDash \ldots$	T_1	T_2
$B_1 \wedge B_2$	Executes S_1 or S_2	Executes S_1
$B_1 \wedge \neg B_2$	Executes S_1	Executes S_1
$\neg B_1 \wedge B_2$	Executes S_2	Executes S_2
$\neg B_1 \wedge \neg B_2$.	Produces runtime error	Executes skip

b. T_1 and T_2 behave the same when one of B_1 and B_2 is true and the other is false. When both are true, T_2 always executes S_1 but T_1 will execute S_1 or S_2. When both of B_1 and B_2 are false, T_1 yields a runtime error but T_2 does nothing.

