A. Why

- Nondeterminism can help us avoid unnecessary determinism.
- Nondeterminism can help us develop programs without worrying about overlapping cases.

B. Objectives

At the end of this class you should know

- The syntax and operational and denotational semantics of nondeterministic statements.

C. Avoiding Unnecessary Design Choices Using Nondeterminism

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

D. Nondeterministic if-fi

- **Syntax**: `if B₁ → S₁ □ B₂ → S₂ □ ... □ Bₙ → Sₙ fi`
  - The box symbols separate the different arms, 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:** 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 none of the guard tests $B_1, B_2, ..., B_n$ are true, abort with a runtime error.
  - If exactly one guard $B_i$ is true then execute $S_i$.
  - If more than one guard is true, then select a corresponding statement and execute it.
    - The selection is made nondeterministically (unpredictably); we’ll discuss this more soon.

• **Example 2:** if $x \geq y \rightarrow max := x$ $\square$ $y \geq x \rightarrow max := y$ fi sets $max$ to the larger of $x$ and $y$.
  - If only one of $x \geq y$ and $y \geq x$ is true, we execute its corresponding assignment.
  - If both are true, we choose one of them and execute its assignment.
  - In this example, the two arms set $max$ to the same value when $x = y$, so it doesn’t matter which one gets used.
  - 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 $z := 1$. If, for correctness’s sake, we need $z \geq 0$ after the *if-fi*, then this is fine. (If we needed *even*(z), for example, we’d have a bug.)
  - We can also have *if-fi* statements that never have to make a nondeterministic choice.
    - **Example 3:** Our usual deterministic if $B$ then $S_1$ else $S_2$ fi can be written if $B \rightarrow S_1 \square \neg B \rightarrow S_2$ fi.

**E. Nondeterministic Choices are Unpredictable**

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

• Let $flip = if T \rightarrow x := 0 \square T \rightarrow x := 1$ fi, which sets $x$ to either 0 or 1. I’ve called it *flip* because it’s similar to a coin flip, but it’s not identical.
  - With a real coin flip, you expect a 50-50 chance of getting 0 or 1, but since flip is nondeterministic, its behavior is completely unpredictable.
  - A thousand calls of *flip* might give us anything: all 0’s, all 1’s, some pattern, random 500 heads and 500 tails, etc.

• **Nondeterminism shouldn’t affect correctness:** We write nondeterministic code when we don’t want to worry about how choices are made: We only want to worry about producing correct results given that a choice has been made.
  - E.g., code written using *flip* should produce a correct final state whether we get heads or tails. Of course, eventually, we’ll replace *flip* with a deterministic coin-flipping routine, and at that point we’ll have to worry about the fairness of the deterministic routine.

**F. Nondeterministic Loop**

• Nondeterministic loops are very similar to nondeterministic conditionals, both in syntax and semantics. We can derive nondeterministic loops using nondeterministic if and a *while* loop.
• **Syntax:** $\text{do } B_1 \rightarrow S_1 \quad \square \ B_2 \rightarrow S_2 \quad \square \ldots \quad \square \ B_n \rightarrow S_n \ \text{od}$

• **Informal semantics:**
  - At the top of the loop, check for any true guards.
  - If no guard is true, the loop terminates.
  - If exactly one guard is true, execute its corresponding statement and jump to the top of the loop.
  - If more than one guard is true, select one of the corresponding guarded statements and execute it. (The choice is nondeterministic.) Once we finish the guarded statement, jump to the top of the loop.
  
  A nondeterministic do loop is equivalent to a regular **while** loop (with a nondeterministic test but) with a nondeterministic if body. Let $BB = (B_1 \lor B_2 \ldots \lor B_n)$ be the disjunction of the guards, then $\text{do } B_1 \rightarrow S_1 \quad \square \ldots \quad \square \ B_n \rightarrow S_n \text{od}$ behaves like $\text{while } BB \text{ do } \text{if } B_1 \rightarrow S_1 \quad \square \ldots \quad \square \ B_n \rightarrow S_n \text{fi od}$. 

**G. Operational Semantics of Nondeterministic if-fi**

• Let $IF = \text{if } B_1 \rightarrow S_1 \quad \square \ B_2 \rightarrow S_2 \quad \square \ldots \quad \square \ B_n \rightarrow S_n \text{fi}$ and let $BB = B_1 \lor B_2 \ldots \lor B_n$.

• To evaluate $IF$,
  - If evaluation of any guard fails ($\sigma(BB) = \bot_e$), then $IF$ causes an error: $\langle IF, \sigma \rangle \rightarrow \langle E, \bot_e \rangle$.
  - If none of the guards are satisfied ($\sigma(BB) = F$), then $IF$ causes an error: $\langle IF, \sigma \rangle \rightarrow \langle E, \bot_e \rangle$.
  - If one or more guarded commands $B_k \rightarrow S_k$ have $\sigma(B_k) = T$, then one such $k$ is chosen nondeterministically and we jump to the beginning of $S_k$: $\langle IF, \sigma \rangle \rightarrow \langle S_k, \sigma \rangle$.

**H. Operational Semantics of Nondeterministic do-od**

• Let $DO = \text{do } B_1 \rightarrow S_1 \quad \square \ B_2 \rightarrow S_2 \quad \square \ldots \quad \square \ B_n \rightarrow S_n \text{od}$ and let $BB = B_1 \lor B_2 \ldots \lor B_n$.

• Evaluation of $DO$ is very similar to evaluation if $IF$:
  - If evaluation of any guard fails ($\sigma(BB) = \bot_e$), then $DO$ causes an error: $\langle DO, \sigma \rangle \rightarrow \langle E, \bot_e \rangle$.
  - If none of the guards are satisfied ($\sigma(BB) = F$), then the loop halts: $\langle DO, \sigma \rangle \rightarrow \langle E, \sigma \rangle$.
  - If one or more guarded commands $B_k \rightarrow S_k$ have $\sigma(B_k) = T$, then one such $k$ is chosen nondeterministically and we jump to the beginning of $S_k$; after it completes, we'll jump back to the top of the loop: $\langle DO, \sigma \rangle \rightarrow \langle S_k; DO, \sigma \rangle$.

**I. Denotational Semantics of Nondeterministic Programs**

• **Notation:**
  - $\Sigma$ is the set of all states (that proper for whatever we happen to be discussing at that time).
  - $\Sigma_\bot = \Sigma \cup \{\text{all flavors of } \bot\} = \Sigma \cup \{\bot_d, \bot_e\}$ right now; other versions can be added later.
  - If we just write $\bot$, then we mean one of $\bot_d$ or $\bot_e$; if both are possible, then we should be explicit, as in $\{\bot_d, \bot_e\}$.
• For a nondeterministic program, to get its denotational semantics, we have to collect all the possible final states (or pseudo-states if we get ⊥): \( M(S, \sigma) = \{ \tau \in \Sigma_\tau \mid \langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle \} \).

• For a deterministic program, there is only one such \( \tau \), so this simplifies to our earlier definition: \( M(S, \sigma) = \{ \tau \} \) where \( \langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle \) and \( \tau \in \Sigma_\tau \).

**Example 4:** Let \( S = if \ T \rightarrow x := 0 \ □ \ T \rightarrow x := 1 \ fi \). Then \( \langle S, \emptyset \rangle \rightarrow^* \langle E, x = 0 \rangle \) and \( \langle S, \emptyset \rangle \rightarrow^* \langle E, x = 1 \rangle \) are both possible, and \( M(S, \emptyset) = \{ x = 0 \}, \{ x = 1 \} \). (Be careful not to write this as \( \{ x = 0, x = 1 \} \), which is a set containing a single, ill-formed state.) For any particular execution of \( S \) in a \( \sigma \), we'll get exactly one of these final states.

**Notation:**

• For convenience, most times we can still abbreviate \( M(S, \sigma) = \{ \tau \} \) to \( M(S, \sigma) = \tau \). But let's agree not to shorten \( M(\text{skip}, \emptyset) = \emptyset \) to \( M(\text{skip}, \emptyset) = \emptyset \), since it might look like we're claiming that \( M(\text{skip}, \emptyset) \) has no final state — it does, the empty state.

• A nondeterministic program can have only one final state.

**Example 5:** The \( \text{max} \) program from Example 2 has only one final state. Let \( \text{Max} = if \ x \geq y \rightarrow \text{max} := x \ □ \ y \geq x \rightarrow \text{max} := y \ fi \), in the nondeterministic case, where \( x = y \), both possible execution paths take us to the same state: \( M(\text{Max}, \{ x = a, y = a \}) = \{ \{ x = a, y = a, \text{max} = a \} \} \).

• **Note:** To keep from confusing the grader, avoid writing things that look like multisets, such as \( "\{ \tau, \tau \} \) where \( \tau = \{ x = a, y = a, \text{max} = a \} \).

• For arbitrary \( S \), if \( M(S, \sigma) \) has > 1 member, then \( S \) is nondeterministic. The Max program shows us that the converse doesn’t hold: If \( M(S, \sigma) \) has just 1 member, \( S \) still could be nondeterministic.

• Also, the size of \( M(S, \sigma) \) can vary depending on \( \sigma \).

**Example 6:** If \( S = if \ x \geq 0 \rightarrow x := x \ □ \ x \leq 8 \rightarrow x := -x \ fi \), then \( M(S, \{ x = 0 \}) = \{ \{ x = 0 \} \} \), but \( M(S, \{ x = 3 \}) = \{ \{ x = 9 \}, \{ x = -3 \} \} \).

**Difference between \( M(S, \sigma) = \{ \tau \} \) and \( \tau \in M(S, \sigma) \)**

• There's a difference between \( M(S, \sigma) = \{ \tau \} \) and \( \tau \in M(S, \sigma) \). They both say that \( \tau \) can be a final state, but \( M(S, \sigma) = \{ \tau \} \) says there's only one final state, but \( \tau \in M(S, \sigma) \) leaves open the possibility that there are other final states.

• In particular, \( M(S, \sigma) = \{ \bot \} \) says \( S \) always causes an error whereas \( \bot \in M(S, \sigma) \) says that \( S \) might cause an error. Remember that when we write \( \bot \), we're being ambiguous as to whether we mean \( \bot_d \) or \( \bot_e \). If both kinds of failure are possible, we should be explicit: \( M(S, \sigma) = \{ \bot_d, \bot_e \} \) or \( \{ \bot_d, \bot_e \} \subseteq M(S, \sigma) \).

**J. Why Use Nondeterministic programs?**

• Without having defined program correctness yet, it's hard to motivate having nondeterministic programs, so I'll just make some general comments and we'll have to come back to this question at later times.
Reason 1: Nondeterminism Makes It Easy to Combine Partial Solutions

- With nondeterministic code, it's straightforward to combine partial solutions to a problem to form a larger solution. This means we can solve a large problem by solving smaller instances of it and combining them.

  • Example 7: Let's solve the Max problem. Say we specify "Max takes $x$ and $y$ and (without changing them), sets $max$ to the larger of $x$ and $y$.”
    - Since the program has to end with $max = x$ or $max = y$, we can approach the problem by asking “When does $max := x$ work?” and “When does $max := y$ work?”.
    - Since $max := x$ is correct exactly when $x \geq y$, the program $if \ x \geq y \Rightarrow max := x \ fi$ is correct.
    - Similarly, since $max := y$ is correct exactly when $x \leq y$, the program $if \ x \leq y \Rightarrow max := y \ fi$ is also correct.
    - We can combine the two partial solutions and get
      
      $if \ x \geq y \Rightarrow max := x$
      $\Box \ x \leq y \Rightarrow max := y$
    
      $fi$

    - This program works when $x \geq y$ or $x \leq y$, and since that covers all possibilities, our program is done.

Reason 2: Nondeterminism Makes it Easy to Handle Overlapping Cases

- In nondeterministic if/do, the order of the guarded commands makes no difference, so we can it doesn't matter if guards overlap in what states satisfy them. That means we can write the code nondeterministically with overlapping cases, not worry about there being overlapping cases, and wait to introduce asymmetry when it's unavoidable — i.e., when we rewrite the code deterministically.

  • Example 8: Let's take the Max program yet again.
    - Both programs below are correct:
      
      $if \ x \geq y \Rightarrow max := x$
      $\Box \ x \leq y \Rightarrow max := y$
    
      $fi$

    - Since the programs behave identically when $x = y$, it doesn't matter if we drop that case from one of the tests, say the second, which yields
      
      $if \ x \geq y \Rightarrow max := x \Box \ x < y \Rightarrow max := y \ fi$ and
      $if \ x \leq y \Rightarrow max := y \ Box \ x \geq y \Rightarrow max := x \ fi$

    - Introducing the asymmetry makes the code correspond to the deterministic statements
      
      $if \ x \geq y \ then \ max := x \ else \ max := y \ fi$ and
      $if \ x \leq y \ then \ max := y \ else \ max := x \ fi$

  • Example 9: Another example of introducing asymmetry is how
• if \( x \geq 0 \rightarrow y := \sqrt{x} \) \( \square \) \( x \leq 0 \rightarrow y := 0 \) \( fi \)
turns into \( if \ x \geq 0 \ then \ y := \sqrt{x} \) \( else \ y := 0 \) \( fi \), while
• if \( x \leq 0 \rightarrow y := 0 \) \( \square \) \( x \geq 0 \rightarrow y := \sqrt{x} \) \( fi \)
turns into \( if \ x \leq 0 \ then \ y := 0 \) \( else \ y := \sqrt{x} \) \( fi \)

**K. Example 10: Array Value Matching**

- As an example of how nondeterministic code can help us write programs, let's look at an array-matching problem. We're given three arrays, \( b_0, b_1, \) and \( b_2 \), all of length \( n \) and all sorted in non-descending order. The goal is to find indexes \( k_0, k_1, \) and \( k_2 \) such that \( b_0[k_0] = b_1[k_1] = b_2[k_2] \) if such values exist.
- But what if no such \( k_0, k_1, \) and \( k_2 \) exist? One solution is to terminate with \( k_0 = k_1 = k_2 = n \). This certainly works, but we need to test each index before testing its value:
  - We can't just test \( b_0[k_0] < b_1[k_1] \), we must test \( k_0 < n \land k_1 < n \land b_0[k_0] < b_1[k_1] \) (and assume that "\( \land \)" is short-circuiting).
  - Instead, we'll use sentinels as an alternative: Assume \( k_0[n] = k_1[n] = k_2[n] = +\infty \) (positive infinity). This lets us write tests like \( b_0[k_0] < b_1[k_1] \) without having to test for \( k_0 \) or \( k_1 = n \).
- How does the program work? Clearly we want to set \( k_0 = k_1 = k_2 = 0 \) initially, and we have to increment \( k_0 \) or \( k_1 \) or \( k_2 \) until we find a match.
- Let's study one pair of indexes, say \( k_0 \) and \( k_1 \). There are three cases:
  1. \( k_1 < n \land b_0[k_0] < b_1[k_1] \). If this happens, we should increment \( k_0 \). Since the arrays are sorted by \( \leq \), incrementing \( k_1 \) can't possibly result in \( b_0[k_0] = b_1[k_1] \), whereas incrementing \( k_0 \) might.
  2. \( k_0 < n \land b_0[k_0] > b_1[k_1] \). Symmetrically, if this happens, we should increment \( k_1 \).
  3. \( b_0[k_0] = b_1[k_1] \). If this happens, we don't want to do anything, since we have a possible match. (Of course, we still need \( b_1[k_1] = b_2[k_2] \) or \( b_0[k_0] = b_2[k_2] \) — they're equivalent in this case.)
- If we write this up as a nondeterministic if-fi, we get
  \( if \ b_0[k_0] < b_1[k_1] \rightarrow k_0 := k_0 + 1 \ \square \ b_0[k_0] > b_1[k_1] \rightarrow k_1 := k_1 + 1 \) \( fi \)
- Repeating for the other two pairs of indexes, we get
  \( if \ b_1[k_1] < b_2[k_2] \rightarrow k_1 := k_1 + 1 \ \square \ b_1[k_1] > b_2[k_2] \rightarrow k_2 := k_2 + 1 \) \( fi \)
  \( if \ b_0[k_0] < b_2[k_2] \rightarrow k_0 := k_0 + 1 \ \square \ b_0[k_0] > b_2[k_2] \rightarrow k_2 := k_2 + 1 \) \( fi \)
- If we repeat these three if-fi statements until none of the < or > cases apply, then we're guaranteed that = holds between each pair. We can combine the 6 cases above into:
// Program 10(a)
//
do b0[k0] < b1[k1] → k0 := k0+1
☐ b0[k0] > b1[k1] → k1 := k1+1
☐ b1[k1] < b2[k2] → k1 := k1+1
☐ b1[k1] > b2[k2] → k2 := k2+1
☐ b0[k0] < b2[k2] → k0 := k0+1
☐ b0[k0] > b2[k2] → k2 := k2+1
od

• If none of the loop guards apply, the ≤ and ≥ combine and ensure b0[k0] = b1[k1] = b2[k2].
• The code can be cleaned up a couple of ways. The obvious one is to combine guards that guard the same command:

// Program 10(b)
//
do b0[k0] < b1[k1] \lor b0[k0] < b2[k2] → k0 := k0+1
☐ b0[k0] > b1[k1] \lor b1[k1] < b2[k2] → k1 := k1+1
☐ b0[k0] > b2[k2] \lor b1[k1] > b2[k2] → k2 := k2+1
od

• The less obvious way is to note that if we don’t have b0[k0] = b1[k1] = b2[k2], then there must be a < relation between two of the three values. This gives us

// Program 11(c)
//
do b0[k0] < b1[k1] → k0 := k0+1
☐ b1[k1] < b2[k2] → k1 := k1+1
☐ b2[k2] < b0[k0] → k2 := k2+1  // (I flipped b0[k0] > b2[k2] for symmetry’s sake)
od

• For all three of the guards to be false, we need b0[k0] ≤ b1[k1] ≤ b2[k2], which only happens when b0[k0] = b1[k1] = b2[k2].
Sequential Nondeterminism

CS 536: Science of Programming, Spring 2021

A. Why

- Nondeterminism can help us avoid unnecessary determinism.
- Nondeterminism can help us develop programs without worrying about overlapping cases.

B. Objectives

At the end of these practice questions you should

- Be able to evaluate nondeterministic conditionals and loops.

C. Nondeterminism

1. Let $IF = \text{if } B_1 \rightarrow S_1 \Box B_2 \rightarrow S_2 \Box \ldots \Box B_n \rightarrow S_n \text{ fi}$ and $BB = B_1 \lor B_2 \lor \ldots \lor B_n$.
   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 $B_2 \rightarrow S_2$.

2. Let $U_1 = \text{if } B_1 \rightarrow S_1 \Box B_2 \rightarrow S_2 \text{ fi}$ and $U_2 = \text{if } B_1 \text{ then } S_1 \text{ else if } B_2 \text{ then } S_2 \text{ fi fi}$.
   a. Fill in the table below to describe what happens for each combination of $B_1$ and $B_2$ being true or false.

<table>
<thead>
<tr>
<th>If $\sigma \models \ldots$</th>
<th>$U_1$</th>
<th>$U_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_1 \land B_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$B_1 \land \neg B_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\neg B_1 \land B_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\neg B_1 \land \neg B_2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   b. For what kinds of states $\sigma$ can statements $U_1$ and $U_2$ behave differently?

3. Let $DO = \text{do } B_1 \rightarrow S_1 \Box B_2 \rightarrow S_2 \Box \ldots \Box B_n \rightarrow S_n \text{ od}$ and $BB = B_1 \lor B_2 \lor \ldots \lor B_n$. What property does $BB$ have to have for us to avoid an infinite loop when executing $DO$?
4. Consider the loop \( i := 0; \text{do } i < 1000 \rightarrow S_1; i := i+1 \quad \text{do } i < 1000 \rightarrow S_2; i := i+1 \text{ od} \) (where neither \( S_1 \) nor \( S_2 \) modifies \( i \)). Do we know anything about how many times or in what pattern we will execute \( S_1 \) vs \( S_2 \)?

5. Consider the loop \( x := 1; \text{do } x \geq 1 \rightarrow x := x+1 \quad \text{do } x \geq 2 \rightarrow x := x-2 \text{ od} \). Can running it lead to an infinite loop?

6. What are the reasons mentioned in the text for why using nondeterminism might be helpful?

7. What is \( M(S, \{x = 1\}) \) where \( S = \text{do } x \leq 20 \rightarrow x := x*2 \quad \text{do } x \leq 20 \rightarrow x := x*3 \text{ od} \) ?

Problems 8 - 10 all refer to the Array Value Matching problem in the notes (Example 10).

8. In the notes, we approached the problem by asking "What do we do if \( b_0[k0] < b_1[k1] \)" and so on for the other 5 tests. Another way to approach the problem is to ask "When do we want to increment \( k0 \) ?" and so on for the other 2 indexes. If we take this approach, which of the three programs 10(a), 10(b), or 10(c) do we wind up with?

9. Translate program 10(c) into a deterministic language like C, Java, or whatever.

10. Compare the guarded commands
    
    \[ b_0[k0] < b_1[k1] \rightarrow k0 := k0+1 \quad \text{and} \]
    
    \[ b_0[k0] < b_1[k1] \rightarrow k0 := k0+1; \text{do } b_0[k0] < b_1[k1] \rightarrow k0 := k0+1 \text{ od} \]
    
    Briefly discuss the difference between these commands. What does your translated program 10(b) look like if you start with the second command instead of the first?