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. An Example of 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 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 if-else statement is deterministic: It can only behave one way. A nondeterministic 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.

D. Nondeterministic if-fi

- **Syntax**: \( B_1 \rightarrow S_1 \circ B_2 \rightarrow S_2 \circ \ldots \circ B_n \rightarrow S_n \text{ fi} \)

  - 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 if \( B_1 \rightarrow S_1 \circ B_2 \rightarrow S_2 \circ \ldots \circ B_n \rightarrow S_n \) 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 \( B_1, B_2, \ldots, B_n \) 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 if \( x \geq y \rightarrow \max := x \circ y \geq x \rightarrow \max := y \) 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., 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 if \( B \rightarrow S_1 \square \neg B \rightarrow S_2 \) fi behaves like our usual deterministic if-else.

E. **Nondeterministic Choices are Unpredictable**

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

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

• Note 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 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.

F. Nondeterministic Loop

• In part 1, we covered nondeterministic conditional statements. Now we’ll look at loops with nondeterministic choices. Happily, it turns out they’re very similar to nondeterministic conditionals.

• Syntax: \( \text{do } B_1 \rightarrow S_1 \square B_2 \rightarrow S_2 \square \ldots \square B_n \rightarrow S_n \text{ od} \)

• 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, nondeterministically select one of the corresponding statements and execute it. Then jump to the top of the loop.

• Relationship between nondeterministic if-fi and do-od: Our nondeterministic do loop is equivalent to a regular while loop with a nondeterministic if body.
  \( \text{do } B_1 \rightarrow S_1 \square B_2 \rightarrow S_2 \square \ldots \square B_n \rightarrow S_n \text{ od} \) behaves like
  \( \text{while } BB \text{ do if } B_1 \rightarrow S_1 \square B_2 \rightarrow S_2 \square \ldots \square B_n \rightarrow S_n \text{ fi od} \) where \( BB = (B_1 \lor B_2 \lor \ldots \lor B_n) \)

G. Operational Semantics of Nondeterministic if-fi

• Let \( \text{IF} = \text{if } B_1 \rightarrow S_1 \square B_2 \rightarrow S_2 \square \ldots \square B_n \rightarrow S_n \text{ fi} \).

• For the semantics of \( \text{IF} \) in state \( \sigma \), first let \( BB = B_1 \lor B_2 \lor \ldots \lor B_n \) (the disjunction of the guards).

• If evaluation of any of the guards causes an error, then \( \text{IF} \) causes an error
  - If \( \sigma(BB) = \bot_e \) (equivalently, if \( \sigma \not\vdash BB \) and \( \sigma \not\vdash \neg BB \)) then \( \langle \text{IF}, \sigma \rangle \rightarrow \langle E, \bot_e \rangle \).

• If none of the guards are satisfied, then \( \text{IF} \) causes an error.
  - If \( \sigma \vdash \neg BB \) (equivalently, if \( [9/16] \sigma(BB) = F \)) then \( \langle \text{IF}, \sigma \rangle \rightarrow \langle E, \bot_e \rangle \).

• If one or more guards are satisfied, then one of them is chosen nondeterministically and we jump to the command it guards.
  - \( [9/16 \text{ starts}] \) If \( \sigma \vdash BB \), then the transition \( \langle \text{IF}, \sigma \rangle \rightarrow \langle S_k, \sigma \rangle \) is allowed, for all \( k \) where \( \sigma(B_k) = T \).
  - If we’re following an execution path then exactly one of the choices is taken; if we’re graphing out the set of all possible executions, then we draw all the choices. \( [9/16 \text{ ends}] \)

H. Operational Semantics of Nondeterministic do-od

• Let \( \text{DO} = \text{do } B_1 \rightarrow S_1 \square B_2 \rightarrow S_2 \square \ldots \square B_n \rightarrow S_n \text{ od} \) and let \( BB = B_1 \lor B_2 \lor \ldots \lor B_n \) (the disjunction of the guards). Then \( \text{DO} \) behaves like \( \text{while } BB \text{ do if } B_1 \rightarrow S_1 \square B_2 \rightarrow S_2 \square \ldots \square B_n \rightarrow S_n \text{ fi od} \).

• If evaluation of \( BB \) fails, then we fail.
I. Denotational Semantics of Nondeterministic Programs

- Recall we've defined \( M(S, \sigma) = \{\tau\} \) if \( \langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle \), where \( \sigma \) is the starting memory state and \( \tau \) is the ending memory state or \( \bot \). With nondeterministic programs, \( \tau \) might not be unique.

- **Example 5:** Let's reuse the program from Example 4: Let \( S = \text{if } x ::= 0 \text{ break } T \text{ then } x ::= 1 \text{ fi} \). Then \( \langle S, \emptyset \rangle \rightarrow^* [9/16] \langle E, x = 0 \rangle \) and \( \langle S, \emptyset \rangle \rightarrow^* \langle E, x = 1 \rangle \) are both possible.

  - Since \( M(S, \sigma) \) is supposed to hold all possible final states, we have \( M(S, \sigma) = \{\{x = 0\}, \{x = 1\}\} \).
  
  - For any single execution of a nondeterministic program, we'll get only one final state: Here, we'll terminate in one of \( \{x = 0\} \) or \( \{x = 1\} \), not both of them.

- **Notation:**
  
  - [9/16] \( \Sigma \) is the set of all states (that proper for whatever we happen to be discussing at this time).
  
  - [9/16] \( \Sigma_\bot = \Sigma \cup \{\text{all flavors of } \bot\} = \Sigma \cup \{\bot_d, \bot_e\} \) right now; other versions can be added later.

  - Again, for convenience, most times we can write \( M(S, \sigma) = \tau \) as a shorthand for \( M(S, \sigma) = \{\tau\} \), but there's one ambiguous case: we shouldn't write \( M(\text{skip}, \emptyset) = \emptyset \) as shorthand for \( M(\text{skip}, \emptyset) = \{\emptyset\} \).

- **Definition:** (A restatement) [9/16] \( M(S, \sigma) = \{\tau \in \Sigma_\bot \mid \langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle\} \), the set of all possible final states for \( S \) starting in \( \sigma \), possibly including \( \bot \). Note \( M(S, \sigma) \neq \emptyset \), because either \( S \) terminates in an actual memory state \( \in \sigma \), or it yields \( \bot \).

  - Even with a nondeterministic program, it's possible to have only one final state.

  - **Example 6:** The \textit{max} program from Example 1 only has one final state. If \( S = \text{if } x \geq y \rightarrow \text{max} ::= x \square y \geq x \rightarrow \text{max} ::= y \text{ fi} \), in the nondeterministic case where \( x = y \), it doesn't matter which guarded command we execute because both set \textit{max} to the same value: \( M(S, \{x = a, y = a\}) = \{\{x = a, y = a, \text{max} = a\}\} \). Note: We're using sets, not multisets, so don't write the same state twice.

  - So if \( S \) is deterministic, \( M(S, \sigma) \) has 1 member, and if for some \( \sigma \), \( M(S, \sigma) \) has > 1 member, then \( S \) is nondeterministic. But \( M(S, \sigma) \) having only 1 member doesn't imply that \( S \) is deterministic.

  - It's also possible to have programs that sometimes have a unique final state and sometimes don't.
• Example 7: If \( S = \text{if } x \geq 0 \rightarrow x := x^2 \text{ } \square \text{ } x \leq 8 \rightarrow x := -x \text{ } fi \), then \( M(S, \{x = 0\}) = \{x = 0\} \), but \( M(S, \{x = 3\}) = \{x = 9\}, \{x = -3\} \).

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

- \( M(S, \sigma) = \{\tau\} \) says that \( \tau \) is the only possible final result.
- \( \tau \in M(S, \sigma) \) says that \( \tau \) is a possible final result of \( S \) in \( \sigma \). If \( M(S, \sigma) \) has other members, then those are also possible results of \( S \) in \( \sigma \).
- In particular, \( M(S, \sigma) = \{\bot\} \) says \( S \) always causes an error; \( \bot \in M(S, \sigma) \) says that \( S \) might cause an error.
- If we just write \( \bot \), then we probably mean one of \( \bot_d \) or \( \bot_e \); we're either being lazy or vague and leaving off the subscript. If we mean both errors, we should probably write them out: \( M(S, \sigma) = \{\bot_d, \bot_e\} \), e.g.

**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.

**Nondeterminism Introduces Less Asymmetry in Tests**

- With an \( n \)-way test (i.e., \( n-1 \) nested if-else-if statements), sometimes we realize that a different ordering of the tests makes for simpler tests. Since regular if-else is asymmetric, it's hard to figure out when branches of nested if-else-if code can be reordered.
- In nondeterministic if/then, the order of the guarded commands makes no difference, so we can shuffle them as we like; sometimes this makes similar or overlapping conditions more obvious.

**Nondeterminism Makes It Easy to Combine Partial Solutions**

- It's often easier to solve a problem by solving parts of it and then combining the solutions.
- It's easy to combine solutions if they both are nondeterministic if statements.

**Example 8:** Here's one way we might develop the \( \text{max} \) program. (I know, I know, it's much more detailed than what you would do in practice, I'm just using it as an illustration.)

- We might start by saying, well, I know the \( \text{max} \) of \( x \) and \( y \) is either \( x \) or \( y \)
  - When is it \( x \)? Ah, \( \text{max} \) should be \( x \) if \( x \geq y \). I can write that as \( \text{if } x \geq y \rightarrow \text{max } := x \text{ } fi \).
  - This is only a partial solution, (it works if \( x \geq y \); if \( x < y \), I get an error) so I'm not done yet.
  - When is \( \text{max } = y \)? When \( y \geq x \). I can write that as \( \text{if } y \geq x \rightarrow \text{max } := y \text{ } fi \).
  - Again, this is only a partial solution: works if \( y \geq x \), fails if \( y < x \).
  - But I can merge the two solutions and get one that works when \( x \geq y \lor y \geq x \) is true.
if \( x \geq y \rightarrow \text{max} := x \) \( \square \) \( y \geq x \rightarrow \text{max} := y \) fi

- Since the disjunction of the guards \((x \geq y \lor y \geq x)\) covers all possible situations, there are no other cases to consider, so we're done!
- Also note that it doesn't matter if we thought of the \(\text{max} := y\) case before the \(\text{max} := x\) case; we'd end up with the same program but with the guarded commands swapped:

\[
\text{if } y \geq x \rightarrow \text{max} := y \quad \square \quad x \geq y \rightarrow \text{max} := x \fi
\]

**Nondeterminism Lets Us Put off Handling Overlapping Situations**

- In Example 8, we might have started with
  - Well, I know the \(\text{max}\) of \(x\) and \(y\) is either \(x\) or \(y\)
  - Hey, if \(x = y\) then \(\text{max} = x = y\). I can write that as if \(x = y \rightarrow \text{max} := x\) fi.
  - Actually, I could write it as if \(x = y \rightarrow \text{max} := y\) fi, so this works too:

\[
\text{if } x = y \rightarrow \text{max} := x \quad \square \quad x = y \rightarrow \text{max} := y \fi
\]
- But the second \(x = y\) doesn't cover any more cases tests than the first \(x = y\) test, so let's just go with

\[
\text{if } x = y \rightarrow \text{max} := x \fi
\]
- Then if we continue as in Example 8, we end up with a program with three guards:

\[
\text{if } x = y \rightarrow \text{max} := x \quad \square \quad x \geq y \rightarrow \text{max} := x \quad \square \quad y \geq x \rightarrow \text{max} := y \fi
\]
- At some point, we'll realize we have some overlapping solutions.
- We might decide to let the \(x = y\) case be swallowed up by another case; since \(x = y \rightarrow x \geq y\), the \(x = y\) case is redundant. We get

\[
\text{if } x \geq y \rightarrow \text{max} := x \quad \square \quad y \geq x \rightarrow \text{max} := y \fi
\]
- Or we might realize that \(x = y \rightarrow y \geq x\), let \(y \geq x\) subsume \(x = y\), and get the same program.
- Or we might decide to **subtract** the \(x = y\) case from the other cases to remove the overlaps

\[
\text{if } x = y \rightarrow \text{max} := x \quad \square \quad x > y \rightarrow \text{max} := x \quad \square \quad y > x \rightarrow \text{max} := y \fi
\]
- Or with the two-test program \((x \geq y \lor y \geq x)\), we might subtract the \(x \geq y\) case from the \(y \geq x\) case and remove that overlap:

\[
\text{if } x \geq y \rightarrow \text{max} := x \quad \square \quad y > x \rightarrow \text{max} := y \fi
\]
- This last program doesn't require a nondeterministic choice, so it can be rewritten using deterministic if:

\[
\text{if } x \geq y \text{ then } \text{max} := x \text{ else } \text{max} := y \fi
\]
- Going back to the \(x =, <, or > y\) program, when you see that two of the guarded command bodies are the same, you can just combine their guards:

\[
\text{if } x = y \rightarrow \text{max} := x \quad \square \quad x > y \rightarrow \text{max} := x \quad \square \quad y > x \rightarrow \text{max} := y \fi
\]
becomes

\[
\text{if } x = y \lor x > y \rightarrow \text{max} := x \quad \square \quad y > x \rightarrow \text{max} := y \fi
\]
which of course gets optimized to

\[
\begin{align*}
&\text{if } x \geq y \rightarrow \max := x \\
&\text{else } \max := y
\end{align*}
\]

and once again we rewrite this deterministically:

\[
\begin{align*}
&\text{if } x \geq y \text{ then } \max := x \text{ else } \max := y
\end{align*}
\]

K. Quicksort Partitioning Example

- As a more detailed example of how to use nondeterminism, we'll look at the partitioning problem that's part of quicksort.
- Recall that in quicksort, we take an array segment and partition it: Reorder its values into a block of all values < a "pivot" value, then the pivot value, and then a block of all values > the pivot. (The < or > blocks might be empty; if both are empty then the array segment is only one element long so is trivially sorted.) After partitioning, quicksort recursively sorts the two < and > blocks.
- The diagram below shows the general case: All the values in the <, =, and > sections are less than, equal to, or greater than the pivot value. The sections marked ?? contain unknown values, so we'll call them the left and right unknown sections.

<table>
<thead>
<tr>
<th></th>
<th>L</th>
<th>lt</th>
<th>eL</th>
<th>eR</th>
<th>gt</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>&lt;</td>
<td>??</td>
<td>=</td>
<td>??</td>
<td>&gt;</td>
<td></td>
</tr>
</tbody>
</table>

**Notation**: \( b[L..lt] < b[eL..eR] \) means all the values in \( b[L..lt] \) are < all the values in \( b[eL..eR] \). Similarly, \( b[eL..eR] < b[gt..R] \) means all the values in \( b[gt..R] \) are > all the values in \( b[eL..eR] \).

- The \( b[eL..eR] \) section (values that are equal to the pivot) will never be empty, but the others might:
  - The < section is empty: Define "Less than empty" as \( LTe = lt = L-1 \)
  - The left unknown section is empty: Define "Left Unknown empty" as \( LUe = eL = lt+1 \)
  - The right unknown is empty: Define "Right Unknown empty" as \( RUe = gt = eR+1 \)
  - The > section is empty: Define "Greater than empty" as \( GTe = gt = R+1 \)
- In a typical *Data Structures and Algorithms* course, the partitioning problem usually has only one unknown area that combines, say, the < and = areas into one \( \leq \) area. Since we're working nondeterministically, we can look at the problem very symmetrically. Once the algorithm is complete, when we translate it into a deterministic algorithm, we'll have to break any overlapping symmetries then.
- Back to the problem: Initially, \( LTe \land GTe \) holds; when \( LUe \land RUe \) holds, we're done partitioning. So a very high-level algorithm to solve the problem is
  \[
  \text{do } \neg LUe \rightarrow \text{process a L.U. entry } \Box \rightarrow \neg RUe \rightarrow \text{process a R.U. entry od}
  \]
- Here's a diagram detailing the borders around the unknown areas. To process a left unknown entry, we'll look at \( b[lt+1] \) to see if it's <, =, or > the pivot value. The goal is to increase the size of the <, =, or > sections by one, at \( lt+1, eL-1, \) or \( gt-1 \). However, we have to account for the left or right unknown areas being small or empty. This will complicate the > case.
Case 1: \( b[lt+1] < \) the pivots

If \( b[lt+1] < \) the pivots, then since \( lt+1 \) is adjacent to the \(<\) section, we just increment \( lt \).

\[
\text{if } b[lt+1] < b[eL] \rightarrow lt := lt+1 \\
\]  

Case 2: \( b[lt+1] = \) the pivots

If \( b[lt+1] = b[eL] \) then we'll move it to be adjacent to the left end of the \(=\) area by swapping \( b[lt+1] \) with the slot left of the \(=\) area; this will increase the \(=\) section. Note this still works if \( b[lt+1] \) is already just left of the \(=\) area (i.e., if \( lt+1 = eL-1 \)).

\[
\text{if } b[lt+1] = b[eL] \rightarrow \text{Swap}(lt+1, eL-1); eL := eL-1 \\
\]

Case 3: \( b[lt+1] > \) the pivots. If \( b[lt+1] > b[eL] \) there are two cases:

\( eR < gt-1 \): This is the simple case: The value at \( gt-1 \) is a right unknown value, so we can swap \( b[lt+1] \) and \( b(gt-1) \) (which increases the \(>\) section) and move \( gt \) left.

\[
\text{if } b[lt+1] > b[eL] \land eR < gt-1 \rightarrow \text{Swap}(lt+1, gt-1); gt := gt-1 \\
\]

\( eR = gt-1 \) If the value at \( gt-1 \) is not unknown, it must be part of the section = the pivots. We can swap \( b[eL-1] \) and \( b[eR] \) to move the \(=\) area leftward and decrement \( eL \) and \( eR \) to compensate.

\[
\text{if } b[lt+1] > b[eL] \land eR = gt-1 \\
\rightarrow \text{Swap}(eL-1, eR); eL := eL-1; eR := eR-1 \\
\]

There is a subtlety here: If the left unknown area consisted of just one value, then it was the one at \( eL-1 \) (i.e., \( eL-1 = lt+1 \)). In that case, we've swapped the \(>\) value into its correct position. On the other hand, if the left unknown area had more than one value, then we've just moved an unknown to \( gt-1 \) and formed a new right unknown area. We could add code to check for these two cases, but they can be detected and handled at the next iteration, so for simplicity, I'll omit them.

- The code for handling the right unknown value at \( gt-1 \) is symmetric to the left unknown code.

Combining all the cases and substituting the code for detecting whether the left/right unknown areas are nonempty gives

\[
\text{do } eL > lt+1 \rightarrow \text{process a L.U. entry} \quad \square \quad eR+1 < gt \rightarrow \text{process a R.U. entry} \quad \text{od} \\
\]

- Process a L.U. entry

\[
\text{if } b[lt+1] < b[eL] \rightarrow lt := lt+1 \\
\]

\[
\square \quad b[lt+1] = b[eL] \rightarrow \text{Swap}(lt+1, eL-1); eL := eL-1 \\
\]

\[
\square \quad b[lt+1] > b[eL] \land eR < gt-1 \rightarrow \text{Swap}(lt+1, gt-1); gt := gt-1 \\
\]

\[
\square \quad b[lt+1] > b[eL] \land eR = gt-1 \\
\rightarrow \text{Swap}(eL-1, eR); eL := eL-1; eR := eR-1 \\
\]  

\fi
• Process a R.U. entry =
  
  if $b[gt-1] > b[eR]$ → $gt := gt-1$ fi

  □ $b[gt-1] = b[eR] → Swap(gt-1, eR+1); eR := eR+1$ fi

  □ $b[gt-1] < b[eR] ∧ eL > lt+1 → Swap(gt-1, lt+1); lt := lt+1$ fi

  □ $b[gt-1] < b[eR] ∧ eL = lt+1 → Swap(eR+1, eL); eL := eL+1; eR := eR+1$ fi

• If you study the code, you'll see that when we choose an unknown entry to process, whether it's a left or right unknown is chosen nondeterministically. To make the algorithm deterministic, we'll have to make that decision deterministically. For example, we might decide to always process a left unknown entry if one exists and move onto right unknown entries afterwards.