Formalism is just a tool

- Formal systems are common
Formalism is just a tool

- Formal systems are common
  - High school algebra
  - Classic formal logic
  - Euclidean geometry
Formalism is just a tool

- Formal systems are common
  - High school algebra
  - Classic formal logic
  - Euclidean geometry
- They serve multiple useful purposes
Formalism is just a tool

- Formal systems are common
  - High school algebra
  - Classic formal logic
  - Euclidean geometry

- They serve multiple useful purposes
  - Limit the possibilities that you may consider
  - Check whether reasoning is correct
  - Enable automated techniques for finding solutions
Formalism is just a tool

- Formal systems are common
  - High school algebra
  - Classic formal logic
  - Euclidean geometry

- They serve multiple useful purposes
  - Limit the possibilities that you may consider
  - Check whether reasoning is correct
  - Enable automated techniques for finding solutions

- Choosing the *right* tool for the job can be hard
Formalism is just a tool

- Several specific systems are common (in CS and program analysis)
Formalism is just a tool

- Several specific systems are common (in CS and program analysis)
  - Order Theory
    - How to compare elements of a set
Formalism is just a tool

• Several specific systems are common (in CS and program analysis)
  – Order Theory
  – Formal Grammars & Automata

Use structure to constrain the elements of a set
Formalism is just a tool

- Several specific systems are common (in CS and program analysis)
  - Order Theory
  - Formal Grammars & Automata
  - Formal Logic (Classical & otherwise)

How and when to infer facts
Formalism is just a tool

- Several specific systems are common (in CS and program analysis)
  - Order Theory
  - Formal Grammars & Automata
  - Formal Logic (Classical & otherwise)

- We are going to revisit these (quickly) with some insights on how they can be useful in practice.
Formalism is just a tool

- Several specific systems are common (in CS and program analysis)
  - Order Theory
  - Formal Grammars & Automata
  - Formal Logic (Classical & otherwise)

- We are going to revisit these (quickly) with some insights on how they can useful in practice.
  - Most students don’t seem to remember them
Formalism is just a tool

• Several specific systems are common (in CS and program analysis)
  – Order Theory
  – Formal Grammars & Automata
  – Formal Logic (Classical & otherwise)

• We are going to revisit these (quickly) with some insights on how they can useful in practice.
  – Most students don’t seem to remember them
  – Even fewer learn that formalism can be useful!
Formalism is just a tool

- Several specific systems are common (in CS and program analysis)
  - Order Theory
  - Formal Grammars & Automata
  - Formal Logic (Classical & otherwise)

- We are going to revisit these (quickly) with some insights on how they can useful in practice.
  - Most students don’t seem to remember them
  - Even fewer learn that formalism can be useful!
  - These techniques are critical for static program analysis
Order Theory

- *Order theory* is a field examining how we compare elements of a set.
Order Theory

- *Order theory* is a field examining how we compare elements of a set.
- Simplest example is numbers on a number line:

```
-4 -3 -2 -1 0 1 2 3 4
```

Set: $\mathbb{Z}$  Relation: $\leq$
Order Theory

- *Order theory* is a field examining how we compare elements of a set.
- Simplest example is numbers on a number line:

\[
\begin{align*}
\text{Set: } & \mathbb{Z} \\
\text{Relation: } & \leq
\end{align*}
\]
Order Theory

- *Order theory* is a field examining how we compare elements of a set.

- Simplest example is numbers on a number line:

  ![Number Line](image)

  - Set: \( \mathbb{Z} \)
  - Relation: \( \leq \)

- \( \leq \) is a *total order* on \( \mathbb{Z} \).
  - Intuitively, \( \forall a, b \in \mathbb{Z} \), either \( a \leq b \) or \( b \leq a \)
Order Theory

- We often want to compare complex data
Order Theory

- We often want to compare complex data
  - Ordinal, multidimensional, ...
Order Theory

- We often want to compare complex data
  - Ordinal, multidimensional, ...
Order Theory

• We often want to compare complex data
  – Ordinal, multidimensional, ...

```
  |   |   |
  | 0 | 1 |
  |   |   |
  | 0 | 1 |

(1, 1) (2, 2)
```
Order Theory

- We often want to compare complex data
  - Ordinal, multidimensional, ...

What is the result of $(1,1) \leq (2,2)$?
We often want to compare complex data

- Ordinal, multidimensional, ...

What is the result of $(1,1) \leq (2,2)$?

We can take $\leq$ to be componentwise comparison.
Order Theory

- We often want to compare complex data
  - Ordinal, multidimensional, ...

What is the result of $(1,2) \leq (2,1)$?
Order Theory

- We often want to compare complex data
  - Ordinal, multidimensional, ...

- Componentwise comparison with tuples yields a partial order

\[ (1,2) \quad (2,2) \]
\[ (1,1) \quad (2,1) \]
Order Theory

- We often want to compare complex data
  - Ordinal, multidimensional, ...

- Componentwise comparison with tuples yields a partial order
  - Intuitively, not all elements are comparable
Order Theory

- We often want to compare complex data
  - Ordinal, multidimensional, ...

- Componentwise comparison with tuples yields a partial order
  - Intuitively, not all elements are comparable

Which of these 4 elements are comparable?
Partial Orders

- A relation $\leq$ is a *partial order* on a set $S$ if $\forall \ a, b, c \in S$
  - Reflexive: $a \leq a$
  - Antisymmetric: $a \leq b \& b \leq a \Rightarrow a = b$
  - Transitive: $a \leq b \& b \leq c \Rightarrow a \leq c$
Partial Orders

- A relation $\leq$ is a partial order on a set $S$ if $\forall a, b, c \in S$
  - Reflexive: $a \leq a$
  - Antisymmetric: $a \leq b \& b \leq a \Rightarrow a = b$
  - Transitive: $a \leq b \& b \leq c \Rightarrow a \leq c$
A relation $\leq$ is a partial order on a set $S$ if $\forall a, b, c \in S$

- Reflexive: $a \leq a$
- Antisymmetric: $a \leq b \& b \leq a \Rightarrow a = b$
- Transitive: $a \leq b \& b \leq c \Rightarrow a \leq c$
Partial Orders

• A relation \( \leq \) is a partial order on a set \( S \) if \( \forall a, b, c \in S \)
  
  – Reflexive: \( a \leq a \)
  
  – Antisymmetric: \( a \leq b \) & \( b \leq a \) \( \Rightarrow a = b \)
  
  – Transitive: \( a \leq b \) & \( b \leq c \) \( \Rightarrow a \leq c \)

How does a total order compare?
Partial Orders

- A relation $\leq$ is a partial order on a set $S$ if $\forall a, b, c \in S$
  - Reflexive: $a \leq a$
  - Antisymmetric: $a \leq b \& b \leq a \Rightarrow a = b$
  - Transitive: $a \leq b \& b \leq c \Rightarrow a \leq c$

- When reasoning about partial orders, we prefer $\sqsubseteq$
Partial Orders

- A relation $\leq$ is a **partial order** on a set $S$ if $\forall a, b, c \in S$
  - Reflexive: $a \leq a$
  - Antisymmetric: $a \leq b \& b \leq a \Rightarrow a = b$
  - Transitive: $a \leq b \& b \leq c \Rightarrow a \leq c$

- When reasoning about partial orders, we prefer $\sqsubseteq$

- Common partial orders include
  - substring, subsequence, subset relationships
Partial Orders

- A relation $\leq$ is a *partial order* on a set $S$ if $\forall \ a,b,c \in S$
  \[
  \begin{align*}
  \text{Reflexive:} & \quad a \leq a \\
  \text{Antisymmetric:} & \quad a \leq b \land b \leq a \Rightarrow a = b \\
  \text{Transitive:} & \quad a \leq b \land b \leq c \Rightarrow a \leq c
  \end{align*}
  \]

- When reasoning about partial orders, we prefer $\subseteq$

- Common partial orders include
  - substring, subsequence, subset relationships

- Examples:
  \[
  \begin{align*}
  ab \leq_{\text{str}} xabyz \\
  ab \leq_{\text{seq}} xaybz \\
  \{a,b\} \subseteq \{a,b,x,y,z\}
  \end{align*}
  \]
Partial Orders

- A relation $\leq$ is a *partial order* on a set $S$ if $\forall a, b, c \in S$
  - Reflexive: $a \leq a$
  - Antisymmetric: $a \leq b \& b \leq a \Rightarrow a = b$
  - Transitive: $a \leq b \& b \leq c \Rightarrow a \leq c$

- When reasoning about partial orders, we prefer $\sqsubseteq$

- Common partial orders include
  - substring, subsequence, subset relationships
  - componentwise orderings
Partial Orders

- A relation ≤ is a **partial order** on a set S if ∀ a, b, c ∈ S

  \[
  \begin{align*}
  \text{Reflexive:} & \quad a \leq a \\
  \text{Antisymmetric:} & \quad a \leq b \land b \leq a \Rightarrow a = b \\
  \text{Transitive:} & \quad a \leq b \land b \leq c \Rightarrow a \leq c
  \end{align*}
  \]

- Common partial orders include
  - substring, subsequence, subset relationships
  - componentwise orderings

\[
\begin{align*}
(1,1) \subseteq (1,2) \\
(1,1) \subseteq (2,2)
\end{align*}
\]
Partial Orders

- A relation $\leq$ is a **partial order** on a set $S$ if $\forall \ a, b, c \in S$
  - Reflexive: $a \leq a$
  - Antisymmetric: $a \leq b \land b \leq a \Rightarrow a = b$
  - Transitive: $a \leq b \land b \leq c \Rightarrow a \leq c$

- When reasoning about partial orders, we prefer $\sqsubseteq$

- Common partial orders include
  - substring, subsequence, subset relationships
  - componentwise orderings
  - functions (considering all input/output mappings)
Partial Orders

- A relation $\leq$ is a *partial order* on a set $S$ if $\forall a, b, c \in S$

  - Reflexive: $a \leq a$
  - Antisymmetric: $a \leq b \& b \leq a \Rightarrow a = b$
  - Transitive: $a \leq b \& b \leq c \Rightarrow a \leq c$

- When reasoning about partial orders, we prefer $\sqsubseteq$

- Common partial orders include
  - substring, subsequence, subset relationships
  - componentwise orderings
  - functions (considering all input/output mappings)
We can express the structure of partial orders using (semi-)lattices.

(1,2) (2,2)
(1,1) (2,1)
Partial Orders

- We can express the structure of partial orders as (semi-)lattices.
Partial Orders

- We can express the structure of partial orders as (semi-)lattices.
Partial Orders

• We can express the structure of partial orders as (semi-)lattices.
Partial Orders

- We can express the structure of partial orders as (semi-)lattices.
Partial Orders

- We can express the structure of partial orders as *(semi-)*lattices.

- If unique least/greatest elements exist, we call them \(\bot\) (bottom)/\(\top\) (top)

![Diagram showing a partial order with elements (0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2) and their relationships.](image)
Partial Orders

- We are often interested in upper and lower bounds.
Partial Orders

- We are often interested in upper and lower bounds.
  - A \textit{join} $a \sqcup b$ is the least upper bound of $a$ and $b$

What is $(0,1) \sqcup (1,0)$?
Partial Orders

- We are often interested in upper and lower bounds.
  - A *join* $a \sqcup b$ is the least upper bound of $a$ and $b$

What is $(0,1) \sqcup (1,0)$?
We are often interested in upper and lower bounds.

- A **join** $a \sqcup b$ is the least upper bound of $a$ and $b$
- A **meet** $a \sqcap b$ is the greatest lower bound of $a$ and $b$

What is $(0,1) \sqcap (1,0)$?
Partial Orders

- We are often interested in upper and lower bounds.
  - A *join* $a \sqcup b$ is the least upper bound of $a$ and $b$
  - A *meet* $a \sqcap b$ is the greatest lower bound of $a$ and $b$

What is $(0,1) \sqcap (1,0)$?
We are often interested in upper and lower bounds.

- A *join* $a \sqcup b$ is the least upper bound of $a$ and $b$
- A *meet* $a \sqcap b$ is the greatest lower bound of $a$ and $b$
- Bounds must be unique and may not exist.
Partial Orders

- We are often interested in upper and lower bounds.
  - A join \( a \sqcup b \) is the least upper bound of \( a \) and \( b \).
  - A meet \( a \sqcap b \) is the greatest lower bound of \( a \) and \( b \).
  - Bounds must be unique and may not exist.
  - For all \( S' \subseteq S \), \( \exists \sqcup S' \& \sqcap S' \Rightarrow \) lattice, \( \exists \sqcup S' \) or \( \exists \sqcap S' \Rightarrow \) semilattice

What is the structure shown?
Partial Orders

- A product of lattices yields a lattice
  - We already saw componentwise orderings for tuples. This is the same.
Partial Orders

- A product of lattices yields a lattice
  - We already saw componentwise orderings for tuples. This is the same.

- Partial orders & lattices can be very useful
  - A formal structure for reasoning about relative value
Partial Orders

• A product of lattices yields a lattice
  – We already saw componentwise orderings for tuples. This is the same.

• Partial orders & lattices can be very useful
  – A formal structure for reasoning about relative value
  – modern cryptography
Partial Orders

- A product of lattices yields a lattice
  - We already saw componentwise orderings for tuples. This is the same.

- Partial orders & lattices can be very useful
  - A formal structure for reasoning about relative value
  - modern cryptography
  - concurrency & distributed systems
Partial Orders

• A product of lattices yields a lattice
  – We already saw componentwise orderings for tuples. This is the same.

• Partial orders & lattices can be very useful
  – A formal structure for reasoning about relative value
  – modern cryptography
  – concurrency & distributed systems
  – dataflow analysis & proving program properties
Formal Grammars & Automata

- Grammars define the structure of elements in a set
  - Alternatively, they generate the set via structure
Formal Grammars & Automata

- Grammars define the structure of elements in a set
  - Alternatively, they generate the set via structure
- They commonly define *formal languages*
  - Sets of strings over a defined alphabet
Formal Grammars & Automata

- Grammars define the structure of elements in a set
  - Alternatively, they generate the set via structure
- They commonly define *formal languages*
  - Sets of strings over a defined alphabet
- They are effective at constraining a search space
A regular language can be expressed via a regular expression.
A regular language can be expressed via a regular expression.

```
regex → symbol
| `(regex)`
| `regex`*
| regex `|` regex
```
A regular language can be expressed via a regular expression

```
regex → symbol
| `( `regex `)`
| regex `*`
| regex `|` regex
```

e.g. a(bc | cd)*e defines L containing abccdbce
Regular Languages & Finite Automata

- A regular language can be expressed via a regular expression
- Finite automata can be used to recognize or generate elements of a regular language
Regular Languages & Finite Automata

- A regular language can be expressed via a regular expression.
- Finite automata can be used to recognize or generate elements of a regular language.
Regular Languages & Finite Automata

- A regular language can be expressed via a regular expression
- Finite automata can be used to recognize or generate elements of a regular language

\[ a(bc \mid cd)^*e \] recognizes \( L \) containing \( abccdbce \)
Regular Languages & Finite Automata

- A regular language can be expressed via a regular expression.
- Finite automata can be used to recognize or generate elements of a regular language.

Example: The regular expression $a(bc \mid cd)^*e$ recognizes the language $L$ containing $abccdbce$. 

Diagram: [Diagram showing a finite automaton with states 1, 2, 3, and 4, labeled with transitions for $a, b, c, d,$ and $e$.]
Regular Languages & Finite Automata

- A regular language can be expressed via a regular expression.
- Finite automata can be used to recognize or generate elements of a regular language.

\[ a(bc \mid cd)^*e \] recognizes \( L \) containing \( abccdbce \).
A regular language can be expressed via a regular expression.

Finite automata can be used to recognize or generate elements of a regular language.

For example, \(a(bc | cd)^*e\) recognizes \(L\) containing \(abccdbce\).
Regular Languages & Finite Automata

- A *regular language* can be expressed via a *regular expression*.

- Finite automata can be used to *recognize* or *generate* elements of a regular language.

  e.g. $a(bc \mid cd)^*e$ recognizes $L$ containing $abccdbce$.
Regular Languages & Finite Automata

- A regular language can be expressed via a regular expression.
- Finite automata can be used to recognize or generate elements of a regular language.
- Recall, regular languages cannot express matched parentheses (Dyck languages)
  \[ a^n b^n \]
Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
• **Context free grammars** add recursion and enable Dyck language recognition

```
Start = A
A → cBd
B → eBf
| g
```
Context Free Grammars & Pushdown Automata

- **Context free grammars** add recursion and enable Dyck language recognition

Start = A  
A → cBd  
B → eBf  
| g

cen gfn d
Context Free Grammars & Pushdown Automata

- **Context free grammars** add recursion and enable Dyck language recognition

\[
\begin{align*}
\text{Start} &= A \\
A &\to cBd \\
B &\to eBf \\
&\mid g
\end{align*}
\]

This requires some kind of memory
Context Free Grammars & Pushdown Automata

- **Context free grammars** add recursion and enable Dyck language recognition

Start = A
A → cBd
B → eBf
| g

ce^n gf^n d
Context Free Grammars & Pushdown Automata

Context free grammars add recursion and enable Dyck language recognition.

Start = A
A → cBd
B → eBf
| g

\[ c e^n g f^n d \]
Context Free Grammars & Pushdown Automata

- **Context free grammars** add recursion and enable Dyck language recognition

Start = A
A → cBd
B → eBf | g

cBd|eBf

c^n g^n d

Generating symbols out of order acts as a form of memory.
Context Free Grammars & Pushdown Automata

- **Context free grammars** add recursion and enable Dyck language recognition

\[
\begin{align*}
\text{Start} &= A \\
A &\rightarrow cBd \\
B &\rightarrow eBf \mid g \\
\end{align*}
\]

\[ce^n g f^n d\]
Context Free Grammars & Pushdown Automata

- **Context free grammars** add recursion and enable Dyck language recognition
  - The grammar for regular expressions was a CFG!

```
regex → symbol
| `( regex )`
| regex `
| regex `| regex`
```
Context Free Grammars & Pushdown Automata

- **Context free grammars** add recursion and enable Dyck language recognition
  - The grammar for regular expressions was a CFG!

```plaintext
regex → symbol
| (` regex `)
| regex `*
| regex `| ` regex
```
Context Free Grammars & Pushdown Automata

- Context free grammars add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via pushdown automata)
Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition

- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

S → xAy | zB
A → aA | t
B → bB | u
• *Context free grammars* add recursion and enable Dyck language recognition

• Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

```
S → xAy | zB
A → aA | t
B → bB | u
```
Context Free Grammars & Pushdown Automata

- Context free grammars add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

\[
S \rightarrow xAy \mid zB \\
A \rightarrow aA \mid t \\
B \rightarrow bB \mid u
\]
Context Free Grammars & Pushdown Automata

- **Context free grammars** add recursion and enable Dyck language recognition

- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

\[
egin{align*}
S & \rightarrow xAy \mid zB \\
A & \rightarrow aA \mid t \\
B & \rightarrow bB \mid u
\end{align*}
\]
Context Free Grammars & Pushdown Automata

- **Context free grammars** add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via pushdown automata)

$S \rightarrow xAy \mid zB$

$A \rightarrow aA \mid t$

$B \rightarrow bB \mid u$

S

xaaty

x

...
Context Free Grammars & Pushdown Automata

- **Context free grammars** add recursion and enable Dyck language recognition

- Augmenting a finite automaton with a stack enables recognition and generation (via pushdown automata)

\[
\begin{align*}
S & \rightarrow xAy \mid zB \\
A & \rightarrow aA \mid t \\
B & \rightarrow bB \mid u
\end{align*}
\]

\[
\begin{align*}
S & \rightarrow xA \\
xaaty
\end{align*}
\]
Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition

- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

S → xAy | zB
A → aA | t
B → bB | u

xaaty
Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

\[
\begin{align*}
S & \rightarrow xAy \mid zB \\
A & \rightarrow aA \mid t \\
B & \rightarrow bB \mid u \\
\end{align*}
\]

\[\text{x\aataty}\]
Context Free Grammars & Pushdown Automata

- **Context free grammars** add recursion and enable Dyck language recognition

- Augmenting a finite automaton with a stack enables recognition and generation (via **pushdown automata**)

S → xAy | zB  
A → aA | t  
B → bB | u  

![Diagram of context-free grammar and automaton]

xaaty
Context Free Grammars & Pushdown Automata

- **Context free grammars** add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*).

\[
S \rightarrow xAy \mid zB
\]
\[
A \rightarrow aA \mid t
\]
\[
B \rightarrow bB \mid u
\]

\[
\begin{align*}
S & \rightarrow xA \\
A & \rightarrow aA \\
A & \rightarrow aA \\
B & \rightarrow t \\
A & \rightarrow u
\end{align*}
\]

xaaty
Context Free Grammars & Pushdown Automata

- **Context free grammars** add recursion and enable Dyck language recognition

- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

```
S → xAy | zB
A → aA | t
B → bB | u
```

```
xaaty
xaaty
```
Context Free Grammars & Pushdown Automata

- **Context free grammars** add recursion and enable Dyck language recognition

- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

```
S → xAy | zB
A → aA | t
B → bB | u
```

Diagram:

![Diagram of context-free grammar and pushdown automaton]
Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition

- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

\[
S \rightarrow xAy \mid zB \\
A \rightarrow aA \mid t \\
B \rightarrow bB \mid u
\]

S

[tbl]

<table>
<thead>
<tr>
<th>S</th>
<th>xA</th>
</tr>
</thead>
</table>

xaaty
Context Free Grammars & Pushdown Automata

- Context free grammars add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via pushdown automata)

S → xAy | zB
A → aA | t
B → bB | u

S xAa
xaat

Context Free Grammars & Pushdown Automata

- **Context free grammars** add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

```
S  →  xAy | zB
A  →  aA | t
B  →  bB | u
```

Is this behavior similar to something more familiar?
Context Free Grammars & Pushdown Automata

- **Context free grammars** add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via pushdown automata)
- Context free grammars play a key role in
  - Precise static program analysis
Context Free Grammars & Pushdown Automata

- **Context free grammars** add recursion and enable Dyck language recognition.
- Augmenting a finite automaton with a stack enables recognition and generation (via pushdown automata).
- **Context free grammars** play a key role in:
  - Precise static program analysis
  - Program synthesis
Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition.
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*).
- *Context free grammars* play a key role in
  - Precise static program analysis
  - Program synthesis
  - Prediction and machine learning on programs
Formal Logic

- Formal logic is a systematic approach to reasoning
  - Separate the messy content of an argument from its structure
Formal Logic

• Formal logic is a systematic approach to reasoning
  – Separate the messy content of an argument from its structure

• Sometimes the process can be automated
  – e.g. satisfiability problems, type inference, ...
Formal Logic

- Formal logic is a systematic approach to reasoning
  - Separate the messy content of an argument from its structure
- Sometimes the process can be automated
  - e.g. satisfiability problems, type inference, ...
- Program analysis has actually been one of the driving forces behind satisfiability in recent years.
Classical Logic

- You likely already know either *propositional* or *first order logic*
  - Systems for reasoning about the truth of sentences
Classical Logic

- You likely already know either *propositional* or *first order logic*
  - Systems for reasoning about the truth of sentences

- Atoms abstract away the actors of the sentences
  - Constants: #t, #f
  - Variables: x, y, z, ...
Classical Logic

- You likely already know either propositional or first order logic
  - Systems for reasoning about the truth of sentences
- Atoms abstract away the actors of the sentences
  - Constants: #t, #f
  - Variables: x, y, z, ...
- Connectives relate the atoms & other propositions to each other
  - ¬ (Not), ∧ (And), ∨ (or)
  - → (Implies), ↔ (Iff)
Classical Logic

- You likely already know either *propositional* or *first order logic*
  - Systems for reasoning about the truth of sentences

- Atoms abstract away the actors of the sentences
  - Constants: #t, #f
  - Variables: x, y, z, ...

- Connectives relate the atoms & other propositions to each other
  - → (Not), ∧ (And), ∨ (or)
  - → (Implies), ↔ (Iff)

\[ x \land \neg y \land z \]
Classical Logic

- First order logic augments with
Classical Logic

- First order logic augments with
  - Quantifiers- $\exists$ (there exists), $\forall$ (for all)
  - Functions & Relations- e.g. father(x), Elephant(y)
Classical Logic

- First order logic augments with
  - Quantifiers- ∃ (there exists), ∀ (for all)
  - Functions & Relations- e.g. father(x), Elephant(y)
- Sentences can be true or false
Classical Logic

- First order logic augments with
  - Quantifiers- $\exists$ (there exists), $\forall$ (for all)
  - Functions & Relations- e.g. $\text{father}(x)$, $\text{Elephant}(y)$
- Sentences can be true or false
  $\forall x (\text{Elephant}(x) \rightarrow \text{Grey}(x))$
Classical Logic

- First order logic augments with
  - Quantifiers- ∃ (there exists), ∀ (for all)
  - Functions & Relations- e.g. father(x), Elephant(y)
- Sentences can be true or false
  \[ \forall x (\text{Elephant}(x) \rightarrow \text{Grey}(x)) \]
  \[ \forall x (\text{Elephant}(x) \rightarrow \text{Elephant}(\text{father}(x))) \]
Classical Logic

- First order logic augments with
  - Quantifiers- \(\exists\) (there exists), \(\forall\) (for all)
  - Functions & Relations- e.g. father\((x)\), Elephant\((y)\)
- Sentences can be true or false
- An interpretation \(I\) of the world along with the rules of logic determine truth via judgement \(\vdash\)
Classical Logic

• First order logic augments with
  – Quantifiers- \( \exists \) (there exists), \( \forall \) (for all)
  – Functions & Relations- e.g. father(x), Elephant(y)

• Sentences can be true or false

• An interpretation \( I \) of the world along with the rules of logic determine truth via judgement (\( \vdash \))

\[
I \vdash x \text{ and } I \vdash y \text{ iff } I \vdash x \land y
\]
Classical Logic

- *Satisfiability*
  - A sentence $s$ is satisfiable $\leftrightarrow \exists I \ (I \vdash s)$
Classical Logic

- **Satisfiability**
  - A sentence $s$ is satisfiable $\iff \exists I (I \vdash s)$

- **Validity**
  - A sentence $s$ is valid $\iff \forall I (I \vdash s)$
Classical Logic

- **Satisfiability**
  - A sentence $s$ is satisfiable $\iff \exists I (I \vdash s)$

- **Validity**
  - A sentence $s$ is valid $\iff \forall I (I \vdash s)$

- We will see later how these can be used for a wide variety of tasks
Classical Logic

- **Satisfiability**
  - A sentence $s$ is satisfiable $\iff \exists I \ (I \vdash s)$

- **Validity**
  - A sentence $s$ is valid $\iff \forall I \ (I \vdash s)$

- We will see later how these can be used for a wide variety of tasks
  - Bug finding
  - Model checking (proving correctness)
  - Explaining defects
  - ...
Inference using classical logic

- Rules express how some judgements enable others

\[ \Gamma \vdash x \quad \Delta \vdash y \]

\[ \Gamma, \Delta \vdash x \land y \]
Inference using classical logic

- Rules express how some judgements enable others

\[ \Gamma \vdash x \quad \Delta \vdash y \]

\[ \Gamma, \Delta \vdash x \land y \]
Inference using classical logic

- Rules express how some judgements enable others

\[
\Gamma \vdash x \quad \Delta \vdash y
\]

\[
\Gamma, \Delta \vdash x \land y
\]
Inference using classical logic

- Rules express how some judgements enable others

\[ \Gamma \vdash x \quad \Delta \vdash y \]

\[ \Gamma, \Delta \vdash x \land y \]

- Proofs can be written by stacking rules
Inference using classical logic

- Rules express how some judgements enable others
  \[ \Gamma \vdash x \quad \Delta \vdash y \]
  \[ \Gamma, \Delta \vdash x \land y \]
- Proofs can be written by stacking rules

\[
\begin{align*}
A & \vdash A \\
A \rightarrow B & \vdash A \rightarrow B \\
A & \vdash A \\
A \rightarrow B, A & \vdash B \\
\rightarrow \text{-E} \\
A, A \rightarrow B, A & \vdash A \times B \\
\times \text{-I} \\
A \rightarrow B, A, A & \vdash A \times B \\
\end{align*}
\]

Hoare Logic

- *Hoare logic* reasons about the behavior of programs and program fragments
Hoare Logic

- *Hoare logic* reasons about the behavior of programs and program fragments.

\[
\{ \varphi \} C \{ \psi \}
\]

Precondition
Hoare Logic

- **Hoare logic** reasons about the behavior of programs and program fragments. 

$$\{ \varphi \} C \{ \psi \}$$

- **Precondition**
- **Command**
Hoare Logic

- **Hoare logic** reasons about the behavior of programs and program fragments

\[
\{ \varphi \} C \{ \psi \}
\]

- Precondition
- Command
- Postcondition
Hoare Logic

• *Hoare logic* reasons about the behavior of programs and program fragments

\[ \{ \varphi \} C \{ \psi \} \]

• If phi holds before C, psi will hold after

\[ \{x=3 \land y=2\} x = 5 \{x=5\} \]
Hoare Logic

- *Hoare logic* reasons about the behavior of programs and program fragments

\[
\{ \varphi \} C \{ \psi \}
\]

- If phi holds before C, psi will hold after

\[
\{ x=3 \land y=2 \} x = 5 \{ x=5 \}
\]

- A *weakest precondition* \(wp(C, \psi)\) captures all states leading to \(\psi\) after \(C\).
Hoare Logic

- *Hoare logic* reasons about the behavior of programs and program fragments

\[ \{ \phi \} C \{ \psi \} \]

- If phi holds before C, psi will hold after

\[ \{x=3 \land y=2\} x = 5 \{x=5\} \]

- A weakest precondition \( \text{wp}(C, \psi) \) captures all states leading to \( \psi \) after \( C \).

\[ \{\#t\} x \leftarrow 5 \{x=5\} \]
Hoare Logic

- *Hoare logic* reasons about the behavior of programs and program fragments

\[ \{ \phi \} C \{ \psi \} \]

- If phi holds before C, psi will hold after

\[ \{ x=3 \land y=2 \} x = 5 \{ x=5 \} \]

- A weakest precondition \( \text{wp}(C, \psi) \) captures all states leading to \( \psi \) after \( C \).

\[ \{ \#t \} x \leftarrow 5 \{ x=5 \} \]

\[ ??? \{ ???? \} \text{if c then } x \leftarrow 5 \{ x=5 \} \]
Intuitionistic & Constructive Logic

- It can be useful to modify or limit rules of inference
Intuitionistic & Constructive Logic

- It can be useful to modify or limit rules of inference
  - Suppose a compiler cannot prove variable $x$ is an int. Is it reasonable for the compiler to assume $x$ is a string?
It can be useful to modify or limit rules of inference

- Suppose a compiler cannot prove variable $x$ is an int. Is it reasonable for the compiler to assume $x$ is a string?

**Constructivism** argues that truth comes from direct evidence.

- We cannot merely assume $p$ or not $p$, we must have evidence.
Intuitionistic & Constructive Logic

- It can be useful to modify or limit rules of inference
  - Suppose a compiler cannot prove variable x is an int. Is it reasonable for the compile to assume x is a string?
- Constructivism argues that truth comes from direct evidence.
  - We cannot merely assume p or not p, we must have evidence
- *Intuitionistic logic* restricts the rules of inference to require direct evidence
Intuitionistic & Constructive Logic

- Classic logic includes several rules including

\[ \vdash p \lor \neg p \]

Law of excluded middle
Intuitionistic & Constructive Logic

- Classic logic includes several rules including:

  \[ \Gamma \vdash p \lor \neg p \]

  \[ \Gamma \vdash \neg \neg p \]

  \[ \Gamma \vdash p \]

  Double negation elimination
Intuitionistic & Constructive Logic

- Classic logic includes several rules including:

\[ \Gamma \vdash \neg \neg p \]

\[ \Gamma \vdash p \]

- Intuitionistic logic excludes these to require direct evidence
Intuitionistic & Constructive Logic

- Classic logic includes several rules including

\[ \Gamma \vdash \neg \neg p \]
\[ \Gamma \vdash p \]

- Intuitionistic logic excludes these to require direct evidence

- Note, this is commonly used in type systems
Linear & Substructural Logic

\[
sellsBurr{	ext{ritos}}(store) \quad \text{has 10 Dollars}(me) \quad \vdash \quad buyBurr{	ext{rito}}(me, store)
\]
\[\text{sellsBurritos}(\text{store}) \quad \text{has10Dollars}(\text{me}) \quad \Rightarrow \quad \text{buyBurrito}(\text{me}, \text{store}) \quad \land \quad \text{buyBurrito}(\text{me}, \text{store})\]
sellsBurritos(store) \ \& \ \text{has10Dollars}(me) \ \vdash \ \text{buyBurrito}(me,store) \ \& \ \text{buyBurrito}(me,store) \ \& \ \text{buyBurrito}(me,store)
sellsBurritos(store) \land has10Dollars(me) \models buyBurrito(me,store) \land buyBurrito(me,store) \land buyBurrito(me,store) \land buyBurrito(me,store)
sellsBurritos(store) \implies \text{buyBurrito}(me, store) \\
\land \text{buyBurrito}(me, store) \\
\land \text{buyBurrito}(me, store) \\
\land \text{buyBurrito}(me, store)

Classical & intuitionistic logic have trouble expressing consumable facts
sellsBurritos(store) ⊢ buyBurrito(me,store)

• Linear logic denotes separates facts into two kinds
  – [Intuitionistic] as before
  – <Linear> cannot be used with contraction or weakening
Linear & Substructural Logic

sellsBurritos(store) ⊨ buyBurrito(me,store)

- Linear logic denotes separates facts into two kinds
  - [Intuitionistic] as before
  - <Linear> cannot be used with contraction or weakening

\[
\begin{align*}
\Gamma, A, A, \Delta & \vdash p \\
\hline
\Gamma, A, \Delta & \vdash p \\
\end{align*}
\]

\[
\begin{align*}
\Gamma, \Delta & \vdash p \\
\hline
\Gamma, A, \Delta & \vdash p \\
\end{align*}
\]
Linear & Substructural Logic

\[
\text{sellsBurritos}(\text{store}) \quad \vdash \quad \text{buyBurrito}(\text{me}, \text{store})
\]

- Linear logic denotes separates facts into two kinds
  - [Intuitionistic] as before
  - \(<\text{Linear}>\) cannot be used with contraction or weakening
  - In essence, linear facts must be consumed \textit{exactly once} in a proof.
Linear & Substructural Logic

sellsBurritos(store) ⊢ buyBurrito(me, store)

- Linear logic denotes separates facts into two kinds
  - [Intuitionistic] as before
  - \(<\text{Linear}>\) cannot be used with contraction or weakening
  - In essence, linear facts must be consumed exactly once in a proof.

Logics that remove additional rules from intuitionistic logic are *substructural*
Linear & Substructural Logic

\[
\text{sellsBurritos}(\text{store}) \quad \quad \text{has10Dollars}(\text{me}) \quad \quad \text{\textit{\smallrightarrow}} \quad \text{buyBurrito}(\text{me,store})
\]

- Linear logic denotes separates facts into two kinds
  - [Intuitionistic] as before
  - \texttt{<Linear>} cannot be used with contraction or weakening
  - In essence, linear facts must be consumed exactly once in a proof.

- This forms the backbone of *ownership types* in languages like Rust!
Separation Logic

- Linear logic allows facts to be used exactly once $\leftrightarrow$ or arbitrarily many times $[]$. 
Separation Logic

- Linear logic allows facts to be used exactly once $<>$ or arbitrarily many times $[]$.
- *Separation logic* (informally) distinguishes separate facts (counting), allowing them to be used separately.
Separation Logic

• Linear logic allows facts to be used exactly once $<>$ or arbitrarily many times $[]$.

• *Separation logic* (informally) distinguishes separate facts (counting), allowing them to be used separately.

• This allows compositional reasoning about software.

$$\{x\mapsto y \ast y\mapsto x\}x = z\{x\mapsto z \ast y\mapsto x\}$$
Separation Logic

- Linear logic allows facts to be used exactly once <> or arbitrarily many times [].
- *Separation logic* (informally) distinguishes separate facts (counting), allowing them to be used separately.
- This allows compositional reasoning about software.

\[
\{x \mapsto y * y \mapsto x\}x = z\{x \mapsto z * y \mapsto x\}
\]

Suppose we used \(\land\) instead, what problem exists?
Separation Logic

- Linear logic allows facts to be used exactly once <> or arbitrarily many times [].

- *Separation logic* (informally) distinguishes separate facts (counting), allowing them to be used separately.

- This allows compositional reasoning about software.

\[
\{x \mapsto y * y \mapsto x\} x = z \{x \mapsto z * y \mapsto x\}
\]

- Separation logic enables efficient compositional reasoning
  - It is the backbone of Facebook’s Infer engine!
Recap

• Formalism is a tool that can simplify reasoning about tasks
Recap

- Formalism is a tool that can simplify reasoning about tasks
- Many solutions involve a careful combination of
  - order theory (for comparison)
  - formal grammars (for structure)
  - formal logic (for inference)