Course Website:

https://kmicinski.com/automated-reasoning

Modern Symbolic AI & Automated Reasoning

Instructor: Kris Micinski

Questions to start us off:

- What **is** reasoning?
  - How can we rigorously define it
  - Can we systematically understand, represent, and implement reasoning procedures?

Example Narrative

Handsome is a dog
Handsome lives in Syracuse
All dogs which live in Syracuse must be vaccinated
Therefore, Handsome must be vaccinated

How can we formalize this?

Handsome is a dog

`dog(andy).`

Handsome lives in Syracuse

`lives(andy, syracuse).`

All dogs which live in Syracuse must be vaccinated

$\forall x.\ dog(x) \wedge lives(x, syracuse) \Longrightarrow requires\_vaccination(x)$

Therefore, Handsome must be vaccinated:

Instantiate the quantified formula with `handsome`:

```
dog(andy) ∧ lives(andy, syracuse)
  ⟹ requires_vaccination(x)
```

Both of the antecedents (things before arrow) are true, thus:

`requires_vaccination(andy)`

Question: how could a **computer** represent this proof?

First, we need to ask ourselves: what's the **logic**?
Broadly, logics formalize (i.e., specify algebraic representation of) formulas, truths, and proofs.

- Our last example used **first-order** logic (FOL) over *atoms*
  - FOL allows universal (∀) and existential (∃) quantifiers
  - Can also include function symbols: $\forall x.\ x > 1 \implies x > 0$
- But also **propositional** logic (no quantifiers)
- Even things like **temporal** logics, which include quantifiers to say things like "in the **future**, the stock price will be higher than it is now."

Now that we picked first-order logic w/ quantifiers, we need to understand how to represent a **proof** of our supposition

For example, we might write a list of valid statements, each either (a) assumptions or (b) statements following from all higher-up statements:

```
(A₀) dog(andy).
(A₁) lives(andy,syracuse).
(A₂) ∀x. dog(x) ∧ lives(x,syracuse)
         ⟹ requires_vaccination(x)
(Instantiate A₂, [x ↦ handsome])
dog(andy) ∧ lives(andy,syracuse)
         ⟹ requires_vaccination(andy)
(⟹, A₀ A₁) requires_vaccination(andy)
```

This style of proof uses a **sequent calculus** formulation

Each line follows from (is conditional upon) *all previous lines*

There are a variety of sequent-style calculi for various logics

```
(A₀) dog(andy).
(A₁) lives(andy,syracuse).
(A₂) ∀x. dog(x) ∧ lives(x,syracuse)
         ⟹ requires_vaccination(x)
(Instantiate A₂, [x ↦ andy])
dog(andy) ∧ lives(andy,syracuse)
         ⟹ requires_vaccination(andy)
(⟹, A₀ A₁) requires_vaccination(andy)
```

The precise formalization of this matters a lot, in terms of reasoning about expressivity, correctness, and completeness (can everything true be proven?) for a given logical system.

In this class, we will detail these philosophical issues, but largely in the context of understanding their impacts on building programs which perform automated reasoning.

E.g., propositional logic is easy (enumerable), but first-order logic (quantifier instantiation) is harder and, in general, requires symbolic search.

We will ask questions such as:

- How to represent (proofs of) knowledge symbolically?
- How can we build proof checkers, which increase our confidence the proof (system) is meaningful?
- How do you efficiently search for proofs of true statements (or refutations of false statements)?

We will also cover rigorous formal systems necessary to understand these, to the degree necessary to understand the correct design of automated reasoning systems.

Some Tools we will cover

P1 — SAT Solvers (MiniSat)

P2 — Query languages and Datalog (e.g., Soufflé)

P3 — Constraint solvers and Satisfiability-Modulo Theory Solvers (e.g., Z3, CVC5, etc…)

P4 — Interactive Theorem Provers (Lean)

Projects (in Python)

P1 — Reachability-Based Verification
P2 — Bounded Model Checking w/ SAT
P3 — Symbolic execution with Z3
P4 — Interactive Theorem Provers (Lean)

We use writing to help ourselves structure our thoughts—revising, editing, restarting along the way

This class examines the process of writing and understand programs using a *systematic, iterative approach*

Want to learn "how to think" about programming

Propositional Logic and SAT

CIS700 — Fall 2023

Kris Micinski

Today we'll look at our first logic: propositional logic

Propositional logic consists of formulas built via connectives applied to **atomic propositions**

The following are propositional formulas:

P ("P holds"), every atomic proposition is trivially a formula

$P \wedge (Q \vee \neg P)$ ("P holds and (Q or P) also holds")

$\neg P \wedge Q \implies Q \wedge \neg P$ ("Not P and Q implies Q and not P")

True ($\top$) and False ($\bot$), but these symbols have many meanings

Let’s consider a universe of four propositional variables:
DoorOpen, DoorClosed, MachineOn, MachineOff

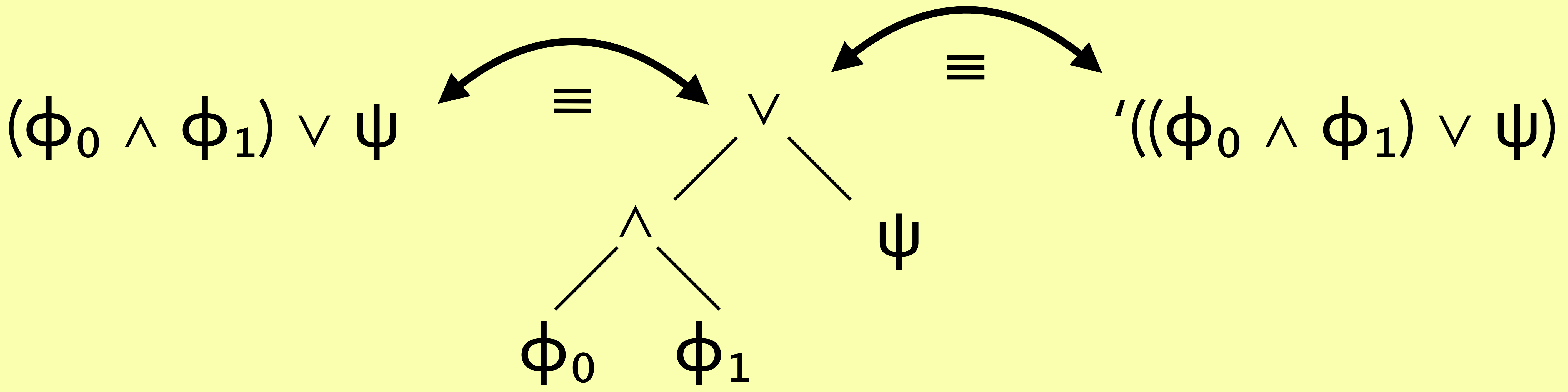
How would you express the following:
“The machine may not be both on and off.”
(Notice that this is xor, even though we excluded it…)

“If the door is open, the machine may not be on.”

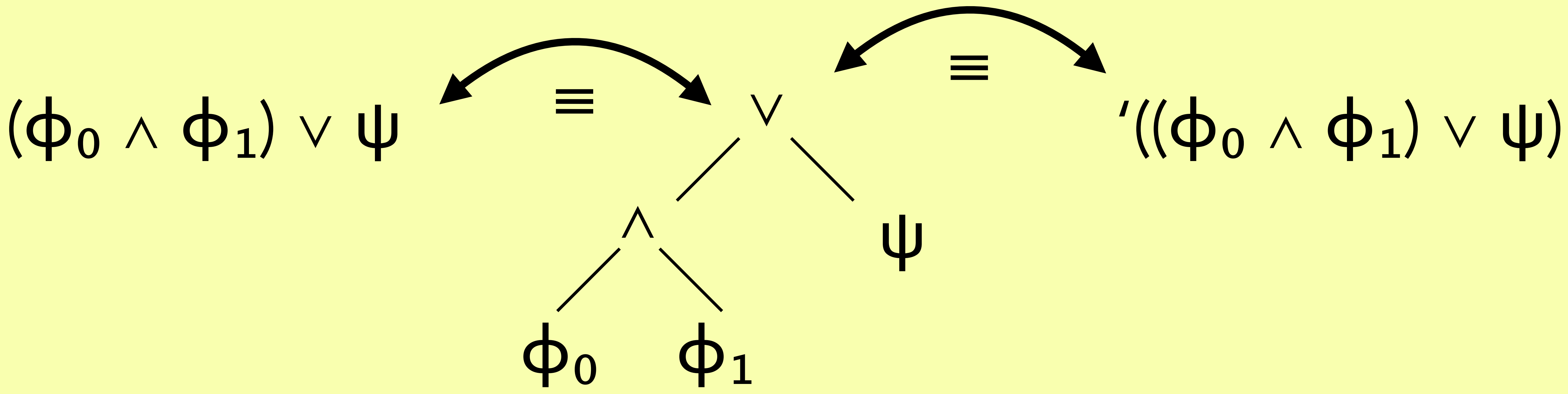
Let's consider a universe of four propositional variables:
DoorOpen, DoorClosed, MachineOn, MachineOff

How would you express the following:
"The machine may not be both on and off."
(Notice that this is xor, even though we excluded it…)
$\neg(\text{MachineOn} \wedge \text{MachineOff})$
"If the door is open, the machine may not be on."
$\text{Open} \Rightarrow \neg\text{MachineOn}$

Propositional formulas are (structurally) recursive structures. All formulas have an implicit recursive structure with constants / propositional variables at their leaves



Scheme's **S-expressions** (structured expressions) systematize this notion. S-expressions are symbolic representations, implemented under the hood via pointers to subtrees



We formalize our expressions as a Racket **predicate**

```
(define (prop-formula? ϕ)
  (match ϕ
    [(? symbol? x) #t]
    ['T #t] ;; for "true"
    ['F #t] ;; for "false"
    [`(¬ ,ϕ) #t]
    [`(,ϕ0 ∧ ,ϕ1) #t]
    [`(,ϕ0 ∨ ,ϕ1) #t]
    [`(,ϕ0 ⇒ ,ϕ1) #t]
    [`(,ϕ0 ⇔ ,ϕ1) #t]
    [_ #f]))
```

We formalize our expressions as a Racket **predicate**

```
(define (prop-formula? ϕ)
  (match ϕ
    [(? symbol? x) #t]
    ['T #t] ;; for "true"
    ['F #t] ;; for "false"
    [`(¬ ,ϕ) #t]
    [`(,ϕ0 ∧ ,ϕ1) #t]      ;; Test…
    [`(,ϕ0 ∨ ,ϕ1) #t]      (prop-formula?
    [`(,ϕ0 ⇒ ,ϕ1) #t]        '(ϕ0∧ ϕ1) ∧ ψ))
    [`(,ϕ0 ⇔ ,ϕ1) #t]      ;; #t
    [_ #f]))
```

Interpretations

A formula is just a *statement*. To speak of a statement's veracity, we need to rigorously define the notion of a "true" statement.

There are differing perspectives on this:

- A statement is true precisely when I can materialize a symbolic proof for it
  - This is the **constructive** view, every statement demands *evidence* represented as data
- Everything is either true or false, the **classical** view
  - $\neg\neg P \implies P$ (**excluded middle**)

We will start by looking at a classical, model-theoretic interpretation in which interpretations are mappings of variables to booleans.

Later, we will consider the importance of *proof theory*, which deals with proofs as materialized objects which may be used to formally derive knowledge.

These two perspectives are different sides of the same coin, but it is important to be mindful of their differences

Model Theoretic Interpretations

Propositional logic has a simple notion for classical interpretations: sets. If there are a finite number of atoms under consideration (say $\mathscr{A}$), the finite sets are enumerable and every subset I of $\mathscr{P}(\mathscr{A})$ forms a partition, such that atoms in I are considered "true" and atoms not in I are "false."

Example: $\mathscr{A}$ = {On, Off}, $\mathscr{P}(\mathscr{A})$ = {{}, {On}, {Off}, {On, Off}}
Each set in the power set is an *interpretation*
We apply interpretations to formulas to determine their truth

From Interpretations to Valuaations

Given an interpretation (i.e., set of atoms which are to be construed as “true”), we can recursively define veracity

Exercise:

Say I = {On, Off}. What should the truth value (“true” or “false”) be for each of these formulas ϕ when…

ϕ is “On”

ϕ is “On $\wedge$ $\neg$Off”

ϕ is “On $\implies$ Off”

ϕ is “On $\wedge$ ($\neg$On $\vee$ Off)”

Interpretations in code

In Scheme (Racket), we can represent an interpretation as a dictionary (hash):

```
(define (interpretation? I)
  (and (hash? I)
       (andmap symbol? (hash-keys I))
       (andmap boolean? (hash-values I))))
```

These hashes are implemented efficiently via a data structure which allows *persistent* ~O(1) lookup / insertion

This is a variation of the **mutable** hash tables generally shown in intro DS classes, consider looking up HAMT

These hashes are implemented efficiently via a data structure which allows *persistent* ~O(1) lookup / insertion

This is a variation of the **mutable** hash tables generally shown in intro DS classes, consider looking up HAMT

```
(hash-ref (hash-set (hash 'x 3) 'x 5) 'x)
```

We can now define `(interp-formula ϕ I)`, i.e., $[\![\phi]\!]$ which considers several different cases based on the structure of ϕ:

- If ϕ is literally true or false, the answer is #t / #f (Racket's "true")
- If ϕ has the form $\phi_0 \oplus \phi_1$, then $[\![\phi]\!]$ is ($\oplus$ $[\![\phi_0]\!]$ $[\![\phi_1]\!]$)
  - Where we assume there is a Racket version of $\oplus$
- If ϕ has the form $\neg\phi$ then $[\![\phi]\!]$ is (`not` $[\![\phi]\!]$)

When the interpretation I results in ϕ being true, we say that I **satisfies** ϕ, often written using the notation $I \vDash \phi$.

In this case, we will call I a "model" of the propositional statement ϕ.

Propositional logic is *decidable*: the set of possible interpretations is finite (assuming formula size is finite) and you can check each interpretation

A formula is **valid** if it is true in *every* interpretation: $\models \phi$, notice that there is nothing to the left of $\models$, suggesting that *every* I will suffice to satisfy ϕ

These statements are called *tautologies*. Which of the following are tautologies?

(I) $\neg P \vee P \vee (\neg P \wedge P)$, (II) $P \implies P \wedge Q$, (III) $P \wedge Q \implies P$

Proof Theory

Logics are described in different ways. We have seen a model theoretic description of propositional logic, which appeals to a semantics (formalized as interpretations).

By contrast, **proof theory** describes *syntactic* objects (proofs) which represent valid *derivations* of new knowledge from old knowledge

We will study several proof systems throughout the course: natural deduction, resolution, sequent calculus, and analytic tableaux

Normal Forms

Working with arbitrarily-structured formulas means that we need to deal with a bunch of different specific forms ($\vee$, $\wedge$, $\Longrightarrow$, $\Leftrightarrow$, …)

This is messy, since there is a much smaller basis. It is possible to encode all formulas in terms of just $\neg$ and $\wedge$, for example.

There are a variety of **normal forms** for propositional logic, i.e., syntactic restrictions on formulas which do not inhibit expressivity

There are **conversion** algorithms into these normal forms

NNF, CNF, and DNF

Negation Normal Form (NNF): All negations have been pushed as far down as possible. $\neg(P \land Q)$ ❌, $(\neg P \lor \neg Q)$ ✅

Conversion algorithm: first expand $P \Longrightarrow Q$ into $\neg P \lor Q$ (etc…), next repeatedly apply De-Morgan's laws and cancel $\neg\neg P$ to P

Conjunctive Normal Form (CNF): A big conjunction of disjunctions ("clauses"): $(A \lor \neg B \lor C) \land (B \lor \neg C) \land (\neg A \lor \neg B)$

To obtain: Start with DNF, then distribute over $\land$

Disjunctive Normal Form (DNF): A big disjunction of conjunctions: $(A \land B \land C) \lor (A \land \neg B \land \neg C) \lor \ldots$

Let's convert the following to NNF:

$(A \land (B \Leftrightarrow A)) \implies B$

$\neg(A \land (B \Leftrightarrow A)) \lor B$ (encoding $\implies$)

$\neg(A \land ((B \implies A) \land (A \implies B))) \lor B$, (encoding $\Leftrightarrow$)

$\underline{\neg(A \land ((\neg B \lor A) \land (\neg A \lor B)))} \lor B$, (encoding $\implies$) Now De Morgan's

$A \lor \underline{\neg((\neg B \lor A) \land (\neg A \lor B)))} \lor B$, continue De Morgan's

$A \lor (\underline{\neg(\neg B \lor A)} \lor \underline{\neg(\neg A \lor B))}) \lor B$, and continue…

$A \lor ((\underline{\neg\neg B} \land \neg A) \lor (\underline{\neg\neg A} \land \neg B))) \lor B$, now cancel $\neg\neg$

$A \lor (B \land \neg A) \lor (A \land \neg B) \lor B$ (NNF, also in DNF)

Starting from here, how do we get to CNF? Naively: use distributivity,

$A \vee (B \wedge \neg A) \vee (A \wedge \neg B) \vee B$

Group everything as binary so that we can distribute:

$(B \wedge \neg A) \vee (A \vee ((A \wedge \neg B) \vee B)))$

Exercise: use distributivity, cancel double negation, to achieve CNF

$(B \vee (A \vee ((A \wedge \neg B) \vee B))) \wedge (\neg A \vee (A \vee ((A \wedge \neg B) \vee B))$, and then…

$(B \vee (A \vee ((A \vee B) \wedge (\neg B \vee B))) \wedge (\neg A \vee (A \vee ((A \wedge \neg B) \vee B))$,

$(B \vee (A \vee ((A \vee B) \wedge (\neg B \vee B))) \wedge (\neg A \vee (A \vee ((A \vee B) \wedge (\neg B \vee B))))$,

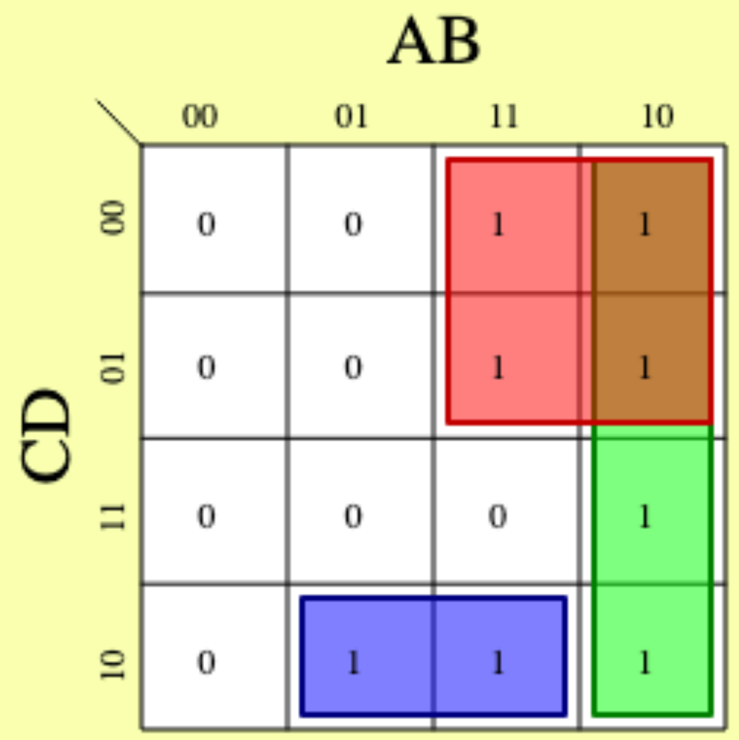
…

$(A \vee B)$

CNF vs. DNF, Digital Logic

In general, we will work with CNF, because it allows a more "dense" representation of formulas. Translation into DNF often results in large (super linear) encoding overhead. Hence, modern solvers often consume input in CNF format.

You may recall this material from a digital logic class—there is serious overlap with minterms/ maxterms/karnaugh maps, worth looking into



Tseitin’s transformation

The distributivity law induces super-linear encoding blowup for formulas—this leads to slowdown of tools which check satisfi ability / validity of these formulas

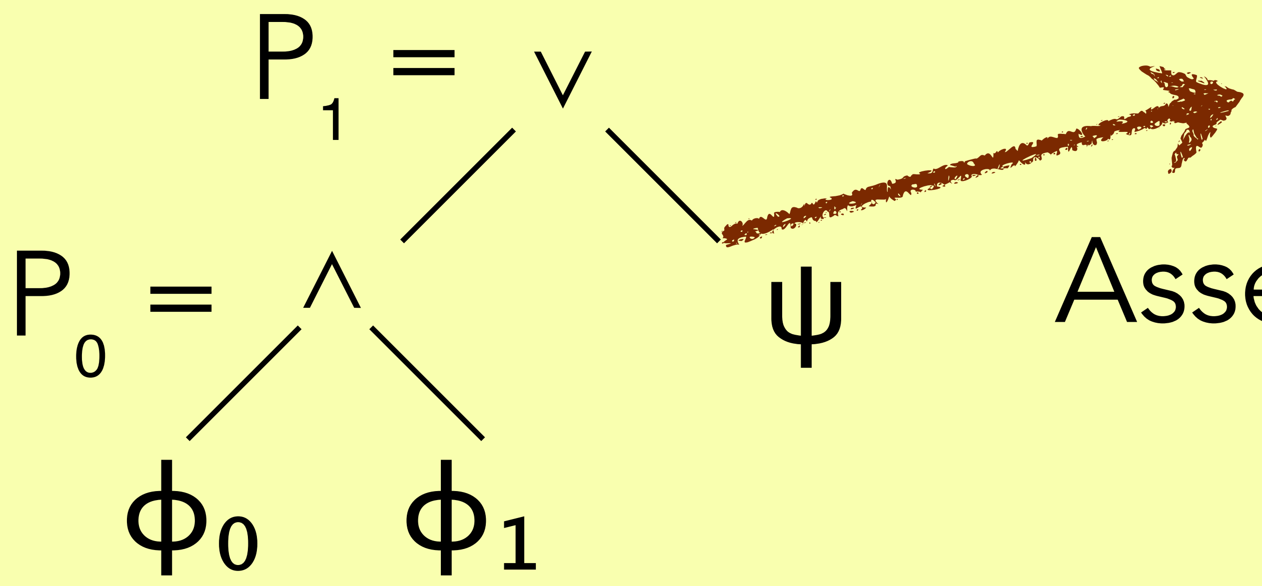
We often want a better transformation from formulas into CNF

A popular transformation is Tseitin’s transformation. Intuitively, it assigns each sub-formula a new propositional atom and asserts a suitable bi-implication

Tseitin’s transformation

- Assign non-literal formulas a new variable
- Add definitions for each, recursively

$$(\phi_0 \wedge \phi_1) \vee \psi$$



$$P_1 \Leftrightarrow P_0 \vee \psi$$

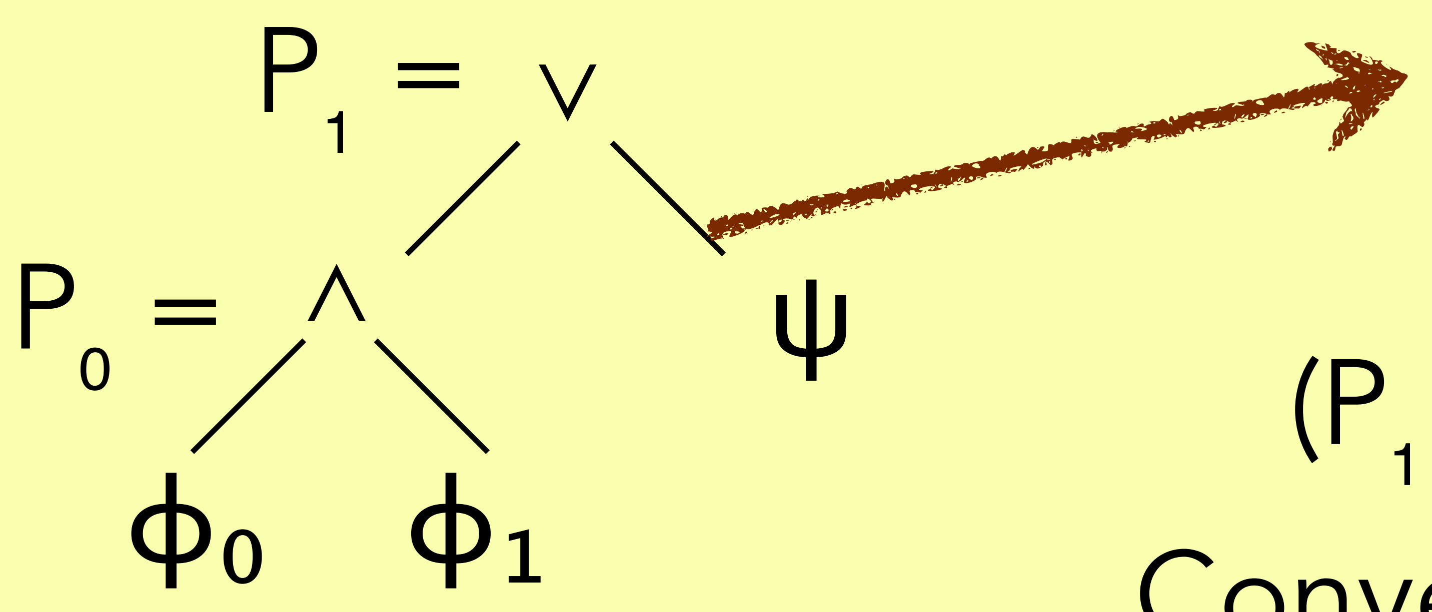
$$P_0 \Leftrightarrow \phi_0 \wedge \phi_1$$

Assert each definition, plus P_1,
via a big conjunction:

$$(P_1 \Leftrightarrow P_0 \vee \psi) \wedge (P_0 \Leftrightarrow \phi_0 \wedge \phi_1) \wedge P_1$$

Tseitin's transformation

$(\phi_0 \wedge \phi_1) \vee \psi$



$P_1 \Leftrightarrow P_0 \vee \psi$

$P_0 \Leftrightarrow \phi_0 \wedge \phi_1$

Which forms…

$(P_1 \Leftrightarrow P_0 \vee \psi) \wedge (P_0 \Leftrightarrow \phi_0 \wedge \phi_1) \wedge P_1$

Convert each definition to CNF:

The first … $(\neg P_1 \vee P_0 \vee \Psi) \wedge (\neg P_0 \vee P_1) \wedge (\neg \Psi \vee P_1)$

Then… $(\neg P_0 \vee \phi_0) \wedge (\neg P_0 \vee \phi_1) \wedge (\neg \phi_0 \vee P_0) \wedge (\neg \phi_1 \vee P_0)$

Why Tseitin’s transformation?

Tseitin’s transformation converts to CNF without the super linear blowup of naive distributivity

Output is a set of definitions (defined via bi-implication), where the set size is linear in the size of the input formula (one new var for each node in the formula)

Each of these definitions is of constant depth (either an atom or an application of a connective to atoms; then apply distributivity to obtain CNF (constant factor blowup)

SAT Solving

Checking whether a formula is satisfiable is called SAT(isfiability) solving. Propositional logic is decidable, via truth tables—but what is the complexity of checking via truth tables? Answer: O(2^n)

Does a better solution exist? In practice yes, but only for formulas which obey “typical” structure

Validity solving via SAT?

Notice that whenever ϕ is a validity (i.e., $\models \phi$), $\neg\phi$ is UNSAT

To see why: consider $\neg\phi$ is SAT, then it has some model I such that $I \models \neg\phi$. By assumption ϕ is a validity, so $I \models \phi$ as well. This gives us a contradiction, since $I \models \neg\phi \land \phi$

k-SAT

To simplify the core machinery of SAT solving, most solvers operate over CNF, a SAT instance given in CNF is k-SAT if the largest clause consists of k literals

Popular instances include 2-SAT and 3-SAT

2-SAT: $(P \lor \neg Q) \land (\neg P \lor Q) \land (\neg Q \lor \neg P)$

3-SAT: $(\neg R \lor P \lor \neg Q) \land (\neg P \lor Q \lor R) \land \ldots$

2-SAT is polynomial time (complete for NL)

2-SAT is a restricted form of k-SAT. Solving 2-SAT can be done in polynomial time and 2-SAT is complete for the class NL (Nondeterministic Logarithm, i.e., Turing machine w/ log memory space)

Algorithm: Start by building a graph of each literal in the program, then for each clause $\phi_0 \vee \phi_1$ add an edge between $\neg\phi_0$ and ϕ_1 and $\neg\phi_1$ and ϕ_0, obtaining a graph G

Last, calculate strongly-connected components of G and check if there is any P such that P and ¬ are in the same SCC

2-SAT Algorithm: Intuition

$$(P \vee \neg Q) \wedge (\neg P \vee Q) \wedge (\neg Q \vee \neg P)$$

Algorithm intuition: every clause has the form $\phi_0 \vee \phi_1$: if $\neg\phi_1$ holds, then the only thing that can make the clause true (and it *must* be true in any satisfying assignment!) will be ϕ_0. Thus, $\neg\phi_1$ "forces" ϕ_0 (unit propagation).

Knowing this, we (a) assemble a "forced-implication" graph and then (b) find its strongly-connected components, finally (c) checking if there are any literals P such that P and ¬P are in the same SCC

Solving 2-SAT via forced implication

$(P \vee \neg Q) \wedge (\neg P \vee Q) \wedge (\neg Q \vee \neg P)$

First, build a graph of literals…

P $\qquad\qquad$ $\neg P$

Q $\qquad\qquad$ $\neg Q$

Building forced implication graph

$$\underline{(P \vee \neg Q)} \wedge (\neg P \vee Q) \wedge (\neg Q \vee \neg P)$$

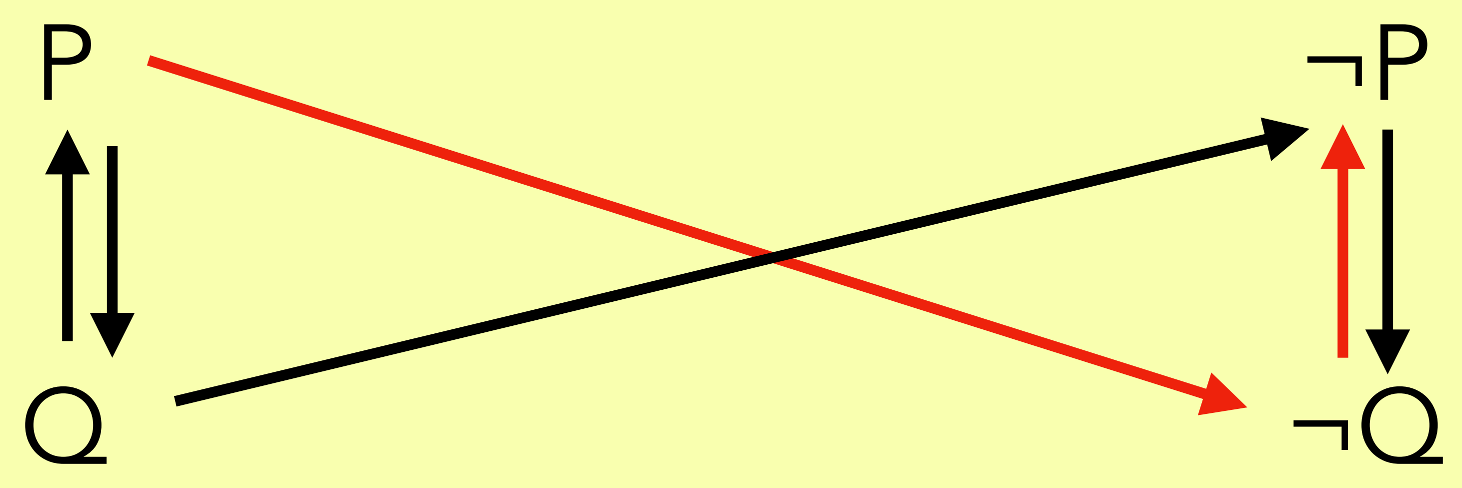
Now, add edges for each clause—start w/ left one…
(Stay in NNF: cancel ¬¬Q to Q!)

$$\begin{tikzcd} P & & \neg P \arrow[d] \\ Q \arrow[u] & & \neg Q \end{tikzcd}$$

Calculate SCCs and check

$\underline{(P \vee \neg Q)} \wedge (\neg P \vee Q) \wedge (\neg Q \vee \neg P)$

Now, add edges for each clause—start w/ left one…
(Stay in NNF: cancel $\neg\neg Q$ to Q!)



This is one big SCC—thus, the formula is unsatisfiable

(E.g., see the path from P to ¬P through ¬Q!)

Resolution

This intuition is closely related to an inference rule named **resolution**, which is stated as follows:

$$\frac{P \lor Q \qquad \neg Q \lor R}{P \lor R}$$

Here we use the natural deduction style (we'll discuss soon); if everything above the line holds, the thing below holds too

"If we know $P \lor Q$ and we know $\neg Q \lor R$, then we know $P \lor R$"

Refutation

We can use resolution to derive that a collection of clauses *cannot* hold simultaneously

Notice that the following cannot hold together:

$$\frac{\dfrac{\dfrac{{}^{1}P \vee \neg Q \qquad {}^{2}Q \qquad {}^{3}\neg P \vee \neg R \qquad {}^{4}\neg R}{{}^{5}P^{1,2}}}{{}^{6}\neg P^{1,2}}}{\bot}$$

Refutation

Refutation can be used to prove validities: if you want to prove ϕ, build a CNF version of $\neg\phi$, then derive $\bot$

Example: say we want to prove $P \implies P \vee Q$

First we build

$\neg(P \implies P \vee Q)$

$= \neg(\neg P \vee (P \vee Q))$

$= \neg\neg P \wedge \neg(P \vee Q)$

$= P \wedge \neg P \wedge Q$

Resolution on first two (single-literal) clauses gives us $\bot$

$$\frac{P \qquad \neg P}{\bot}$$

Refutation in practice

Refutation offers a general strategy to prove validities that is (potentially) much cheaper than enumerating truth tables

Resolution-based solving is quite popular, in a variety of logics. Examples we will see include DPLL and SLD resolution

However, real problem instances may involve (hundreds of) thousands of clauses—resolution generates new clauses and may not be productive, and there is no easy way to check in advance whether a resolution will be “worth it.”

3-SAT (and all k > 2) is NP complete

2-SAT is polynomial time because SCC computation can be done in polynomial time—we will soon see another restricted form of logic (Datalog) which shares similar decidability properties

3-SAT is much harder—3-SAT is one of Karp's 21 reductions. E.g., reduce 3-coloring to SAT, various reductions exist

MiniSAT

Very small, well-written SAT solver—utilizes a combination of approaches which we will discuss later

Accepts input in DIMACS format:

- comment lines begin `c My comment here`
- Problem line begins `p cnf V C` for V variables and C clauses
- Rest of the file is clauses, written as whitespace-separated sequences of integers: N is a positive occurrence, -N is a negative (negated) occurrence

DIMACS Exercise

Convert the following CNF formula to DIMACS input:

$(P \lor \neg Q \lor \neg R) \land (Q \lor R \lor \neg P) \land (\neg R \lor \neg Q \lor \neg P)$

```
c Solution…
c P=1, Q=2, R=3
p cnf 3 3
1 -2 -3 0
2 3 -1 0
-3 -2 -1 0
```

Limits of Propositional Logic; Looking Ahead

- The lack of quantifiers or structured values is a significant expressivity issue. Propositional logic lacks a notion of functions or their application
- What about statements like: $\forall$ x $\forall$ y. y > 0 $\Longrightarrow$ x+y > x
- These are **first order**, and require formalizing a domain of discourse (things quantifiers may bind), function symbols, …
- FOL is *much* more powerful, but commensurately more challenging to formalize

Questions and exercises

- Q0 (easy): What is the difference between constructive logic and classical logic? (We will learn more about constructive logic soon!)
- Q1 (harder): Show that $\neg$ and $\wedge$ form a minimal basis for propositional logic. Do this by expressing the other operators in terms of these
- Q2 (harder): Do $\neg$ and $\vee$ **also** form a minimal basis for propositional logic? Any others…?

Questions and exercises

- Put this into NNF:

  $$\neg(P \wedge \neg(Q \vee P)) \wedge (P \implies Q)$$

- Take the output of that question, now further convert to CNF and DNF
- Use Tseitin's transformation on the above formula to convert it to CNF

Questions and exercises

- (Optional) Download and install MiniSAT; read about the DIMACS input format and translate the examples from today's class

Designing an Adder

- Model bits via logical propositions. Consider a two-element bitvector of two bits: B0 and B1
- Design a two-bit adder, which takes two inputs, B10/B00 and B11/B01, and produces three outputs: the sum bits S0/S1 and a carry C

Are not and or a suitable basis for propositional logic? Which other operators form a minimal basis

No, "not" and "or" alone are not suitable as a minimal basis for propositional logic. A minimal basis for propositional logic typically consists of operators that can express all other logical operations. The most common minimal basis for propositional logic consists of the operators "not" (also known as negation) and "and" (also known as conjunction), which can be used to derive all other logical operations.

With just "not" and "and," you can construct the other logical operators as follows:

1. Implication (→): p → q can be expressed as ¬p ∨ q (not p or q).
2. Exclusive or (⊕): p ⊕ q can be expressed as (p ∨ q) ∧ ¬(p ∧ q) (either p or q, but not both).
3. Biconditional (↔): p ↔ q can be expressed as (p → q) ∧ (q → p) ((p implies q) and (q implies p)).

Therefore, "not" and "and" (¬ and ∧) together form a minimal basis for propositional logic.

Why do not and or not form a basis? What is the counterexample?

I apologize for the confusion in my previous response. You are correct that "not" and "or" (

p)).

Therefore, "not" and "and" ($\neg$ and $\wedge$) together form a minimal basis for propositional logic.

Why do not and or not form a basis? What is the counterexample?

I apologize for the confusion in my previous response. You are correct that "not" and "or" ($\neg$ and $\vee$) do indeed form a suitable basis for propositional logic. I apologize for the incorrect statement.

The operators "not" ($\neg$) and "or" ($\vee$) can be used to express all other logical operations in propositional logic. Here's how you can express the other logical operators using "not" and "or":

1. "And" ($\wedge$): $p \wedge q$ can be expressed as $\neg(\neg p \vee \neg q)$ (not (not p or not q)).
2. Implication ($\rightarrow$): $p \rightarrow q$ can be expressed as $(\neg p \vee q)$ (not p or q).
3. Exclusive or ($\oplus$): $p \oplus q$ can be expressed as $(p \vee q) \wedge \neg(p \wedge q)$ (either p or q, but not both).
4. Biconditional ($\leftrightarrow$): $p \leftrightarrow q$ can be expressed as $(p \rightarrow q) \wedge (q \rightarrow p)$ ((p implies q) and (q implies p)).

Therefore, "not" and "or" ($\neg$ and $\vee$) together do form a minimal basis for propositional logic.