

# Satisfiability

Generate-and-test / NP / NP-complete  
SAT, weighted MAX-SAT, CNF-SAT, DNF-SAT, 3-CNF-SAT, Tseitin, TAUT, QSAT  
Some applications: LSAT 😊, register allocation

# From a previous lecture ...

- So, Prof. Eisner, what are **declarative** methods??
  - A declarative program states only *what* is to be achieved
  - A procedural program describes explicitly *how* to achieve it
- **Sorting in a declarative language**
  - “Given array X, find an array Y such that
    - (1) Y is a permutation of X
    - (2) Y’s elements are in non-decreasing order”
  - Compiler is free to invent any sorting algorithm! (Hard?!)
  - You should be aware of when compiler will be efficient/inefficient
- **Sorting in a procedural language**
  - “Given array X, run through it from start to finish, swapping adjacent elements that are out of order ...”
  - Longer and probably buggier
  - Never mentions conditions (1) and (2), except in comments

# Generate-and-test problems

(a common form of declarative programming)

- Problem specified by a fast “checking” function  $f(X,Y)$
- **Input:** string  $x$
- **Output:** Some string  $y$  such that  $f(x,y) = \text{true}$   
 $x, y$  may encode any data you like
- Examples:
  - $f(x,y)$  is true iff  $y$  is sorted permutation of  $x$
  - $f(x,y)$  is true iff  $y$  is timetable that satisfies everyone's preferences  $x$  (*note that  $x$  and  $y$  are encodings as strings*)
- **NP** = {all generate-and-test problems with “easy”  $f$ }
  - $f(x,y)$  can be computed in polynomial time  $O(|x|^k)$ ,  
for *any*  $y$  we should consider (e.g., legal timetables)
  - Could do this for all  $y$  in parallel (NP = “*nondeterministic* polynomial time”)

# Generate-and-compare problems

(a common form of declarative programming)

- Problem is specified by a fast “scoring” function  $f(X,Y)$
- **Input:** string  $x$
- **Output:** Some string  $y$  such that  $f(x,y)$  is **maximized**  
 $x, y$  may encode any data you like
- Examples:
  - $f(x,y)$  evaluates how **well** the timetable  $y$  satisfies everyone’s preferences  $x$  (*note that  $x$  and  $y$  are encodings as strings*)
- **OptP** = {all generate-&-compare problems with easy  $f$ }
  - $f(x,y)$  can be computed in polynomial time  $O(|x|^k)$ ,  
for *any*  $y$  we should consider (e.g., legal timetables)

# An LSAT Practice Problem

*(can we encode this in logic?)*

- When the animated “Creature Buddies” go on tour, they are played by puppets.
- Creatures: Dragon, Gorilla, Kangaroo, Tiger
- Names: Audrey, Hamish, Melville, Rex
- Chief Puppeteers: Ben, Jill, Paul, Sue
- Asst. Puppeteers: Chris, Emily, Faye, Zeke

# An LSAT Practice Problem

*(can we encode this in logic?)*

Creatures: Dragon, Gorilla, Kangaroo, Tiger  
Names: Audrey, Hamish, Melville, Rex  
Chief Puppeteers: Ben, Jill, Paul, Sue  
Asst. Puppeteers: Chris, Emily, Faye, Zeke

- Melville isn't the puppet operated by Sue and Faye.
- Hamish's chief puppeteer (not Jill) is assisted by Zeke.
- Ben does the dragon, but Jill doesn't do the kangaroo.
- Chris assists with the tiger.
- Rex (operated by Paul) isn't the gorilla (not named Melville).

1. What is the Dragon's name?
2. Who assists with puppet Melville?
3. Which chief puppeteer does Zeke assist?
4. What kind of animal does Emily assist with?

Is there a technique that is guaranteed to solve these?

- Generate & test?
- Is the "grid method" faster? Always works?

		creature				assistant				chief							
		D	G	K	T	C	E	F	Z	B	J	P	S				
name	A																
	H																
	M							x					x				
	R																
chief	B									Melville isn't the puppet operated by Sue and Faye.							
	J																
	P																
	S							✓									
	C																
assistant	E																
	F																
	Z																

		creature				assistant				chief			
		D	G	K	T	C	E	F	Z	B	J	P	S
name	A												
	H												
	M							x					x
	R												
chief	B							x					
	J							x					
	P							x					
	S					x	x	✓	x				
assistant	C												
	E												
	F												
	Z												

Melville isn't the puppet operated by Sue and Faye.



		creature				assistant				chief			
		D	G	K	T	C	E	F	Z	B	J	P	S
name	A												
	H								✓		✗		
	M							✗					✗
	R												
chief	B							✗					
	J							✗	✗				
	P							✗					
	S					✗	✗	✓	✗				
assistant	C												
	E												
	F												
	Z												

This English gave a few facts: e.g., Jill is *not* assisted by Zeke.

Hamish's chief puppeteer (not Jill) is assisted by Zeke.

		creature				assistant				chief			
		D	G	K	T	C	E	F	Z	B	J	P	S
name	A								x				
	H					x	x	x	✓		x		
	M							x	x				x
	R								x				
chief	B							x					
	J							x	x				
	P							x					
	S					x	x	✓	x				
assistant	C												
	E												
	F												
	Z												

Hamish's chief puppeteer  
(not Jill) is assisted by  
Zeke.

		creature				assistant				chief			
		D	G	K	T	C	E	F	Z	B	J	P	S
name	A								x				
	H					x	x	x	✓		x		
	M							x	x				x
	R								x				
chief	B	✓	x	x	x			x					
	J	x		x				x	x				
	P	x						x					
	S	x				x	x	✓	x				
assistant	C												
	E												
	F												
	Z												

Ben does the dragon, but  
Jill doesn't do the kangaroo.

		creature				assistant				chief			
		D	G	K	T	C	E	F	Z	B	J	P	S
name	A								x				
	H					x	x	x	✓		x		
	M							x	x				x
	R								x				
chief	B	✓	x	x	x			x					
	J	x		x				x	x				
	P	x						x					
	S	x				x	x	✓	x				
assistant	C	x	x	x	✓								
	E				x								
	F				x								
	Z				x								

Chris assists with the tiger.

		creature				assistant				chief			
		D	G	K	T	C	E	F	Z	B	J	P	S
name	A								x			x	
	H					x	x	x	✓		x	x	
	M		x					x	x			x	x
	R		x						x	x	x	✓	x
chief	B	✓	x	x	x			x					
	J	x		x				x	x				
	P	x						x					
	S	x				x	x	✓	x				
assistant	C	x	x	x	✓								
	E				x								
	F				x								
	Z				x								

Rex (operated by Paul) isn't the gorilla (not named Melville).







		creature				assistant				chief			
		D	G	K	T	C	E	F	Z	B	J	P	S
name	A								x			x	
	H					x	x	x	✓		x	x	
	M		x					x	x			x	x
	R		x						x	x	x	✓	x
chief	B	✓	x	x	x			x					
	J	x		x				x	x				
	P	x	x					x					
	S	x				x	x	✓	x				
assistant	C	x	x	x	✓								
	E				x								
	F				x								
	Z				x								

Paul doesn't  
operate the  
gorilla.

Rex (operated by Paul) isn't  
the gorilla (not named  
Melville).

# Grid method

(for puzzles that match people with roles, etc.)

- Each statement simply tells you to fill a particular cell with  or 
- If you fill a cell with , then fill the rest of its row and column with 
- If a row or column has one blank and the rest are , then fill the blank with 

# Is that enough?

No! It didn't even solve the puzzle before.

We need more powerful reasoning patterns (“propagators”):

- We were told that Ben operates the dragon.
- We already deduced that Ben doesn't work with Faye.
- What should we conclude?



		creature				assistant				chief			
		D	G	K	T	C	E	F	Z	B	J	P	S
name	A								✗			✗	
	H					✗	✗	✗	✓		✗	✗	
	M		✗					✗	✗			✗	✗
	R		✗						✗	✗	✗	✓	✗
chief	B	✓	✗	✗	✗			✗					
	J	✗		✗				✗					
	P	✗	✗					✗					
	S	✗				✗	✗	✓	✗				
assistant	C	✗	✗	✗	✓								
	E				✗								
	F				✗								
	Z				✗								

Combine facts from  
different 4x4 grids:  
If XY ✓ and YZ ✗  
then conclude XZ ✗

		creature				assistant				chief			
		D	G	K	T	C	E	F	Z	B	J	P	S
name	A								x			x	
	H					x	x	x	✓		x	x	
	M		x					x	x			x	x
	R		x						x	x	x	✓	x
chief	B	✓	x	x	x			x					
	J	x		x				x					
	P	x	x					x					
	S	x				x	x	✓	x				
assistant	C	x	x	x	✓								
	E				x								
	F				x								
	Z				x								

Combine facts from  
different 4x4 grids:  
If XY ✓ and YZ ✓  
then conclude XZ ✓

		creature				assistant				chief			
		D	G	K	T	C	E	F	Z	B	J	P	S
name	A								✗			✗	
	H					✗	✗	✗	✓		✗	✗	
	M		✗					✗	✗			✗	✗
	R		✗						✗	✗	✗	✓	✗
chief	B	✓	✗	✗	✗			✗					
	J	✗		✗				✗					
	P	✗	✗					✗					
	S	✗				✗	✗	✓	✗				
assistant	C	✗	✗	✗	✓								
	E				✗								
	F				✗								
	Z				✗								

Combine facts from  
different 4x4 grids:  
If XY ✓ and YZ ✗  
then conclude XZ ✗

		creature				assistant				chief			
		D	G	K	T	C	E	F	Z	B	J	P	S
name	A								✗			✗	
	H					✗	✗	✗	✓		✗	✗	
	M		✗					✗	✗			✗	✗
	R	✗	✗						✗	✗	✗	✓	✗
chief	B	✓	✗	✗	✗	✗		✗					
	J	✗		✗				✗					
	P	✗	✗					✗					
	S	✗				✗	✗	✓	✗				
assistant	C	✗	✗	✗	✓								
	E				✗								
	F	✗			✗								
	Z				✗								

Combine facts from  
different 4x4 grids:  
If XY ✓ and YZ ✗  
then conclude XZ ✗

		creature				assistant				chief							
		D	G	K	T	C	E	F	Z	B	J	P	S				
name	A								✗			✗					
	H				✗	✗	✗	✗	✓		✗	✗					
	M		✗					✗	✗			✗	✗				
	R	✗	✗						✗	✗	✗	✓	✗				
chief	B	✓	✗	✗	✗	✗		✗									
	J	✗		✗				✗									
	P	✗	✗					✗									
	S	✗			✗	✗	✗	✓	✗								
assistant	C	✗	✗	✗	✓	Combine facts from different 4x4 grids: If XY ✓ and YZ ✗ then conclude XZ ✗											
	E				✗												
	F	✗			✗												
	Z				✗												

Combine facts from  
different 4x4 grids:  
If XY ✓ and YZ ✗  
then conclude XZ ✗

		creature				assistant				chief			
		D	G	K	T	C	E	F	Z	B	J	P	S
name	A								✗			✗	
	H				✗	✗	✗	✗	✓		✗	✗	✗
	M		✗					✗	✗			✗	✗
	R	✗	✗					✗	✗	✗	✗	✓	✗
chief	B	✓	✗	✗	✗	✗		✗					
	J	✗		✗				✗					
	P	✗	✗					✗					
	S	✗			✗	✗	✗	✓	✗				
assistant	C	✗	✗	✗	✓								
	E				✗								
	F	✗			✗								
	Z				✗								

Combine facts from  
different 4x4 grids:  
If XY ✓ and YZ ✗  
then conclude XZ ✗

		creature				assistant				chief			
		D	G	K	T	C	E	F	Z	B	J	P	S
name	A								✗			✗	
	H				✗	✗	✗	✗	✓		✗	✗	✗
	M		✗					✗	✗			✗	✗
	R	✗	✗					✗	✗	✗	✗	✓	✗
chief	B	✓	✗	✗	✗	✗		✗					
	J	✗		✗				✗	✗				
	P	✗	✗					✗	✗				
	S	✗			✗	✗	✗	✓	✗				
assistant	C	✗	✗	✗	✓								
	E				✗								
	F	✗			✗								
	Z				✗								

Combine facts from different 4x4 grids:  
 If XY ✓ and YZ ✗  
 then conclude XZ ✗

		creature				assistant				chief			
		D	G	K	T	C	E	F	Z	B	J	P	S
name	A								✗			✗	
	H				✗	✗	✗	✗	✓		✗	✗	✗
	M		✗					✗	✗			✗	✗
	R	✗	✗					✗	✗	✗	✗	✓	✗
chief	B	✓	✗	✗	✗	✗		✗		No new conclusions from this one			
	J	✗		✗				✗	✗				
	P	✗	✗					✗	✗				
	S	✗			✗	✗	✗	✓	✗				
assistant	C	✗	✗	✗	✓	Combine facts from different 4x4 grids: If XY ✓ and YZ ✗ then conclude XZ ✗							
	E				✗								
	F	✗			✗								
	Z				✗								



		creature				assistant				chief			
		D	G	K	T	C	E	F	Z	B	J	P	S
name	A								x			x	
	H				x	x	x	x	✓		x	x	x
	M		x					x	x			x	x
	R	x	x					x	x	x	x	✓	x
chief	B	✓	x	x	x	x		x					
	J	x		x				x	x				
	P	x	x					x	x				
	S	x			x	x	x	✓	x				
assistant	C	x	x	x	✓								
	E				x								
	F	x			x								
	Z				x								

Now we can make some easy progress!

		creature				assistant				chief			
		D	G	K	T	C	E	F	Z	B	J	P	S
name	A								x	x		x	
	H				x	x	x	x	✓	✓	x	x	x
	M		x					x	x	x		x	x
	R	x	x					x	x	x	x	✓	x
chief	B	✓	x	x	x	x		x					
	J	x		x				x	x				
	P	x	x					x	x				
	S	x			x	x	x	✓	x				
assistant	C	x	x	x	✓								
	E				x								
	F	x			x								
	Z				x								

Now we can make some easy progress!

		creature				assistant				chief			
		D	G	K	T	C	E	F	Z	B	J	P	S
name	A								x	x	x	x	
	H				x	x	x	x	✓	✓	x	x	x
	M		x					x	x	x	✓	x	x
	R	x	x					x	x	x	x	✓	x
chief	B	✓	x	x	x	x		x					
	J	x		x				x	x				
	P	x	x					x	x				
	S	x			x	x	x	✓	x				
assistant	C	x	x	x	✓								
	E				x								
	F	x			x								
	Z				x								

Now we can make some easy progress!

		creature				assistant				chief			
		D	G	K	T	C	E	F	Z	B	J	P	S
name	A								x	x	x	x	✓
	H				x	x	x	x	✓	✓	x	x	x
	M		x					x	x	x	✓	x	x
	R	x	x					x	x	x	x	✓	x
chief	B	✓	x	x	x	x	x	x	✓				
	J	x		x				x	x				
	P	x	x					x	x				
	S	x			x	x	x	✓	x				
assistant	C	x	x	x	✓								
	E				x								
	F	x			x								
	Z				x								

The "transitivity rules" let us continue as before. (In fact, they'll let us completely merge name and chief: we don't need to keep track of both anymore.)

# But does the grid method always apply?

(assuming the statements given are enough to solve the problem)

“Each statement simply tells you to fill a particular cell with  or ” – always true?

People: Alice, Bob, Cindy

Roles: Hero, Villain, Jester

- Alice is not the Jester.
- The Villain and the Hero got legally married in Nigeria.

Let's try it on the board and see what happens ...

# But does the grid method always apply?

(assuming the statements given are enough to solve the problem)

- Alice is not the Jester.
- The Villain and the Hero got legally married in Nigeria.

The problem is that the second statement didn't tell us exactly which cells to fill in. It gave us choices.

If these English statements aren't just telling us about cells to fill in, what kinds of things are they telling us?

Can we encode them formally using some little language?

Then the computer can solve them ...

# Logical formulas (inductive definition)

- If  $A$  is a [boolean] **variable**, then  $A$  and  $\sim A$  are “literal” formulas.
- If  $F$  is a formula, then so is  $\sim F$  (“not  $F$ ”).

"not F"

F		$\sim F$					
0		1					
0		1					
1		0					
1		0					

# Logical formulas (inductive definition)

- If  $A$  is a [boolean] **variable**, then  $A$  and  $\sim A$  are “literal” formulas.
- If  $F$  is a **formula**, then so is  $\sim F$  (“not  $F$ ”).
- If  $F$  and  $G$  are **formulas**, then so are  $F \wedge G$ ,  $F \vee G$ , etc.
- “**Truth tables**” – like the multiplication table, but for booleans

“not  $F$ ”

F	G	$\sim F$					
0	0	1					
0	1	1					
1	0	0					
1	1	0					



# Logical formulas (inductive definition)

- If  $A$  is a [boolean] **variable**, then  $A$  and  $\sim A$  are “literal” formulas.
- If  $F$  is a **formula**, then so is  $\sim F$  (“not  $F$ ”).
- If  $F$  and  $G$  are **formulas**, then so are  $F \wedge G$ ,  $F \vee G$ , etc.
- “**Truth tables**” – like the multiplication table, but for booleans

“not  $F$ ”   “ $F$  and  $G$ ”

F	G	$\sim F$	$F \wedge G$				
0	0	1	0				
0	1	1	0				
1	0	0	0				
1	1	0	1				

# Logical formulas (inductive definition)

- If  $A$  is a [boolean] **variable**, then  $A$  and  $\sim A$  are “literal” formulas.
- If  $F$  is a **formula**, then so is  $\sim F$  (“not  $F$ ”).
- If  $F$  and  $G$  are **formulas**, then so are  $F \wedge G$ ,  $F \vee G$ , etc.
- “**Truth tables**” – like the multiplication table, but for booleans

“not  $F$ ”   “ $F$  and  $G$ ”   “ $F$  or  $G$ ”

F	G	$\sim F$	$F \wedge G$	$F \vee G$			
0	0	1	0	0			
0	1	1	0	1			
1	0	0	0	1			
1	1	0	1	1			

# Logical formulas (inductive definition)

- If  $A$  is a [boolean] **variable**, then  $A$  and  $\sim A$  are “literal” formulas.
- If  $F$  is a **formula**, then so is  $\sim F$  (“not  $F$ ”).
- If  $F$  and  $G$  are **formulas**, then so are  $F \wedge G$ ,  $F \vee G$ , etc.
- “**Truth tables**” – like the multiplication table, but for booleans

“not  $F$ ”   “ $F$  and  $G$ ”   “ $F$  or  $G$ ”   “ $F$  xor  $G$ ”

F	G	$\sim F$	$F \wedge G$	$F \vee G$	$F \oplus G$		
0	0	1	0	0	0		
0	1	1	0	1	1		
1	0	0	0	1	1		
1	1	0	1	1	0		

# Logical formulas (inductive definition)

- If  $A$  is a [boolean] **variable**, then  $A$  and  $\sim A$  are “literal” formulas.
- If  $F$  is a **formula**, then so is  $\sim F$  (“not  $F$ ”).
- If  $F$  and  $G$  are **formulas**, then so are  $F \wedge G$ ,  $F \vee G$ , etc.
- “**Truth tables**” – like the multiplication table, but for booleans

“not  $F$ ”   “ $F$  and  $G$ ”   “ $F$  or  $G$ ”   “ $F$  xor  $G$ ”   “ $F$  iff  $G$ ”

F	G	$\sim F$	$F \wedge G$	$F \vee G$	$F \oplus G$	$F \leftrightarrow G$	
0	0	1	0	0	0	1	
0	1	1	0	1	1	0	
1	0	0	0	1	1	0	
1	1	0	1	1	0	1	

# Logical formulas (inductive definition)

- If  $A$  is a [boolean] **variable**, then  $A$  and  $\sim A$  are “literal” formulas.
- If  $F$  is a **formula**, then so is  $\sim F$  (“not  $F$ ”).
- If  $F$  and  $G$  are **formulas**, then so are  $F \wedge G$ ,  $F \vee G$ , etc.
- “**Truth tables**” – like the multiplication table, but for booleans

“not  $F$ ”   “ $F$  and  $G$ ”   “ $F$  or  $G$ ”   “ $F$  xor  $G$ ”   “ $F$  iff  $G$ ”   “ $F$  implies  $G$ ”

F	G	$\neg F$ $\sim F$	$F \wedge G$	$F \vee G$	$F \oplus G$	$F \leftrightarrow G$	$F \rightarrow G$
0	0	1	0	0	0	1	1
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	0
1	1	0	1	1	0	1	1

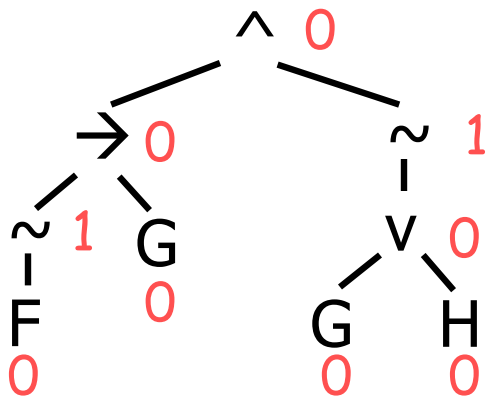
# Logical formulas (inductive definition)

- If A is a [boolean] **variable**, then A and  $\sim A$  are “literal” formulas.
- If F is a **formula**, then so is  $\sim F$  (“not F”).
- If F and G are **formulas**, then so are  $F \wedge G$ ,  $F \vee G$ , etc.

↑  
Base case

↑   ↑  
Build bigger formulas out of littler ones

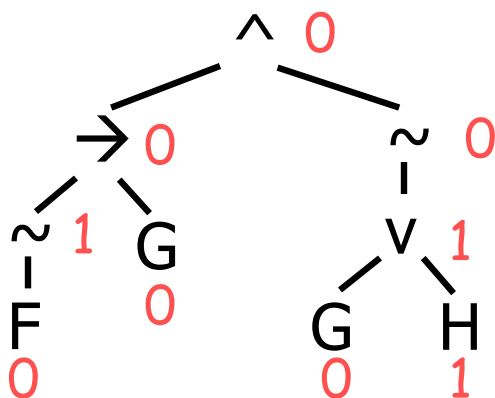
- So this is a formula:  $(\sim F \rightarrow G) \wedge \sim(G \vee H)$
- Given an **assignment** of values to the variables F, G, H,
  - We can compute the value of the whole formula, bottom-up
  - Just like evaluating arithmetic expressions:  $(-F/G) \times -(F \times H)$



F	G	H	$(\sim F \rightarrow G) \wedge \sim(G \vee H)$
0	0	0	0

# Logical formulas (inductive definition)

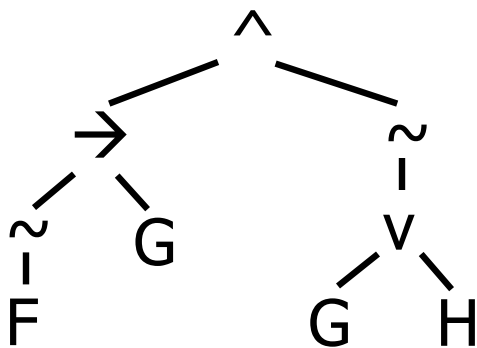
- If A is a [boolean] **variable**, then A and  $\sim A$  are “literal” formulas.
  - If F is a **formula**, then so is  $\sim F$  (“not F”).
  - If F and G are **formulas**, then so are  $F \wedge G$ ,  $F \vee G$ , etc.
- 
- So this is a formula:  $(\sim F \rightarrow G) \wedge \sim(G \vee H)$
  - Given an **assignment** of values to the variables F, G, H,
    - We can compute the value of the whole formula, bottom-up
    - Just like evaluating arithmetic expressions:  $(-F/G) \times -(F \times H)$



F	G	H	$(\sim F \rightarrow G) \wedge \sim(G \vee H)$
0	0	0	0
0	0	1	0

# Logical formulas (inductive definition)

- If  $A$  is a [boolean] **variable**, then  $A$  and  $\sim A$  are “literal” formulas.
  - If  $F$  is a **formula**, then so is  $\sim F$  (“not  $F$ ”).
  - If  $F$  and  $G$  are **formulas**, then so are  $F \wedge G$ ,  $F \vee G$ , etc.
- 
- So this is a formula:  $(\sim F \rightarrow G) \wedge \sim(G \vee H)$
  - Given an **assignment** of values to the variables  $F, G, H$ ,
    - We can compute the value of the whole formula, bottom-up
    - Just like evaluating arithmetic expressions:  $(-F/G) \times -(F \times H)$



F	G	H	$(\sim F \rightarrow G) \wedge \sim(G \vee H)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1



# Logical formulas (inductive definition)

- If A is a [boolean] **variable**, then A and  $\sim A$  are “literal” formulas.
- If F is a **formula**, then so is  $\sim F$  (“not F”).
- If F and G are **formulas**, then so are  $F \wedge G$ ,  $F \vee G$ , etc.
- So this is a formula:  $(\sim F \rightarrow G) \wedge \sim(G \vee H)$
- Given an **assignment** of values to the variables F, G, H,
  - We can compute the value of the whole formula, bottom-up
  - Just like evaluating arithmetic expressions:  $(-F/G) \times -(F \times H)$

This particular formula is **satisfied** (has value 1) for only one assignment:  
 $F=1, G=0, H=0$

So it's equivalent to  
 $F \wedge \sim G \wedge \sim H$

F	G	H	$(\sim F \rightarrow G) \wedge \sim(G \vee H)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1

# Logical formulas (inductive definition)

- If A is a [boolean] **variable**, then A and  $\sim A$  are “literal” formulas.
- If F is a **formula**, then so is  $\sim F$  (“not F”).
- If F and G are **formulas**, then so are  $F \wedge G$ ,  $F \vee G$ , etc.
- So this is a formula:  $(\sim F \rightarrow G) \wedge \sim(G \vee H)$
- Given an **assignment** of values to the variables F, G, H,
  - We can compute the value of the whole formula, bottom-up
  - Just like evaluating arithmetic expressions:  $(-F/G) \times -(F \times H)$

This particular formula is **satisfied** (has value 1) for only one assignment:  
 $F=1, G=0, H=0$

F	G	H	$(\sim F \rightarrow G) \wedge \sim(G \vee H)$
1	0	0	1
otherwise			0

So it's equivalent to  
 $F \wedge \sim G \wedge \sim H$

More concise form of truth table

# Logical formulas (inductive definition)

- If A is a [boolean] **variable**, then A and  $\sim A$  are “literal” formulas.
- If F is a **formula**, then so is  $\sim F$  (“not F”).
- If F and G are **formulas**, then so are  $F \wedge G$ ,  $F \vee G$ , etc.
- So this is a formula:  $(\sim F \rightarrow G) \wedge \sim(G \vee H)$
- Given an **assignment** of values to the variables F, G, H,
  - We can compute the value of the whole formula, bottom-up
  - Just like evaluating arithmetic expressions:  $(-F/G) \times -(F \times H)$

This particular formula is **satisfied** (has value 1) for only one assignment:  
 $F=1, G=0, H=0$

So it's equivalent to  
 $F \wedge \sim G \wedge \sim H$

F	G	H	$(\sim F \rightarrow G) \wedge \sim(G \vee H)$
0	*	*	0
1	0	0	1
1	0	1	0
1	1	*	0

Compressed version with “don't care” values \*

# Satisfying assignments

- An “assignment” of values to boolean variables is a choice of 0 (false) or 1 (true) for each variable
  - Given an assignment, can compute whether formula is satisfied (true)
- Satisfiability problem:
  - Input: a formula
  - Output: a satisfying assignment to its variables, if one exists. Otherwise return “UNSAT”.

# Asking for a satisfying assignment

- Let's try encoding the hero-villain problem on the board:

People: Alice, Bob, Cindy

Roles: Hero, Villain, Jester

- Alice is not the Jester.
- The Villain and the Hero got legally married in Nigeria.

		"not F"	"F and G"	"F or G"	"F xor G"	"F iff G"	"F implies G"
		$\neg F$	$F \wedge G$	$F \vee G$	$F \oplus G$	$F \leftrightarrow G$	$F \rightarrow G$
F	G	$\sim F$	$F \wedge G$	$F \vee G$	$F \oplus G$	$F \leftrightarrow G$	$F \rightarrow G$
0	0	1	0	0	0	1	1
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	0
1	1	0	1	1	0	1	1

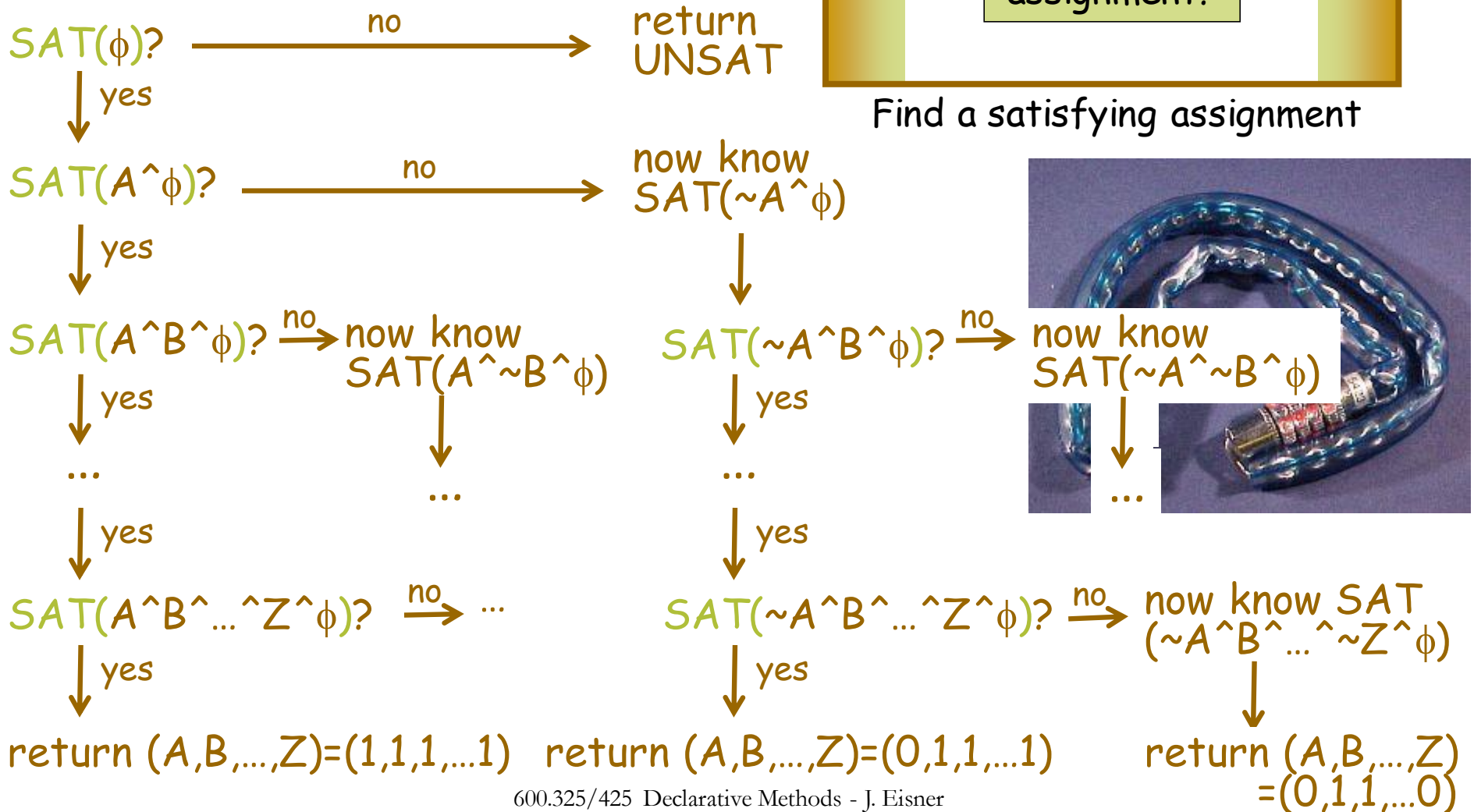
# Satisfying assignments

- An “assignment” of values to boolean variables is a choice of 0 (false) or 1 (true) for each variable
  - Given an assignment, can compute whether formula is satisfied (true)
- **Satisfiability problem:**
  - Input: a formula
  - Output: a satisfying assignment to its variables, if one exists. Otherwise return “UNSAT”.
- **Traditional version:**
  - Input: a formula
  - Output: “**true**” if there exists a satisfying assignment, else “**false**.”
  - **Why isn’t this useless in practice??**



# A Reduction

*Unlocking an A..Z assignment in 26 steps*



# Another LSAT Practice Problem


- When the goalie has been chosen, the Smalltown Bluebirds hockey team has a starting lineup that is selected from two groups:
  - 1st Group: John, Dexter, Bart, Erwin
  - 2nd Group: Leanne, Roger, George, Marlene, Patricia
- Certain requirements are always observed (for the sake of balance, cooperation, and fairness).



# Another LSAT Practice Problem

1st Group: John, Dexter, Bart, Erwin

2nd Group: Leanne, Roger, George, Marlene, Patricia

- Two players are always chosen from the 1st group.  
Three players are always chosen from the 2nd group. 
  - George will only start if Bart also starts.
  - Dexter and Bart will not start together.
  - If George starts, Marlene won't start.
  - The 4 fastest players are: John, Bart, George and Patricia.  
3 of the 4 fastest players will always be chosen.
- "3 of 5" here. Could you efficiently encode "13 of 26"?
1. Who always starts?
  2. If Marlene starts, what subset of first-group players starts with her?
  3. If George starts, who must also start?  
(M or J, D or L, D or J, J or P, M or R)
  4. Which of these pairs cannot start together? (ED, GJ, RJ, JB, PM)
- These questions are different:  $\forall$  not  $\exists$

# Encoding “at least 13 of 26”

*(without listing all 38,754,732 subsets!)*

A	B	C	...	L	M	...	Y	Z
$A \geq 1$	$A-B \geq 1$	$A-C \geq 1$		$A-L \geq 1$	$A-M \geq 1$		$A-Y \geq 1$	$A-Z \geq 1$
	$A-B \geq 2$	$A-C \geq 2$		$A-L \geq 2$	$A-M \geq 2$		$A-Y \geq 2$	$A-Z \geq 2$
		$A-C \geq 3$		$A-L \geq 3$	$A-M \geq 3$		$A-Y \geq 3$	$A-Z \geq 3$
26 original variables A ... Z, plus $< 26^2$ new variables such as $A-L \geq 3$				...	...		...	...
				$A-L \geq 12$	$A-M \geq 12$		$A-Y \geq 12$	$A-Z \geq 12$
					$A-M \geq 13$		$A-Y \geq 13$	$A-Z \geq 13$

- SAT formula should require that  $A-Z \geq 13$  is true ... and what else?
- yadayada  $\wedge A-Z \geq 13 \wedge (A-Z \geq 13 \rightarrow (A-Y \geq 13 \vee (A-Y \geq 12 \wedge Z)))$   
 $\wedge (A-Y \geq 13 \rightarrow (A-X \geq 13 \vee (A-X \geq 12 \wedge Y))) \wedge \dots$

# Encoding “at least 13 of 26”

(without listing all 38,754,732 subsets!)

A	B	C	...	L	M	...	Y	Z
$A \geq 1$	$\leftarrow A-B \geq 1$	$\leftarrow A-C \geq 1$		$\leftarrow A-L \geq 1$	$\leftarrow A-M \geq 1$		$\leftarrow A-Y \geq 1$	$\leftarrow A-Z \geq 1$
	$\leftarrow A-B \geq 2$	$\leftarrow A-C \geq 2$		$\leftarrow A-L \geq 2$	$\leftarrow A-M \geq 2$		$\leftarrow A-Y \geq 2$	$\leftarrow A-Z \geq 2$
		$\leftarrow A-C \geq 3$		$\leftarrow A-L \geq 3$	$\leftarrow A-M \geq 3$		$\leftarrow A-Y \geq 3$	$\leftarrow A-Z \geq 3$
26 original variables A ... Z, plus $< 26^2$ new variables such as $A-L \geq 3$				...	...		...	...
				$\leftarrow A-L \geq 12$	$\leftarrow A-M \geq 12$		$\leftarrow A-Y \geq 12$	$\leftarrow A-Z \geq 12$
					$\leftarrow A-M \geq 13$		$\leftarrow A-Y \geq 13$	$\leftarrow A-Z \geq 13$

- SAT formula should require that  $A-Z \geq 13$  is true ... and what else?

- yadayada  $\wedge A-Z \geq 13 \wedge$  
 $(A-Z \geq 13 \rightarrow (A-Y \geq 13 \vee (A-Y \geq 12 \wedge Z)))$   
 $\wedge (A-Y \geq 13 \rightarrow (A-X \geq 13 \vee (A-X \geq 12 \wedge Y))) \wedge \dots$

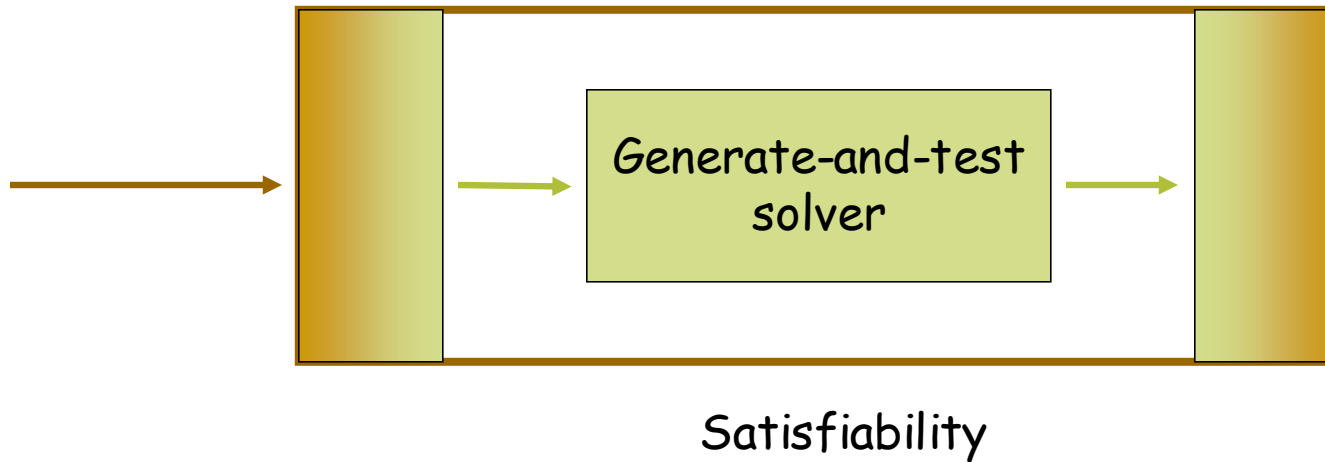
one “only if” definitional constraint for each new variable

# Chinese Dinner After the LSAT



# Relation to generate-and-test

A general-purpose "generate-and-test solver" is really called a Nondeterministic Turing Machine.



Inner solver is general-purpose and can be given any checking function  $f$  as part of its input.

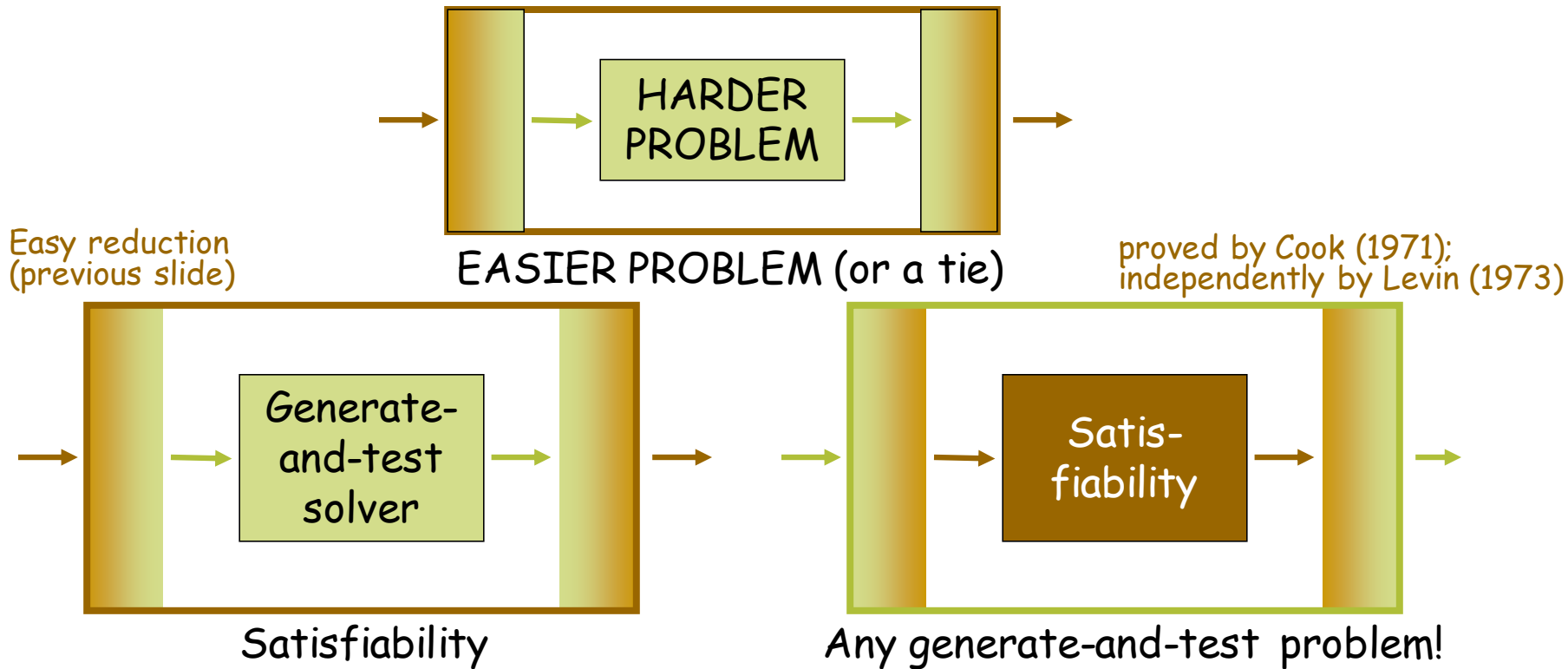
How does this reduction work?

Recall the inner solver's input and output:

**Input:** string  $x$ ; checking function  $f(x,y)$

**Output:** Some string  $y$  such that  $f(x,y) = \text{true}$

# Relation to generate-and-test

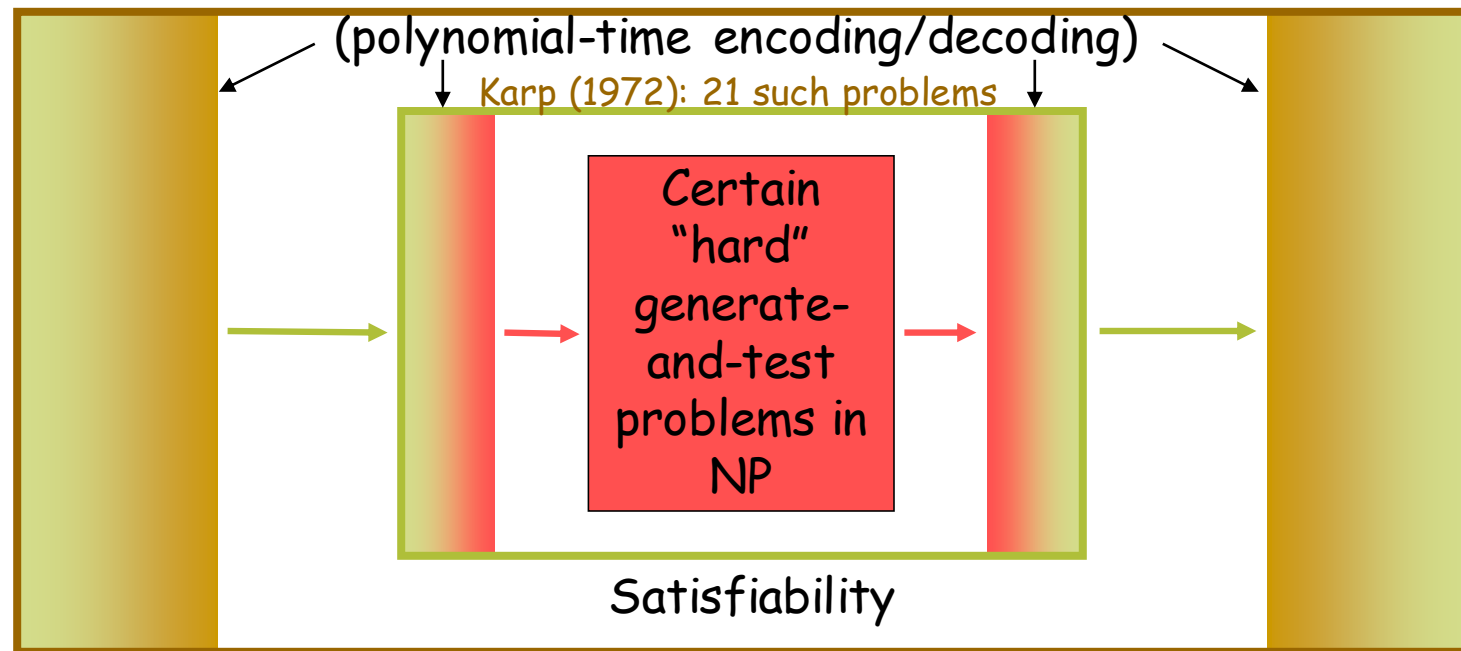


**Interreducible problems! (under polynomial encoding/decoding)**

So if we could find a fast solver for SAT, we'd solve half the world's problems  
(but create new problems, e.g., cryptography wouldn't work anymore)  
(neither would some theoretical computer scientists)

So probably impossible. But a "pretty good" SAT solver would help the world.

# NP-completeness



Any generate-and-test problem in NP

These innermost problems are called NP-complete

They are all interreducible with SAT

A "pretty good" solver for **any** of them would help the world equally

Many practically important examples: see course homepage for pointers

You may well discover some yourself - most things you want are NP-complete

# Local register allocation

*(an NP-complete planning problem from systems)*

- Computer has N fast registers, and unlimited slow memory
- Given a straight sequence of computations using  $M > N$  vars:
  - ...  $x = y * z$ ;  $w = x + y$ ; ...
- Generate assembly-language code:
  - #####  $x = y * z$  #####
  - ~~LOAD R2, (address of y)~~
  - ~~LOAD R3, (address of z)~~
  - MUL **R1** R2 R3
  - ~~STORE (address of x), R1~~
  - #####  $w = x + y$  #####
  - ~~LOAD R1, (address of x)~~ x is already in a register (R1)
  - ~~LOAD R2, (address of y)~~ y is already in a register (R2)
  - ADD **R3** R1 R2 Clever to pick R3? Is z the best thing to overwrite?
  - ~~STORE (address of w), R3~~
- Can we eliminate or minimize loads/stores? *Reorder code?*

Do we need these? Not if y, z are already in registers

Do we need this? Not if we can arrange to keep x in R1 until we need it again



# Register allocation as a SAT problem

- Tempting to state everything a reasonable person would know:
  - Don't have the same variable in two registers
  - Don't store something unless you'll need it again ("liveness")
  - Don't store something that's already in memory
  - Only ever store the result of the computation you just did
  - ...
- Yes, looks like an optimal solution will observe these constraints
- But you don't have to state them!
- They are part of the solution, not part of the problem
  - (violating them is allowed, merely dumb)
- Stating these extra constraints *might speed up* the SAT solver
  - Helps the solver prune away stupid possibilities
  - But that's just an optimization you can add later
  - *Might* slow things down if the overhead exceeds the benefit
  - A great SAT solver might discover these constraints for itself

# Assumptions about the input code

- Assume all statements look like  $x = y * z$ 
  - Preprocess input to eliminate more complex statements:

$$a = b * c + d * e \longrightarrow \begin{cases} \text{temp1} = b * c \\ \text{temp2} = d * e \\ a = \text{temp1} + \text{temp2} \end{cases}$$

- Now all intermediate quantities are associated with variables
  - So if needed, we can store them to memory and reload them later
- Assume variables are constant once computed
  - Preprocess: If  $z$  takes on two different values, split it into  $z_1$  and  $z_2$

$$\begin{array}{l} x = y * z \\ z = x + 1 \\ a = a + z \end{array} \longrightarrow \begin{cases} x = y * z_1 \\ z_2 = x + 1 \\ a_2 = a_1 + z_2 \end{cases}$$

- When  $z_2$  is in a register or memory, we know which version we have
- 
- "static single assignment (SSA) form"

# How to set up SAT variables?

(one possibility)

Typical when encoding  
a planning problem

Other planning problems:

- Get dressed
- Solve Rubik's cube

## ■ K time steps, M program vars, N registers

- ❑ SAT variable **2y3** is true iff at time 2,  $y$  is in R3
- ❑ SAT variable **2y** is true iff at time 2,  $y$  is in memory
- ❑ An assignment to all variables defines a “configuration” of the machine at each time  $t$
- ❑ How many variables total?

– Time 0 config –  
load-compute-store

– Time 1 config –  
load-compute-store

⇒ Time 2 config –

...

– Time K-1 config –  
load-compute-store

– Time K config –

## ■ How will we decode the SAT solution? (which only tells us what the configs are)

- ❑ For each time  $t$ , decoder will generate a **load-compute-store** sequence of machine code to get to  $t$ 's configuration from  $t-1$ 's

1. To put some new values in registers, generate **loads**
2. To put other new values in registers, generate **computes**  
(might use newly loaded registers; order multiple computes topologically)

preferred  
(faster  
than loads)

3. To put new values in memory, generate **stores**

(might use newly computed registers)

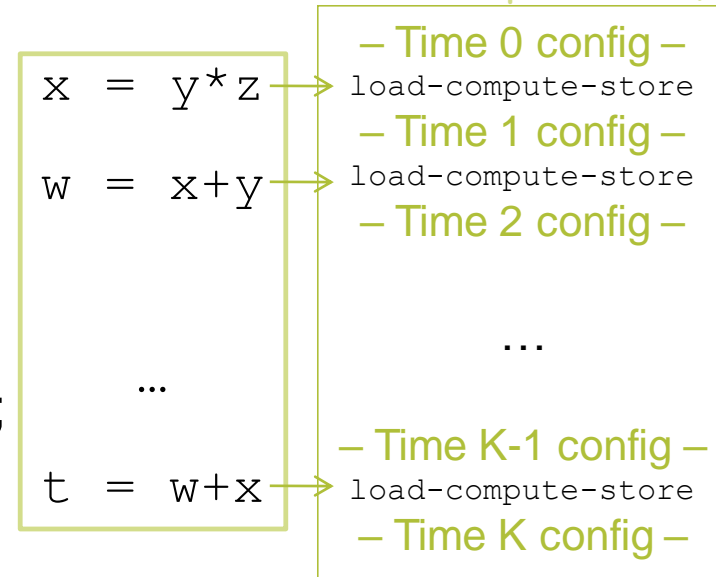
# How to set up SAT variables?

(one possibility)

Allows dumb solution from a few slides ago (not most efficient, but proves the formula with K steps is SAT)

## ■ K time steps

- ❑ How do we pick K?
- ❑ Large enough to compile input
- ❑ Not too large, or solver is slow
- ❑ Solution: Let  $K = \#$  of input instructions; we know that's large enough ...



## ■ How will we decode the SAT solution?

- ❑ For each time  $t$ , will generate a load-compute-store sequence of machine code to get to  $t$ 's configuration from  $t-1$ 's
- 1. To put some new values in registers, generate loads
- 2. To put other new values in registers, generate computes  
(might use newly loaded registers; order multiple computes topologically)
- 3. To put new values in memory, generate stores  
(might use newly computed registers)

preferred  
(faster)

# Example: How do we get from t-1 to t?

	R1	R2	R3	Mem
Time t-1	y			y,z
Time t	x		z	x,y,z

- ❑ **LOAD** R3, (address of z)
- ❑ **MUL** R1 R1 R3 // compute x into R1, overwriting y
- ❑ **STORE** (address of x), R1

- 
- ❑ For each time t, generate a **load-compute-store** sequence to get to t's configuration from t-1's
    1. To put some new values in registers, generate loads
    2. To put other new values in registers, generate computes  
(might use newly loaded registers; order multiple computes topologically)
    3. To put new values in memory, generate stores  
(might use newly computed registers)

– Time 0 –  
 $x = y * z$

– Time 1 –  
 $w = x + y$

– Time 2 –

...

– Time K-1 –  
 $t = w + x$


– Time K –

# What is the SAT formula?

This setup may even reorder or eliminate computations!

- Many constraint clauses combined with “and”:
  - At time K, desired output vars are in mem/registers
  - At time 0, input vars are in mem/certain registers
    - ... but otherwise, no vars are held anywhere!
    - E.g.,  $\sim 0y \wedge 0y1 \wedge \sim 0y2 \wedge \dots \wedge \sim 0x \wedge \sim 0x1 \wedge \sim 0x2 \dots$
  - No two vars can be in same register at same time
  - If w is in a register at time t but not t-1, where  $w=x*y$ , then
    - either w was in memory at time t-1 (load)
    - or (better) x, y were also in registers at time t (compute)
  - If w is in memory at time t but not t-1, then
    - w was also in a register at time t (store)
- What's missing?

This box turns into lots of constraints (why?)



If these hold, we can find an adequate load-compute-store sequence for  $t-1 \rightarrow t$

# Where did we say to minimize loads/stores???

## ■ Additional constraints:

- ❑ No loads
- ❑ No stores

## ■ But what if SAT then fails?

- ❑ Retry with a weaker constraint: “at most 1 load/store” ...
- ❑ Continue by linear search or binary chop

If  $w$  is in a register at time  $t$  but not  $t-1$ , then  
either  $w$  was in memory at time  $t-1$   
or (better)  $x, y$  were also in registers at time  $t$

$\left\{ \begin{array}{l} D \\ E \end{array} \right.$  If  $w$  is in memory at time  $t$  but not  $t-1$ , then  
 $w$  was also in a register at time  $t$

Strengthen our constraints  
 $A \rightarrow (B \vee C)$   
to  $A \rightarrow C$

and  $D \rightarrow E$   
to  $\sim D$

# Alternatively, use MAX-SAT

- Specify our input formula as a *conjunction* of several subformulas (known as “clauses” or “constraints”)
- A **MAX-SAT** solver seeks an assignment that satisfies **as many clauses as possible**
- Such conjunctive formulas are very common:
  - each clause encodes a **fact** about the input, or a **constraint**
- Can specify a *weight* for each clause (not all equally important)
- **Weighted MAX-SAT**: seek an assignment that satisfies a subset of clauses with **the maximum total weight possible**
  - How does this relate to unweighted MAX-SAT?
  - Can you implement either version by wrapping the other?
- How do we encode our register problem?
  - Specifically, how do we implement “hard” vs. “soft” constraints?



# Minimizing loads/stores with weighted MAX-SAT

We require certain clauses to be satisfied (e.g., so that we'll be able to decode the satisfying assignment into a load-compute-store sequence).

We give these **hard constraints** a weight of  $\infty$ . The MAX-SAT solver must respect them.

It's okay to violate "don't load" and "don't store," but each violation costs 150 cycles of runtime.

We give each of these **soft constraints** a weight of 150. The MAX-SAT solver will try to satisfy as many as it can.

Include both kinds ...

weight  $\infty$ :  $A \rightarrow (B \vee C)$

weight 150:  $A \rightarrow C$

weight  $\infty$ :  $D \rightarrow E$

weight 150:  $\sim D$

If  $w$  is in a register at time  $t$  but not  $t-1$ , then  
either  $w$  was in memory at time  $t-1$   
or (better)  $x, y$  were also in registers at time  $t$

$\left\{ \begin{array}{l} D \\ E \end{array} \right.$  If  $w$  is in memory at time  $t$  but not  $t-1$ , then  
 $w$  was also in a register at time  $t$

Strengthen our constraints  
 $A \rightarrow (B \vee C)$   
to  $A \rightarrow C$

and  $D \rightarrow E$   
to  $\sim D$

# Eliminating infinite-weight clauses ...

- Can specify a *weight* for each clause (not all equally important)
- **Weighted MAX-SAT**: seek an assignment that satisfies a subset of clauses with **the maximum total weight possible**
- What if the solver doesn't allow  $\infty$  as a clause weight?

## Hard constraints

weight $\infty$ :	$A \rightarrow (B \vee C)$
weight $\infty$ :	$D \rightarrow E$
	...

## Soft constraints

weight 20:	$A \rightarrow C$
weight 14:	$\sim D$
	...

TOTAL WEIGHT 999

- Just give weight  $999+1 = 1000$  to each hard constraint in this case
- This acts like weight  $\infty$ , since solver would rather violate all soft constraints than violate even one hard constraint!
- So solver will only violate hard constraints if it can't avoid doing so
  - Check the solution. If any hard constraints are violated, then the set of hard constraints must be UNSAT.

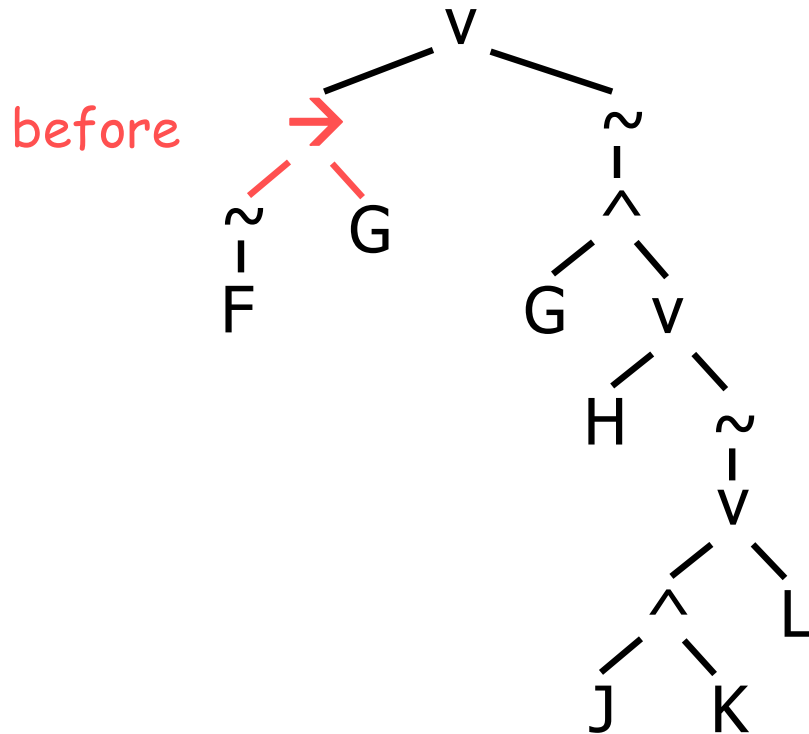
# Simpler formulas, simpler solvers

- If  $A$  is a [boolean] variable, then  $A$  and  $\sim A$  are “literal” formulas.
- If  $F$  and  $G$  are formulas, then so are
  - $F \wedge G$  (“ $F$  and  $G$ ”)
  - $F \vee G$  (“ $F$  or  $G$ ”)
  - $F \rightarrow G$  (“If  $F$  then  $G$ ”; “ $F$  implies  $G$ ”)
  - $F \leftrightarrow G$  (“ $F$  if and only if  $G$ ”; “ $F$  is equivalent to  $G$ ”)
  - $F \oplus G$  (“ $F$  or  $G$  but not both”; “ $F$  differs from  $G$ ”)
  - $\sim F$  (“not  $F$ ”)
- Is all of this fancy notation really necessary?
  - Or is some of it just syntactic sugar?
  - Can we write a front-end preprocessor that reduces the user’s formula to a simpler notation? That would simplify solver’s job.

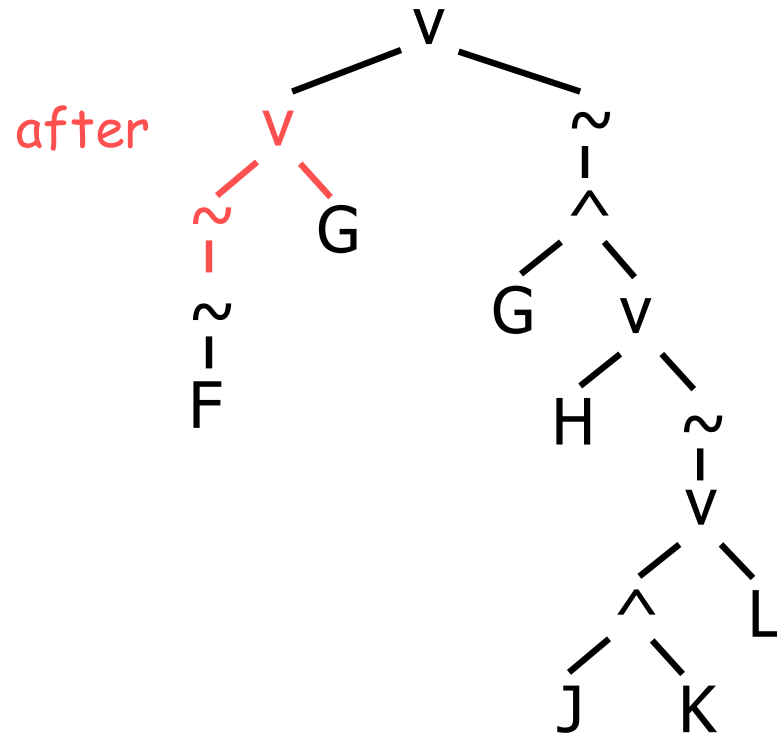
# Simpler formulas, simpler solvers

- If  $A$  is a [boolean] variable, then  $A$  and  $\sim A$  are “literal” formulas.
- If  $F$  and  $G$  are formulas, then so are
  - $F \wedge G$
  - $F \vee G$
  - ~~$F \rightarrow G$~~   $\dashrightarrow (\sim F \vee G)$
  - ~~$F \leftrightarrow G$~~   $\dashrightarrow (F \rightarrow G) \wedge (G \rightarrow F)$       Alternatively:  $(F \wedge G) \vee (\sim F \wedge \sim G)$
  - ~~$F \oplus G$~~   $\dashrightarrow (F \wedge \sim G) \vee (\sim F \wedge G)$
  - $\sim F$

Step 1: Eliminate  $\leftrightarrow$ , xor, then  $\rightarrow$



Step 1: Eliminate  $\leftrightarrow$ , xor, then  $\rightarrow$



# Simpler formulas, simpler solvers

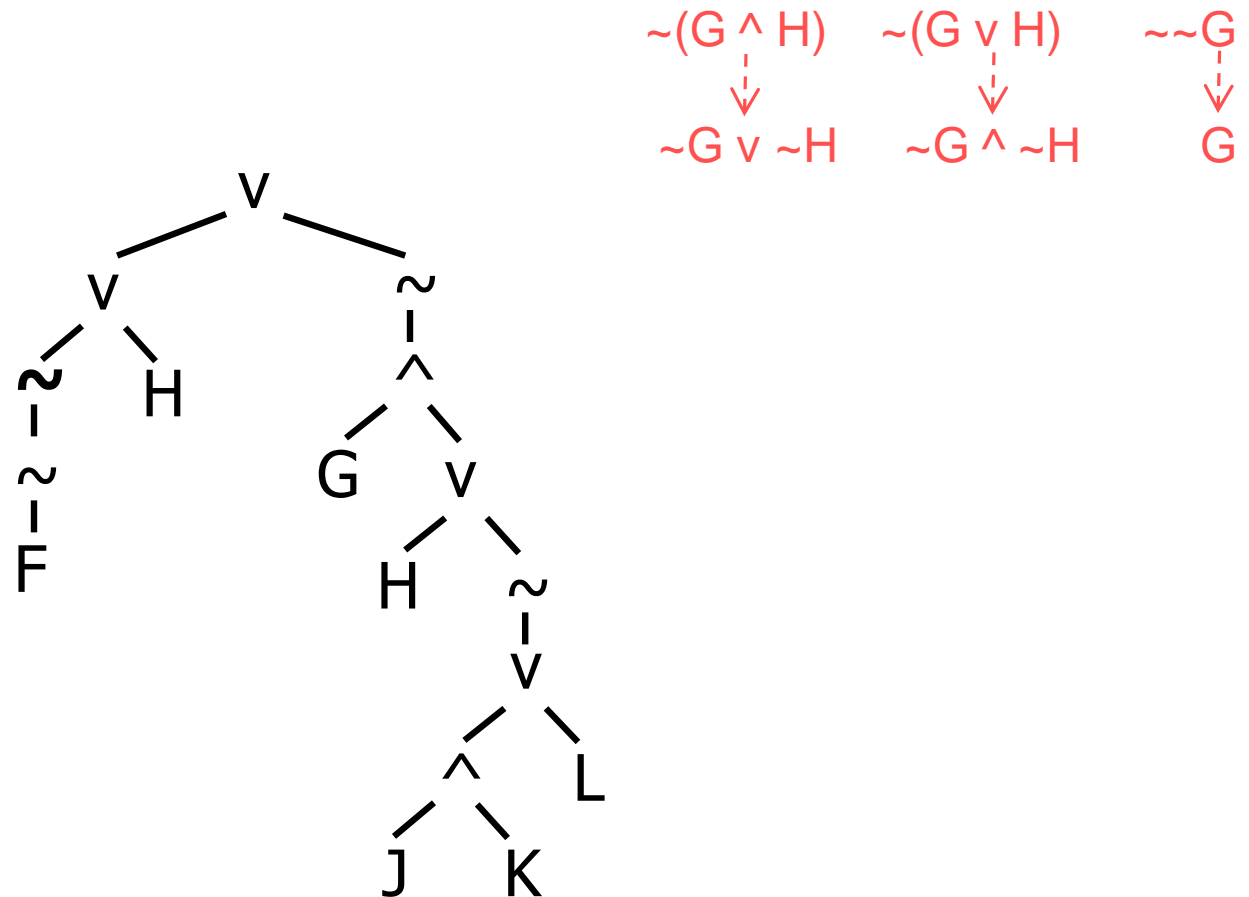
- If  $A$  is a [boolean] variable, then  $A$  and  $\sim A$  are “literal” formulas.
- If  $F$  and  $G$  are formulas, then so are
  - $F \wedge G$
  - $F \vee G$
  - ~~$F \rightarrow G$~~   $\rightarrow (\sim F \vee G)$
  - ~~$F \leftrightarrow G$~~   $\rightarrow (F \rightarrow G) \wedge (G \rightarrow F)$       Alternatively:  $(F \wedge G) \vee (\sim F \wedge \sim G)$
  - ~~$F \oplus G$~~   $\rightarrow (F \wedge \sim G) \vee (\sim F \wedge G)$
  - ~~$\sim F$~~       If not already a literal, must have form  $\sim(G \wedge H)$  or  $\sim(G \vee H)$  or  $\sim\sim G$ 

$\downarrow$   
 $\sim G \vee \sim H$

$\downarrow$   
 $\sim G \wedge \sim H$

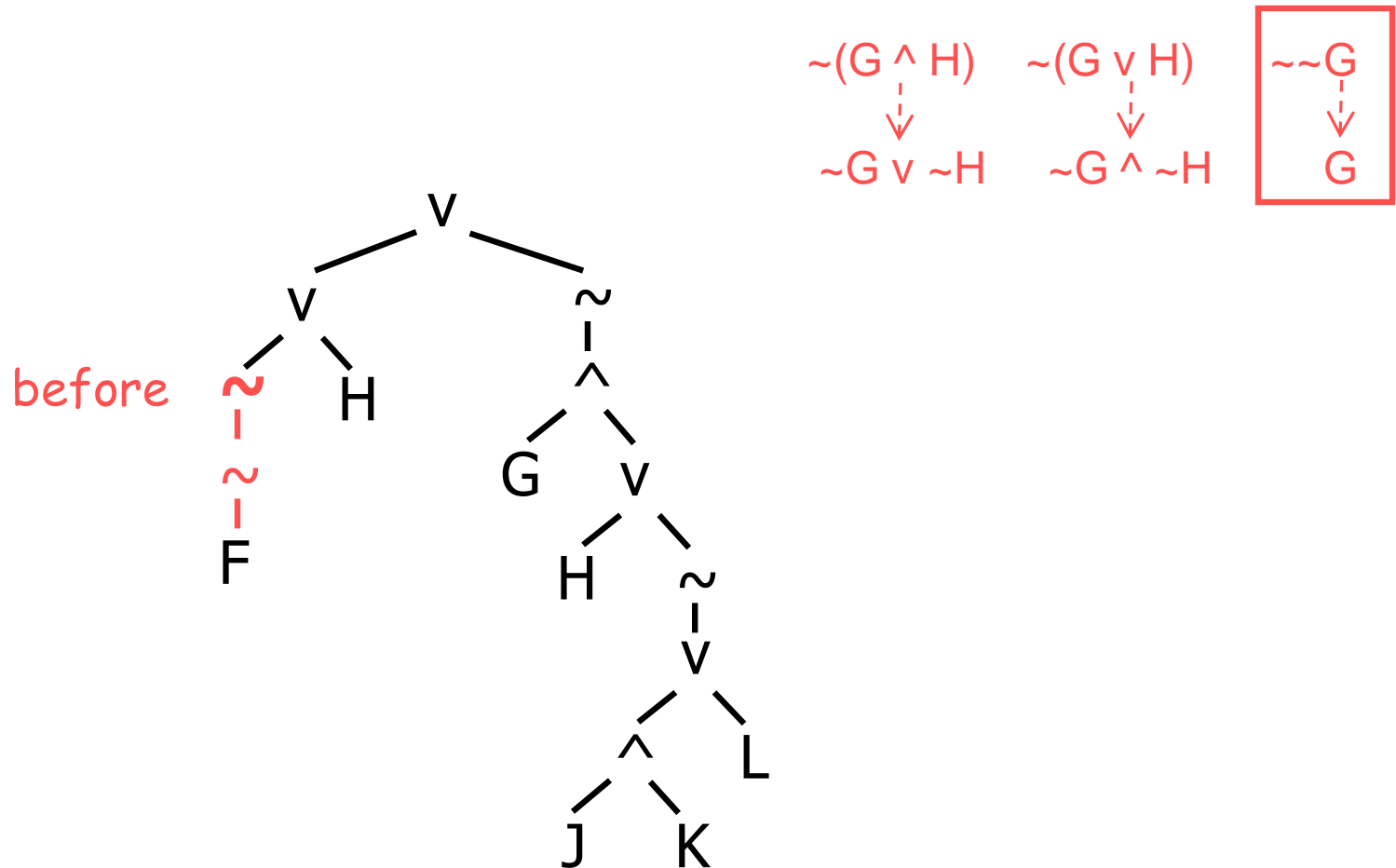
$\downarrow$   
 $G$

## Step 2: Push $\sim$ down to leaves

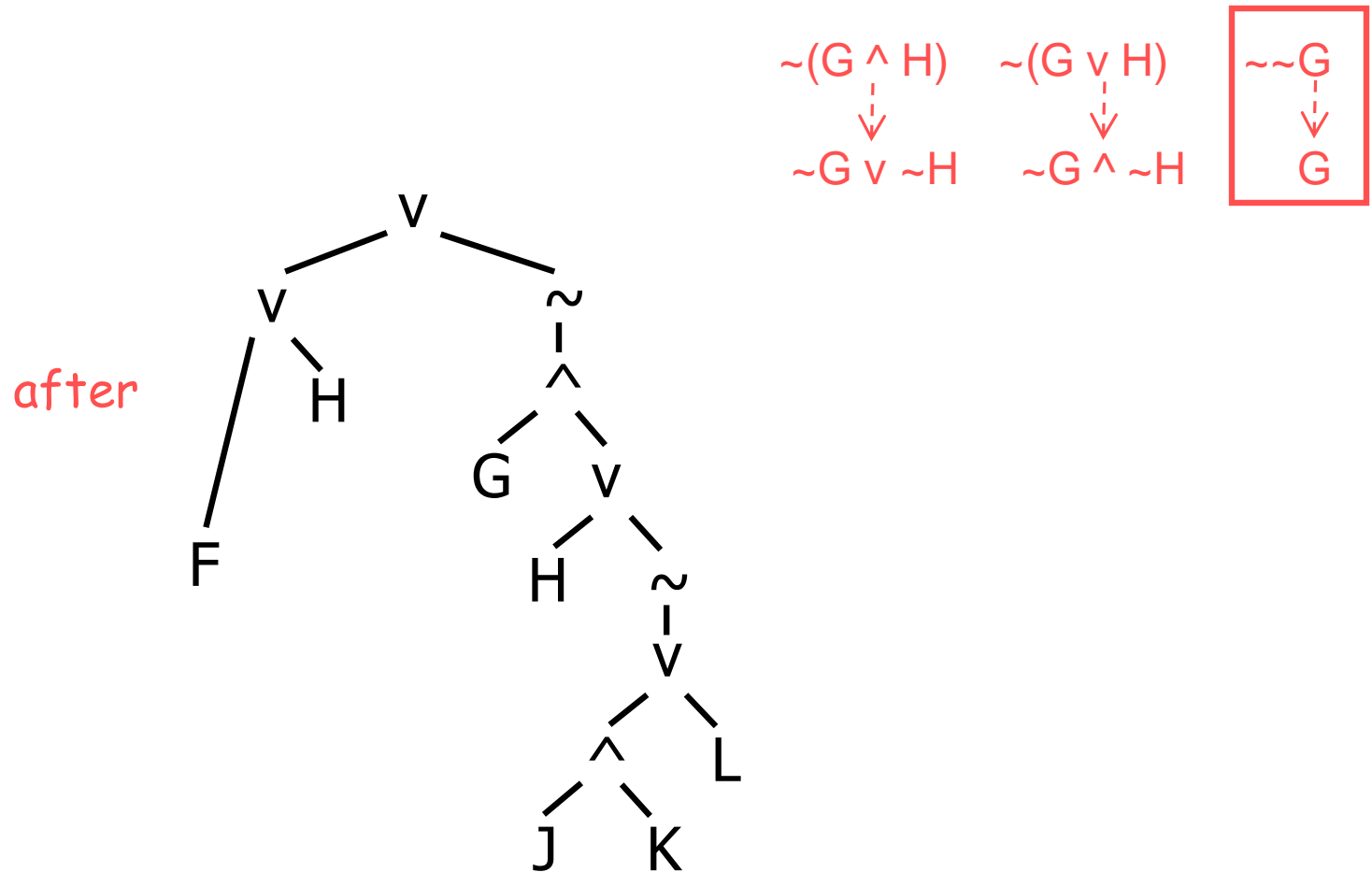




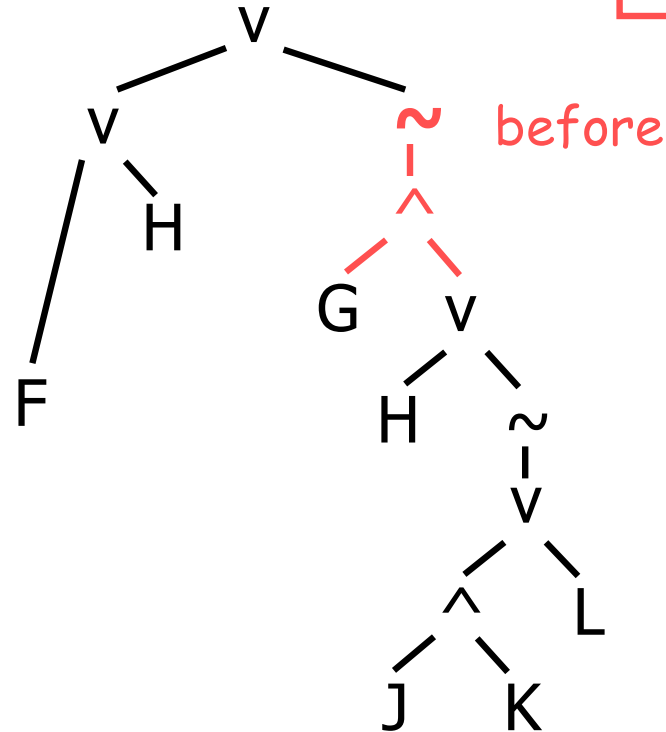
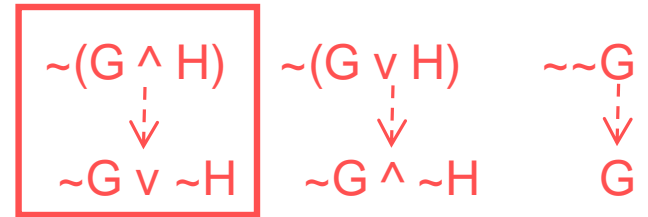
## Step 2: Push $\sim$ down to leaves



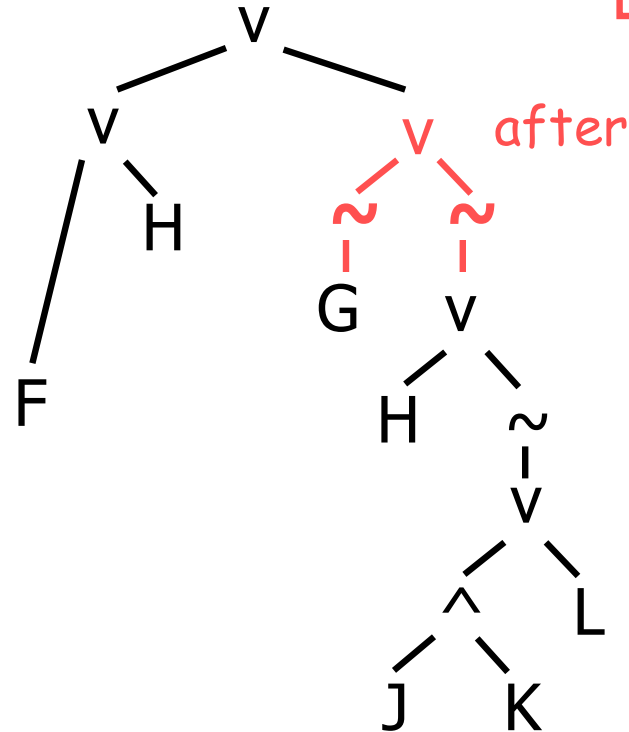
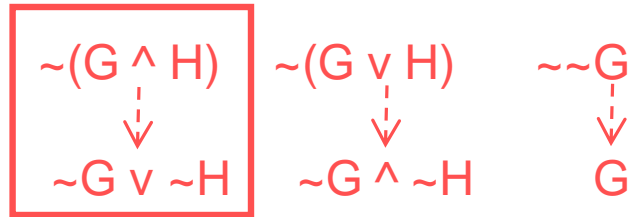
## Step 2: Push $\sim$ down to leaves



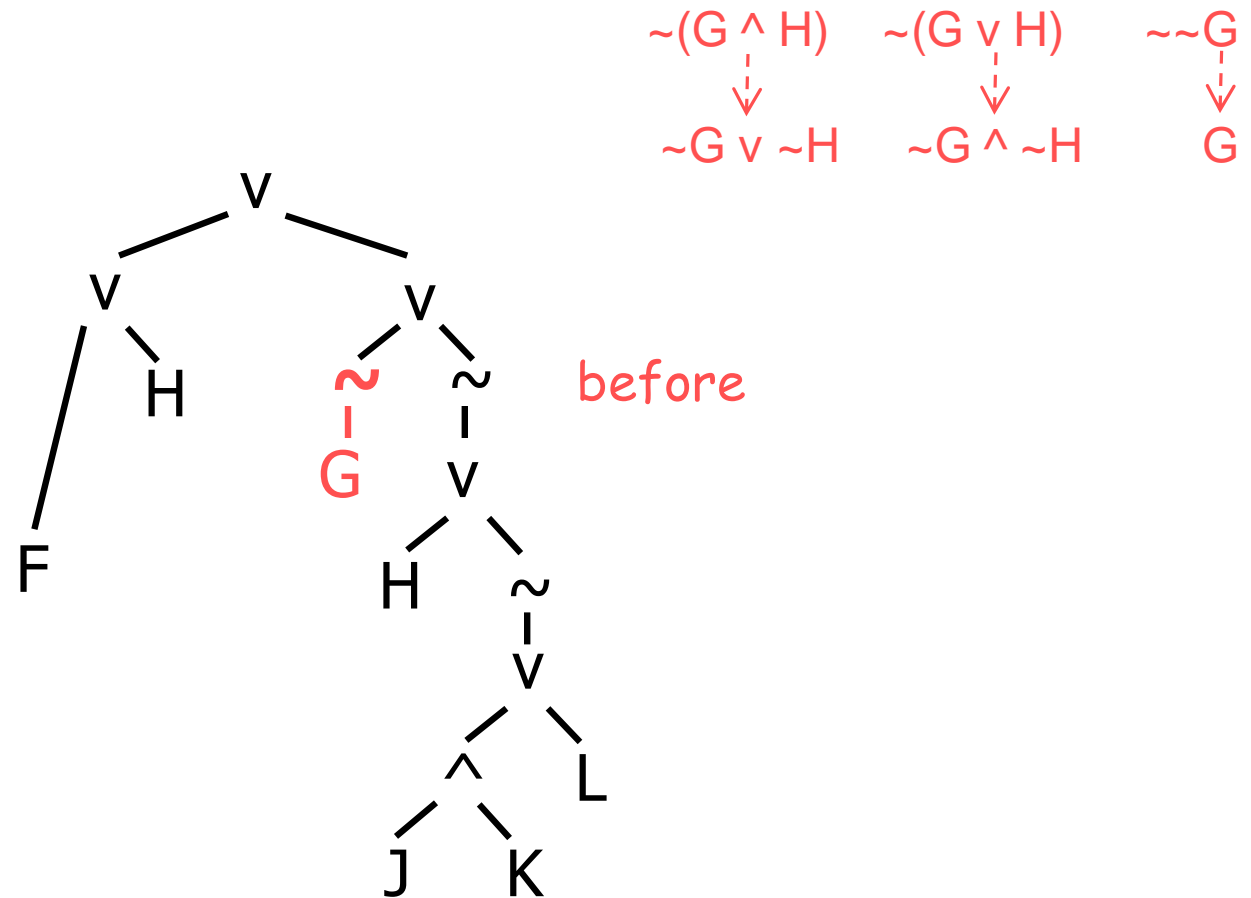
## Step 2: Push $\sim$ down to leaves



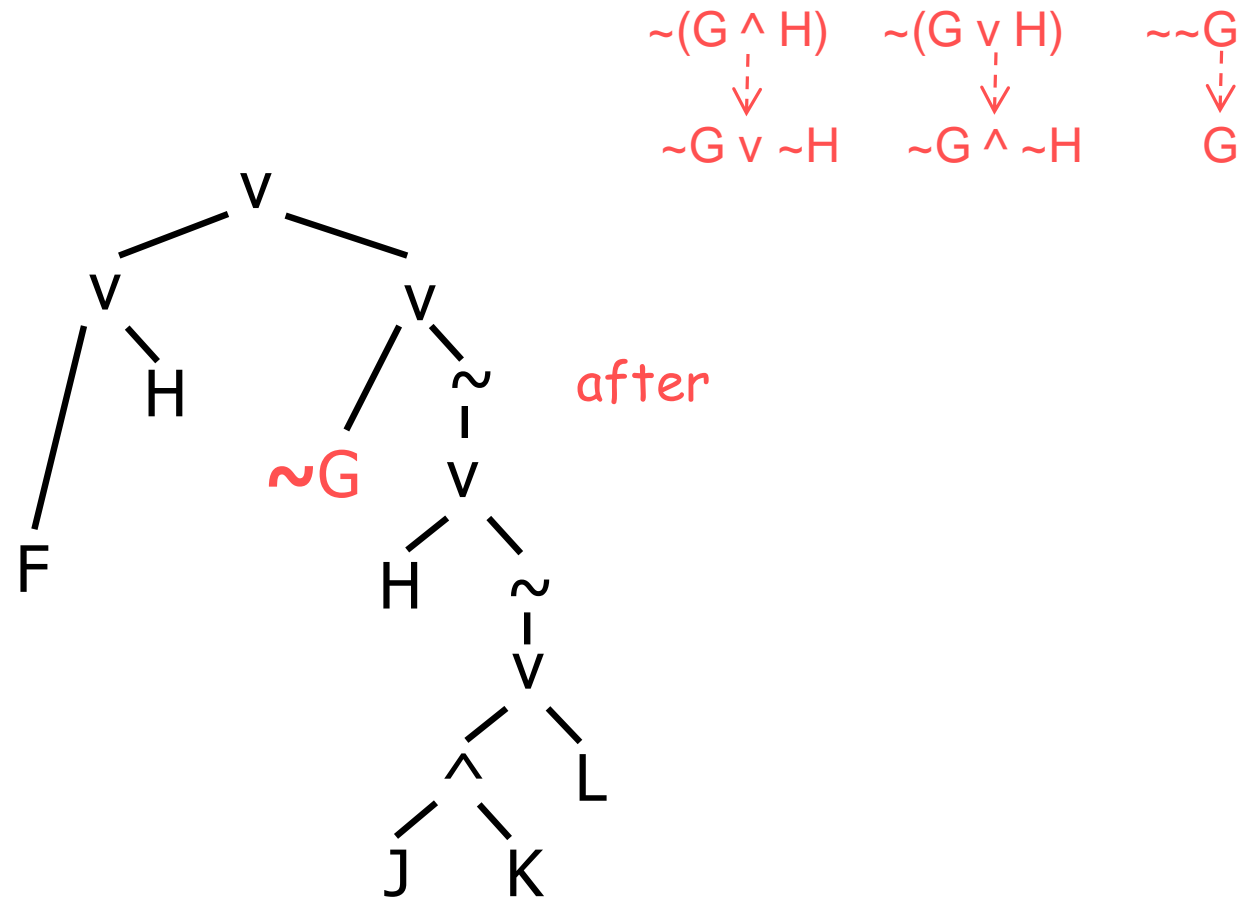
## Step 2: Push $\sim$ down to leaves



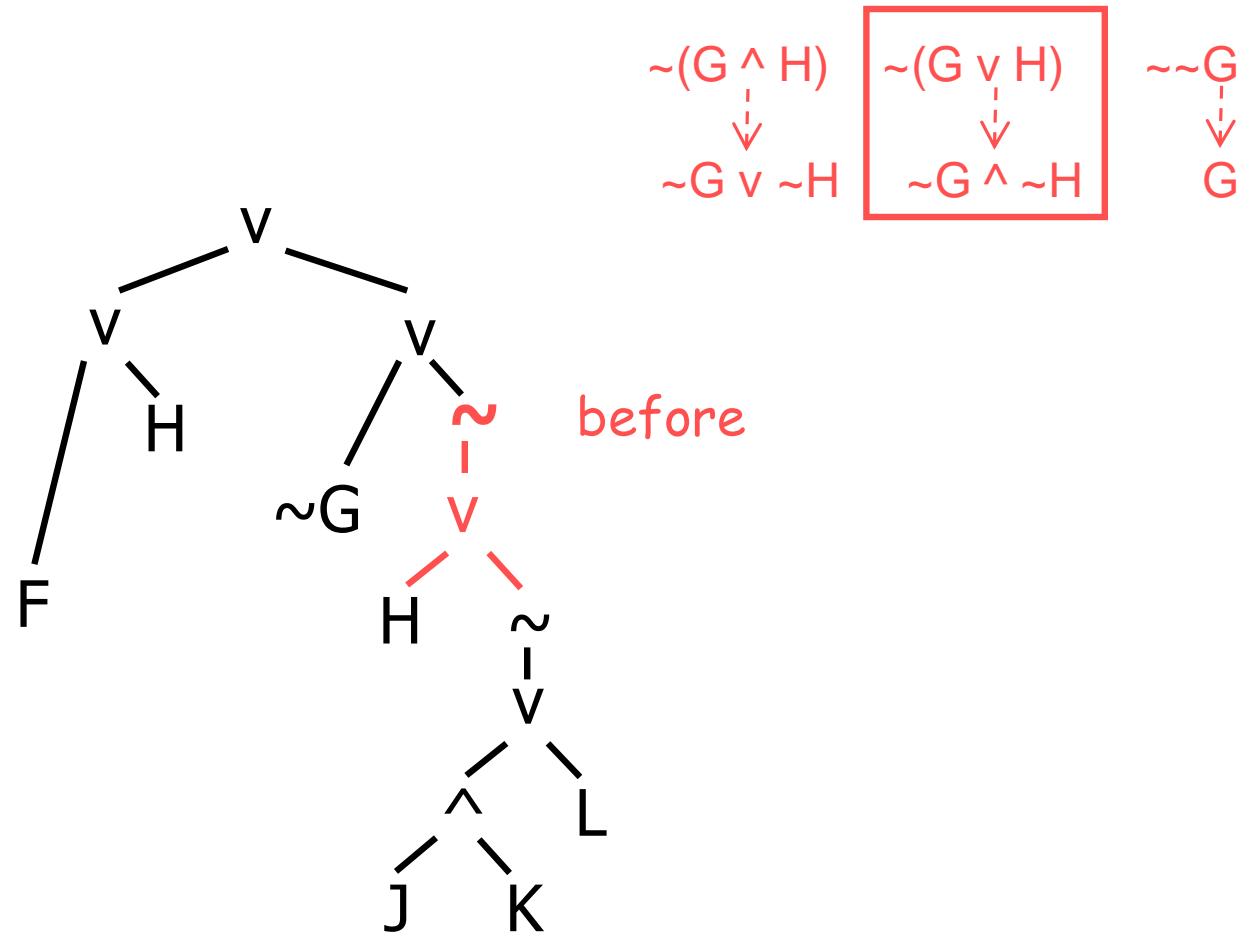
## Step 2: Push $\sim$ down to leaves



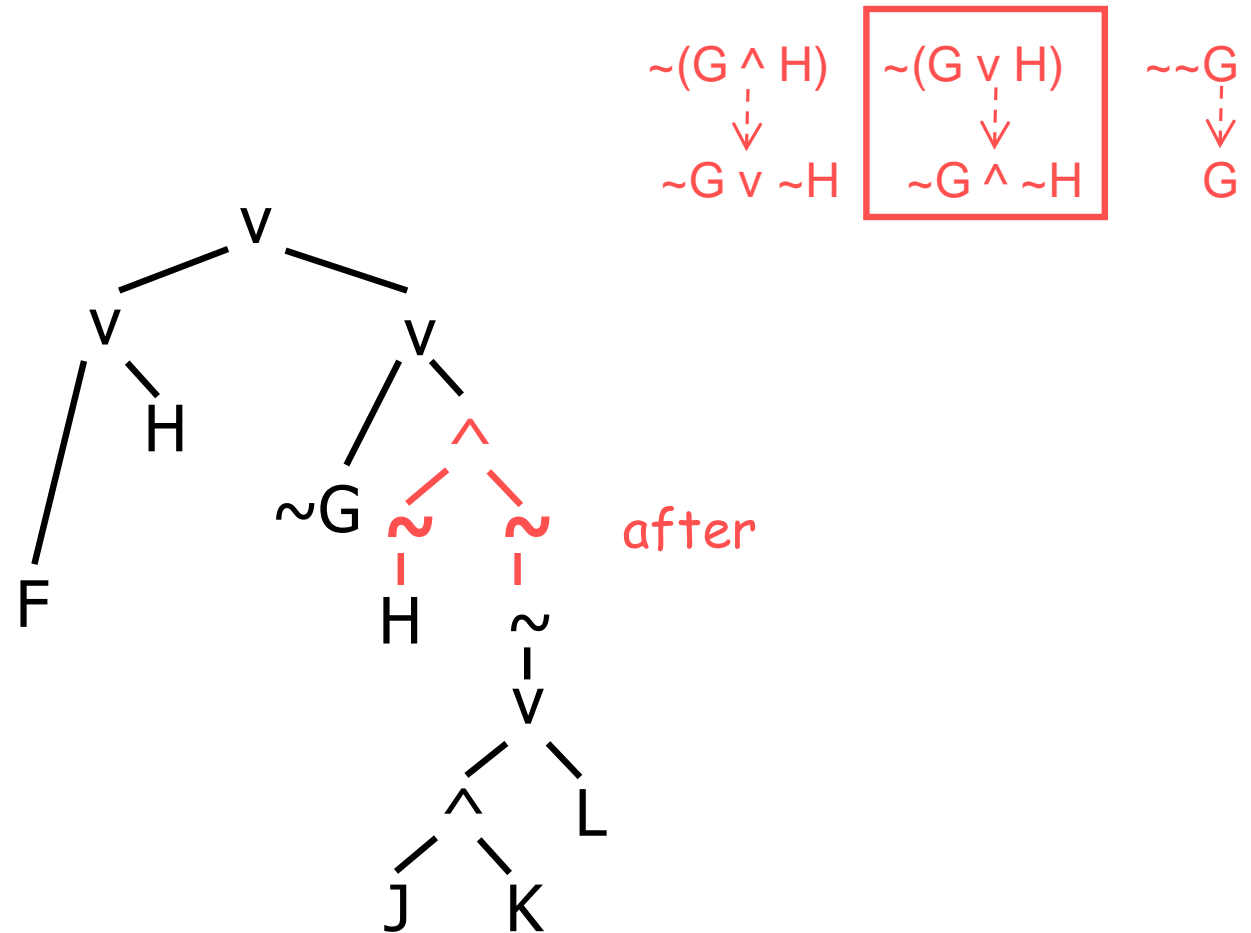
## Step 2: Push $\sim$ down to leaves



## Step 2: Push $\sim$ down to leaves

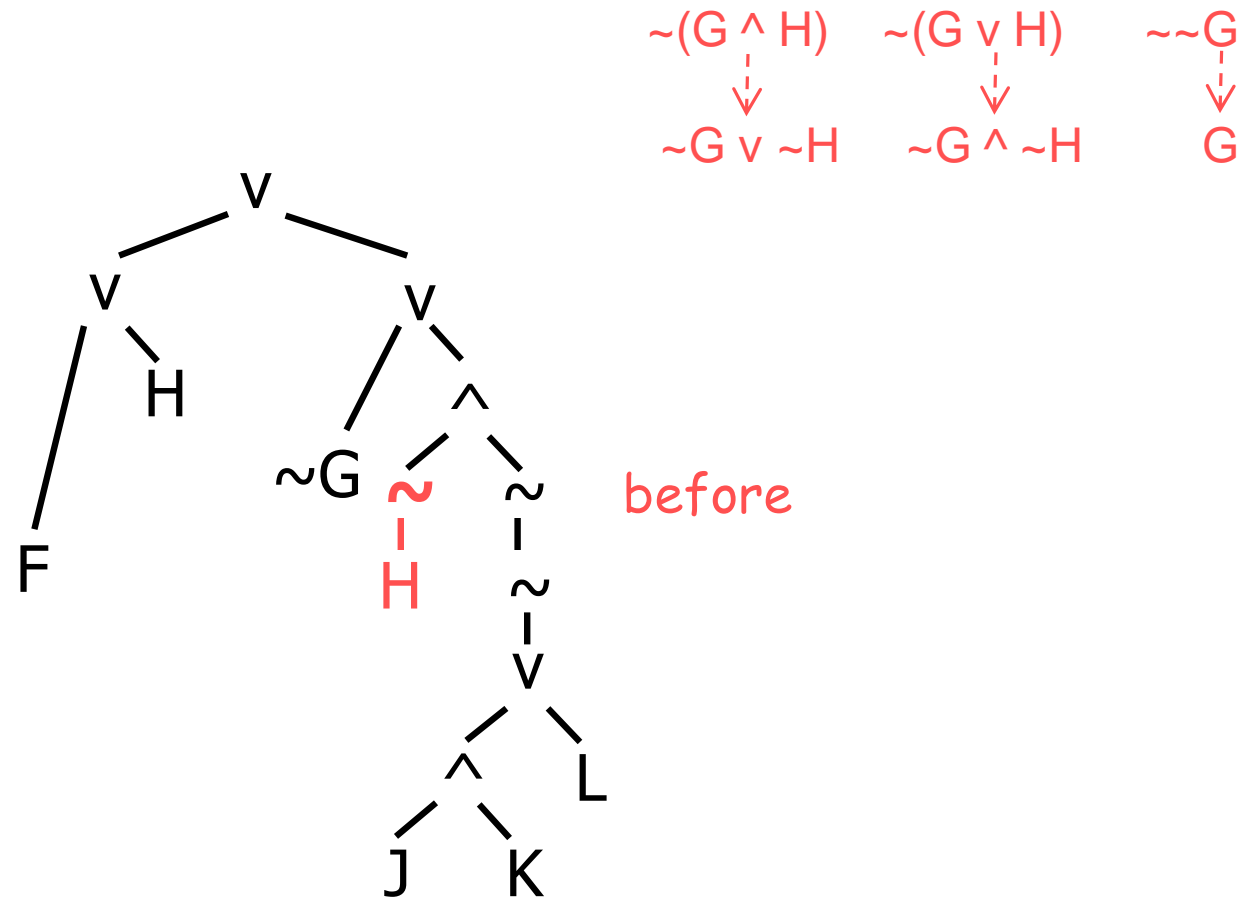


## Step 2: Push $\sim$ down to leaves

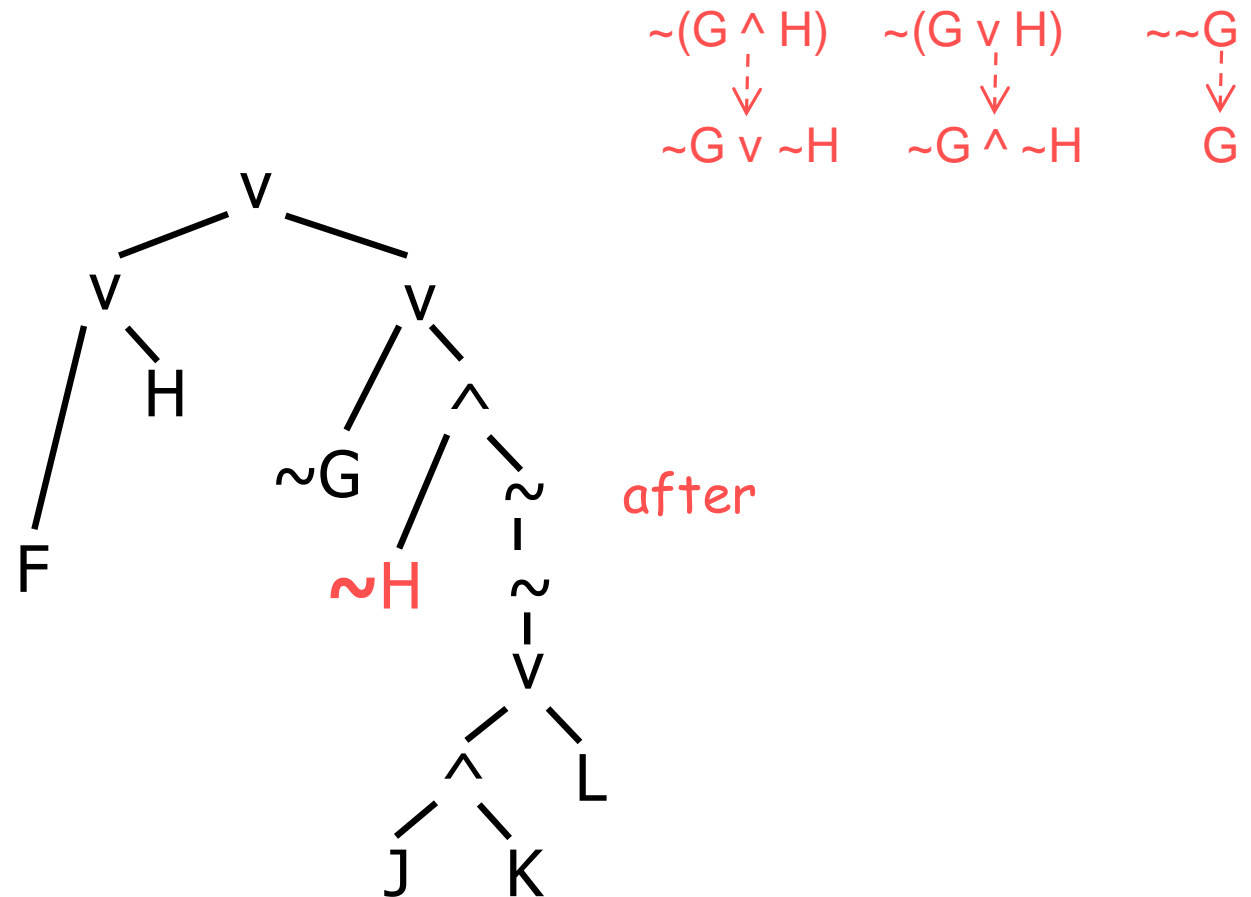




## Step 2: Push $\sim$ down to leaves



## Step 2: Push $\sim$ down to leaves



## Step 2: Push $\sim$ down to leaves

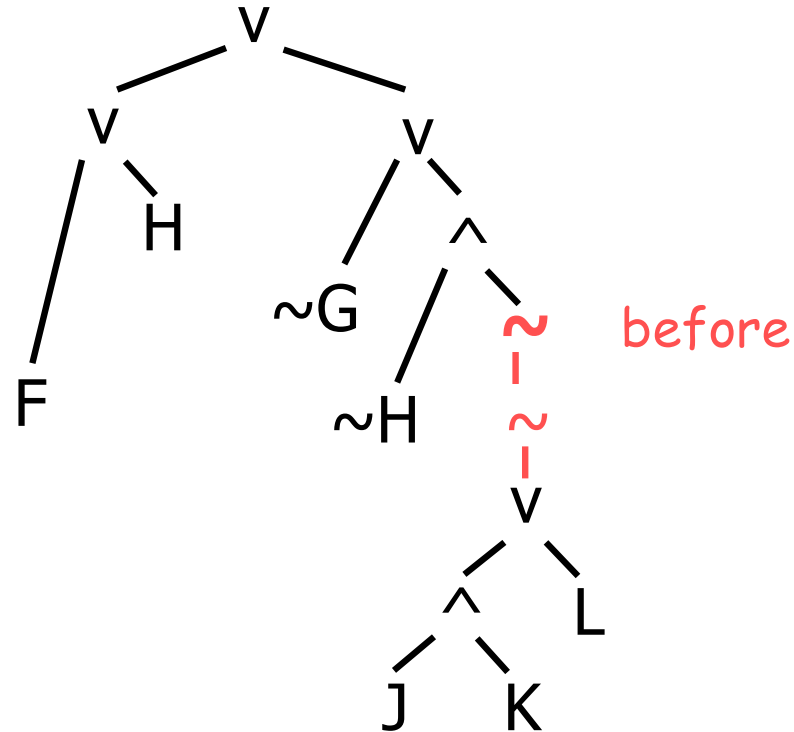
$$\sim(G \wedge H)$$

$$\downarrow$$
$$\sim G \vee \sim H$$

$$\sim(G \vee H)$$

$$\downarrow$$
$$\sim G \wedge \sim H$$

$$\sim \sim G$$
$$\downarrow$$
$$G$$



## Step 2: Push $\sim$ down to leaves

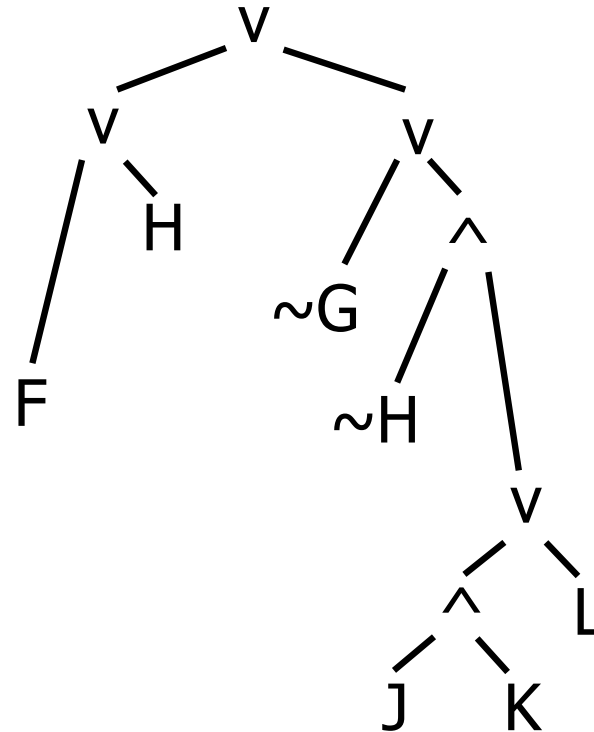
$$\sim(G \wedge H)$$

$$\downarrow$$
$$\sim G \vee \sim H$$

$$\sim(G \vee H)$$

$$\downarrow$$
$$\sim G \wedge \sim H$$

$$\sim\sim G$$
$$\downarrow$$
$$G$$

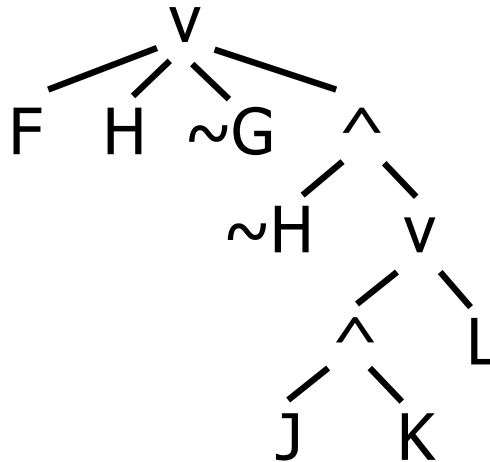


after

## Step 2: Push $\sim$ down to leaves

same tree redrawn

(For simplicity, we're now drawing root as  $v$  with 4 children. Used to be  $(F \vee H) \vee (G \vee \dots)$ , but  $F \vee H \vee G \vee \dots$  is okay since  $\vee$  is associative.)



$$\begin{array}{ccc} \sim(G \wedge H) & \sim(G \vee H) & \sim\sim G \\ \downarrow & \downarrow & \downarrow \\ \sim G \vee \sim H & \sim G \wedge \sim H & G \end{array}$$

- Now everything is  $\wedge$  and  $\vee$  over literals. Yay!

# Duality ...

So essentially the same: just swap the meanings of "true" and "false" (in both input and output)

Reduce green to brown:  
encode by negating inputs,  
decode by negating outputs

$$\begin{array}{ccc}
 \sim(G \wedge H) & \sim(G \vee H) & \sim\sim G \\
 \downarrow & \downarrow & \downarrow \\
 \sim G \vee \sim H & \sim G \wedge \sim H & G \\
 \underbrace{\hspace{10em}} & & \\
 \text{"de Morgan's laws"} & & 
 \end{array}$$

"Primal"	"Dual"	Reduction	Reverse reduction
$\vee$	$\wedge$	$F \wedge G = \sim(\sim F \vee \sim G)$	$F \vee G = \sim(\sim F \wedge \sim G)$
$\exists$	$\forall$	$\forall x \phi(x) = \sim \exists x \sim \phi(x)$	$\exists x \phi(x) = \sim \forall x \sim \phi(x)$
SAT	TAUT	$\text{TAUT}(\phi) = \sim \text{SAT}(\sim \phi)$	$\text{SAT}(\phi) = \sim \text{TAUT}(\sim \phi)$
NP	co-NP	$\forall y f(x,y) = \sim \exists y \sim f(x,y)$	$\exists y f(x,y) = \sim \forall y \sim f(x,y)$

F	G	$F \vee G$
0	0	0
0	1	1
1	0	1
1	1	1

Truth table for OR

Negate inputs  
and outputs

F	G	$F \wedge G$
1	1	1
1	0	0
0	1	0
0	0	0

Truth table for AND  
(listed upside-down from usual order)

# Duality ...

So essentially the same: just swap the meanings of "true" and "false" (in both input and output)

Reduce green to brown:  
encode by negating inputs,  
decode by negating outputs

$$\begin{array}{ccc}
 \sim(G \wedge H) & \sim(G \vee H) & \sim\sim G \\
 \downarrow & \downarrow & \downarrow \\
 \sim G \vee \sim H & \sim G \wedge \sim H & G \\
 \underbrace{\hspace{10em}} & & \\
 \text{"de Morgan's laws"} & & 
 \end{array}$$

"Primal"	"Dual"	Reduction	Reverse reduction
$\vee$	$\wedge$	$F \wedge G = \sim(\sim F \vee \sim G)$	$F \vee G = \sim(\sim F \wedge \sim G)$
$\exists$	$\forall$	$\forall x \phi(x) = \sim \exists x \sim \phi(x)$	$\exists x \phi(x) = \sim \forall x \sim \phi(x)$
SAT	TAUT	$\text{TAUT}(\phi) = \sim \text{SAT}(\sim \phi)$	$\text{SAT}(\phi) = \sim \text{TAUT}(\sim \phi)$
NP	co-NP	$\forall y f(x,y) = \sim \exists y \sim f(x,y)$	$\exists y f(x,y) = \sim \forall y \sim f(x,y)$

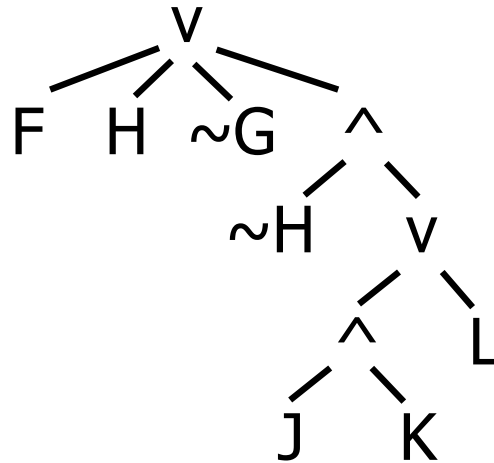
## At least one (not empty): like $\vee$

- $\exists x$  = "there exists x such that"
- SAT = is the formula satisfiable? (there exists a satisfying assignment)
- NP = generalization of SAT (problems that return **true** iff  $\exists y f(x,y)$ , where f can be computed in polynomial time  $O(|x|^k)$ )

## All (complement is not empty): like $\wedge$

- $\forall x$  = "all x are such that"
- TAUT = is the formula a tautology? (all assignments are satisfying)
- co-NP = generalization of TAUT (problems that return **true** iff  $\forall y f(x,y)$ , where f can be computed in polynomial time  $O(|x|^k)$ )

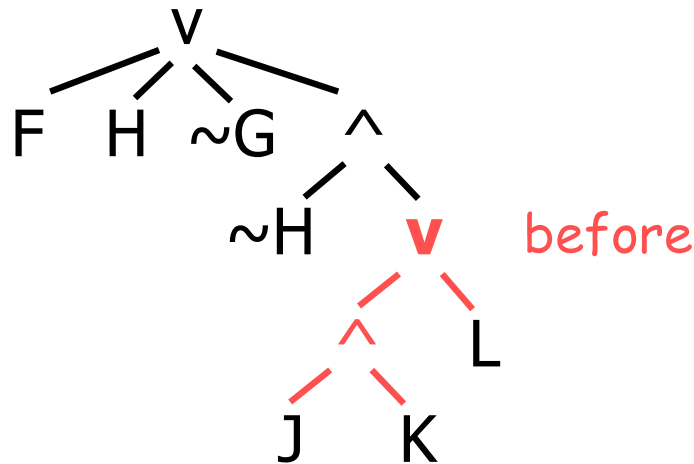
## Step 3: Push $v$ down below $\wedge$



- Now everything is  $\wedge$  and  $v$  over literals. Yay!
- Can we make it even simpler??
  - Yes. We can also push  $v$  down, to just above the leaves.
  - Then the whole tree will be  $\wedge$  at the root, over  $v$ , over  $\sim$  at the leaves.
  - That's called "conjunctive normal form" (CNF).

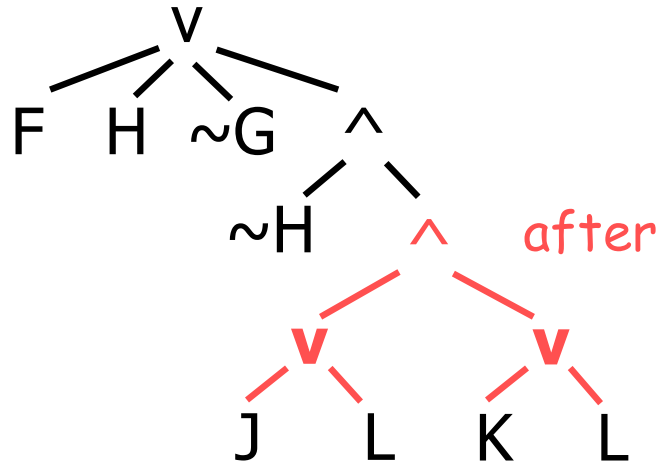


## Step 3: Push $v$ down below $\wedge$



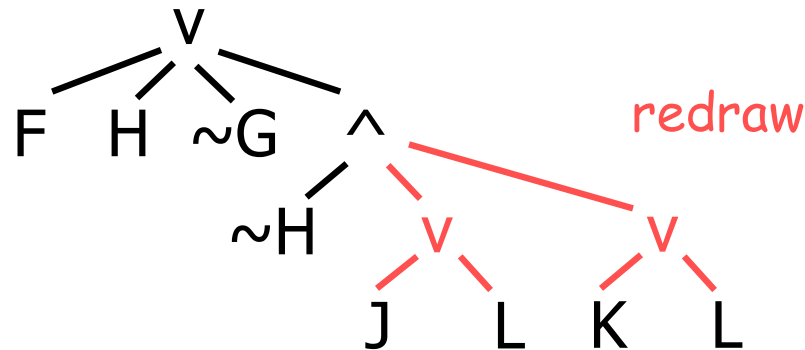
- Now everything is  $\wedge$  and  $v$  over literals. Yay!
- Can we make it even simpler??
  - Yes. We can also push  $v$  down, to just above the leaves.
  - Then the whole tree will be  $\wedge$  at the root, over  $v$ , over  $\sim$  at the leaves.
  - That's called "conjunctive normal form" (CNF).

## Step 3: Push $v$ down below $\wedge$



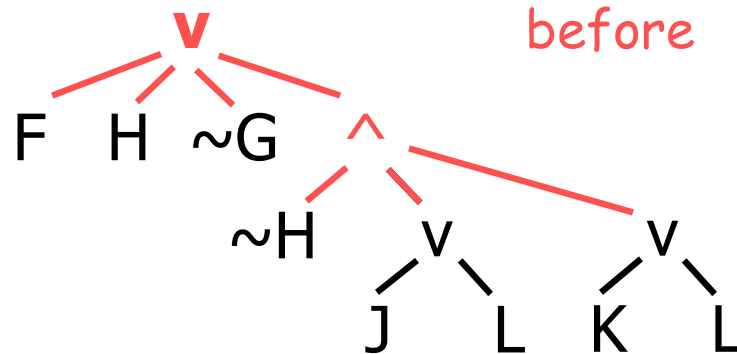
- Now everything is  $\wedge$  and  $v$  over literals. Yay!
- Can we make it even simpler??
  - Yes. We can also push  $v$  down, to just above the leaves.
  - Then the whole tree will be  $\wedge$  at the root, over  $v$ , over  $\sim$  at the leaves.
  - That's called "conjunctive normal form" (CNF).

## Step 3: Push $v$ down below $\wedge$



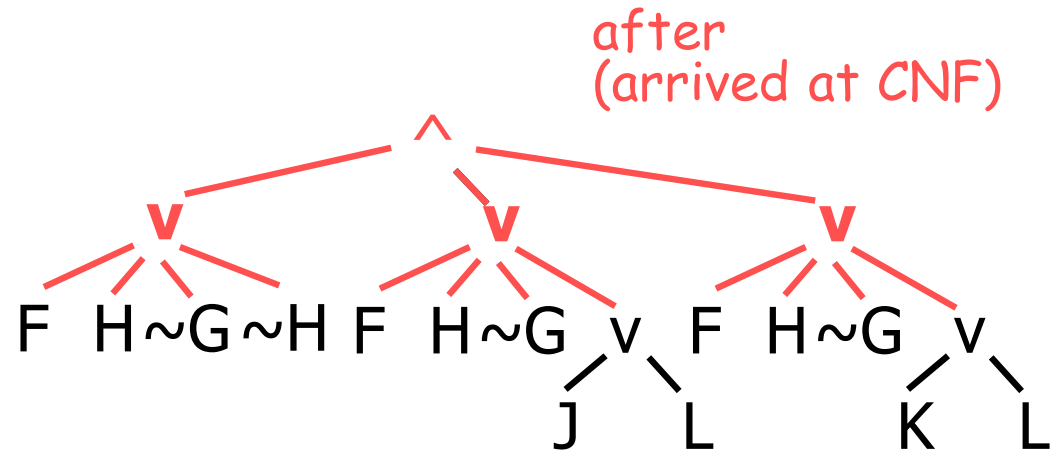
- Now everything is  $\wedge$  and  $v$  over literals. Yay!
- Can we make it even simpler??
  - Yes. We can also push  $v$  down, to just above the leaves.
  - Then the whole tree will be  $\wedge$  at the root, over  $v$ , over  $\sim$  at the leaves.
  - That's called "conjunctive normal form" (CNF).

## Step 3: Push $v$ down below $\wedge$



- Now everything is  $\wedge$  and  $v$  over literals. Yay!
- Can we make it even simpler??
  - Yes. We can also push  $v$  down, to just above the leaves.
  - Then the whole tree will be  $\wedge$  at the root, over  $v$ , over  $\sim$  at the leaves.
  - That's called "conjunctive normal form" (CNF).

## Step 3: Push $v$ down below $\wedge$



- Now everything is  $\wedge$  and  $\vee$  over literals. Yay!
- Can we make it even simpler??
  - Yes. We can also push  $\vee$  down, to just above the leaves.
  - Then the whole tree will be  $\wedge$  at the root, over  $\vee$ , over  $\sim$  at the leaves.
  - That's called "conjunctive normal form" (CNF).

# Simpler formulas, simpler solvers

## ■ Conjunctive normal form (CNF):

- Conjunction of disjunctions of literals
- $(A \vee B \vee \sim C) \wedge (\sim B \vee D \vee E) \wedge (\sim A \vee \sim D) \wedge \sim F \dots$   
clauses
- Can turn any formula into CNF
- So all we need is a CNF solver (still NP-complete: no free lunch)

## ■ Disjunctive normal form (DNF):

- Disjunction of conjunctions of literals
- $(A \wedge B \wedge \sim C) \vee (\sim B \wedge D \wedge E) \vee (\sim A \wedge \sim D) \vee \sim F \dots$
- Can turn any formula into DNF
- So all we need is a DNF solver (still NP-complete: no free lunch)

Wait a minute!  
Something wrong.  
Can you see a  
fast algorithm?

# What goes wrong with DNF

- Satisfiability on a DNF formula takes only linear time:

$$(A \wedge B \wedge \sim C) \vee (\sim B \wedge D \wedge E) \vee (\sim A \wedge \sim D) \vee \sim F \dots$$

- Proof that we can convert any formula to DNF

- First convert to  $\vee/\wedge$  form, then do it recursively:

- **Base case:** Literals don't need to be converted
- **To convert  $A \vee B$ ,** first convert A and B individually

- $(A_1 \vee A_2 \vee \dots A_n) \vee (B_1 \vee B_2 \vee \dots B_m)$

- $= A_1 \vee A_2 \vee \dots \vee A_n \vee B_1 \vee B_2 \vee \dots \vee B_m$

Because  $\wedge$  distributes over  $\vee$ , just as  $*$  distributes over  $+$ :

$$(a_1 + a_2 + \dots + a_n) * (b_1 + b_2 + \dots + b_m)$$

$$= a_1 b_1 + a_1 b_2 + \dots + a_n b_m$$

- **To convert  $A \wedge B$ ,** first convert A and B individually

- $(A_1 \vee A_2 \vee \dots \vee A_n) \wedge (B_1 \vee B_2 \vee \dots \vee B_m)$

- $= (A_1 \wedge B_1) \vee (A_1 \wedge B_2) \vee \dots \vee (A_2 \wedge B_1) \vee (A_2 \wedge B_2) \vee \dots (A_n \wedge B_m)$

- Hmm, this is quadratic blowup. What happens to  $A \wedge B \wedge C \dots$  ?

- $(A_1 \vee A_2) \wedge (B_1 \vee B_2) \wedge (C_1 \vee C_2) \wedge \dots = \text{what?}$

- Exponential blowup – really just generate and test of all combinations

- So it doesn't help that we have a linear DNF solver

- It's like having a “linear-time” factoring algorithm where the input has to be in unary notation: 2010 encoded as “1111111111111111...”

- Or more precisely, a SAT algorithm where input is a truth table! ( $*$  allowed)

# Just the same thing goes wrong with CNF

- DNF and CNF seem totally symmetric: just swap  $\vee$ ,  $\wedge$ 
    - You can convert to CNF or DNF by essentially same algorithm
    - DNF blows up when you try to convert  $A \wedge B \wedge C \dots$
    - CNF blows up when you try to convert  $A \vee B \vee C \dots$
  - *But there is another way to convert to CNF!* (Tseitin 1970)
    - Only quadratic blowup after we've reduced to  $\vee$ ,  $\wedge$ ,  $\sim$
    - Idea: introduce a new “switching variable” for each  $v$
- $(B_1 \wedge B_2 \wedge B_3) \vee (C_1 \wedge C_2 \wedge C_3)$  this is satisfiable iff ...
- $= (Z \rightarrow (B_1 \wedge B_2 \wedge B_3)) \wedge (\sim Z \rightarrow (C_1 \wedge C_2 \wedge C_3))$  ... this is satisfiable
- $= (\sim Z \vee (B_1 \wedge B_2 \wedge B_3)) \wedge (Z \vee (C_1 \wedge C_2 \wedge C_3))$  (solver picks Z)
- distribute  $\vee$  over  $\wedge$  as before, but gives 3+3 clauses, not  $3 \times 3$
- $= (\sim Z \vee B_1) \wedge (\sim Z \vee B_2) \wedge (\sim Z \vee B_3) \wedge (Z \vee C_1) \wedge (Z \vee C_2) \wedge (Z \vee C_3)$



# Efficiently encode any formula as CNF

- We rewrote  $(B_1 \wedge B_2 \wedge B_3) \vee (C_1 \wedge C_2 \wedge C_3)$   
 $= (\sim Z \vee B_1) \wedge (\sim Z \vee B_2) \wedge (\sim Z \vee B_3) \wedge (Z \vee C_1) \wedge (Z \vee C_2) \wedge (Z \vee C_3)$
- Recursively eliminate  $\vee$  from  $(A_1 \wedge A_2) \vee ((B_1 \wedge B_2) \vee (C_1 \wedge C_2))$  :  
 $= (A_1 \wedge A_2) \vee ((\sim Z \vee B_1) \wedge (\sim Z \vee B_2) \wedge (Z \vee C_1) \wedge (Z \vee C_2))$  *switching variable Z as before*  
*switching variable Y*  
 $= (\sim Y \vee A_1) \wedge (\sim Y \vee A_2) \wedge (Y \vee \sim Z \vee B_1) \wedge (Y \vee \sim Z \vee B_2) \wedge (Y \vee Z \vee C_1) \wedge (Y \vee Z \vee C_2)$
- Input formula suffers at worst quadratic blowup. Why? Each of its literals simply becomes a clause with the switching variables that affect it:
- For example, we start with a copy of  $B_1$  that falls in the 2nd arg of the  $Y \vee$  and the 1st arg of the  $Z \vee$ . So it turns into a clause  $(Y \vee \sim Z \vee B_1)$ .

# Efficiently encode any formula as CNF

- We rewrote  $(B_1 \wedge B_2 \wedge B_3) \vee (C_1 \wedge C_2 \wedge C_3)$   
 $= (\sim Z \vee B_1) \wedge (\sim Z \vee B_2) \wedge (\sim Z \vee B_3) \wedge (Z \vee C_1) \wedge (Z \vee C_2) \wedge (Z \vee C_3)$
- Recursively eliminate  $\vee$  from  $(A_1 \wedge A_2) \vee ((B_1 \wedge B_2) \vee (C_1 \wedge C_2)) \wedge D$  :  
 $= (A_1 \wedge A_2) \vee ((\sim Z \vee B_1) \wedge (\sim Z \vee B_2) \wedge (Z \vee C_1) \wedge (Z \vee C_2) \wedge D)$  *switching variable Z as before*  
 $= (\sim Y \vee A_1) \wedge (\sim Y \vee A_2) \wedge (Y \vee \sim Z \vee B_1) \wedge (Y \vee \sim Z \vee B_2) \wedge (Y \vee Z \vee C_1) \wedge (Y \vee Z \vee C_2) \wedge (Y \vee D)$  *switching variable Y*
- Input formula suffers at worst quadratic blowup. Why? Each of its literals simply becomes a clause with the switching variables that affect it:
- For example, we start with a copy of  $B_1$  that falls in the 2nd arg of the  $Y \vee$  and the 1st arg of the  $Z \vee$ . So it turns into a clause  $(Y \vee \sim Z \vee B_1)$ .

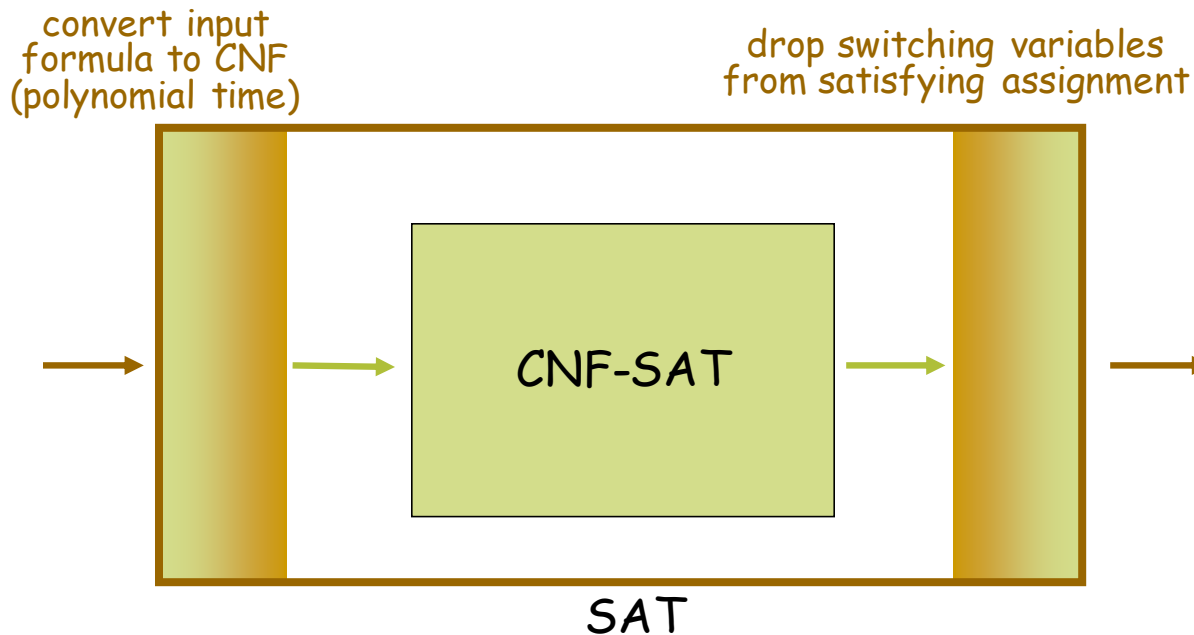
# So why can't we use this trick for DNF?

- We rewrote  $(B_1 \wedge B_2 \wedge B_3) \vee (C_1 \wedge C_2 \wedge C_3)$  as a “short” CNF formula:  
$$= (\sim Z \vee B_1) \wedge (\sim Z \vee B_2) \wedge (\sim Z \vee B_3) \wedge (Z \vee C_1) \wedge (Z \vee C_2) \wedge (Z \vee C_3)$$
- But we can just switch  $\vee$  and  $\wedge$  – they're symmetric!
- So why not rewrite  $(B_1 \vee B_2 \vee B_3) \wedge (C_1 \vee C_2 \vee C_3)$  as this “short” DNF?  
$$= (\sim Z \wedge B_1) \vee (\sim Z \wedge B_2) \vee (\sim Z \wedge B_3) \vee (Z \wedge C_1) \vee (Z \wedge C_2) \vee (Z \wedge C_3)$$
  - Because we'd get a polytime SAT solver and win the Turing Award.
  - And because the rewrite is clearly wrong. It yields something easy to satisfy.
- The CNF/DNF difference is because we introduced extra variables.
- In CNF, original formula was satisfiable if new formula is satisfiable for **either**  $Z=\text{true}$  **or**  $Z=\text{false}$ . But wait! We're trying to switch **or** and **and**:
- In DNF, original formula is satisfiable if new formula is satisfiable for **both**  $Z=\text{true}$  **and**  $Z=\text{false}$ . Look at the formulas above and see it's so!
  - Alas, that's not what a SAT checker checks. We'd have to call it many times (once per assignment to the switching variables  $Y, Z, \dots$ ).

# CNF and DNF are duals ...

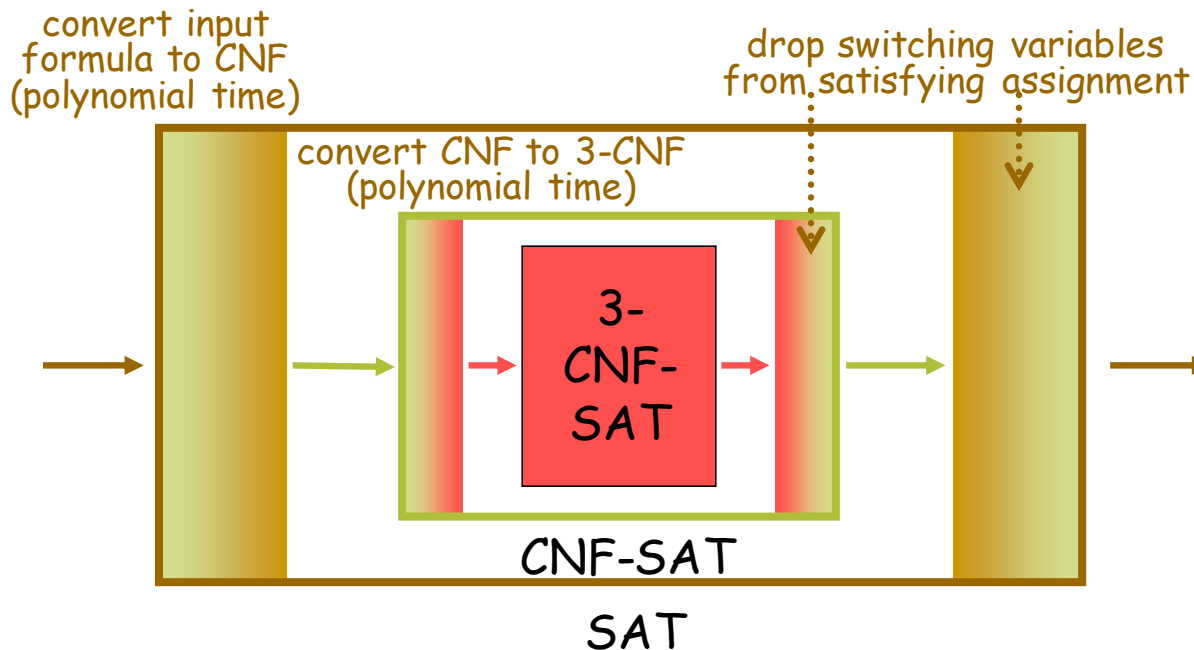
"Primal"	"Dual"	Reduction	Reverse reduction
$\vee$	$\wedge$	$F \wedge G = \sim(\sim F \vee \sim G)$	$F \vee G = \sim(\sim F \wedge \sim G)$
$\exists$	$\forall$	$\forall x \phi(x) = \sim \exists x \sim \phi(x)$	$\exists x \phi(x) = \sim \forall x \sim \phi(x)$
SAT	TAUT	$\text{TAUT}(\phi) = \sim \text{SAT}(\sim \phi)$	$\text{SAT}(\phi) = \sim \text{TAUT}(\sim \phi)$
NP	co-NP	$\forall y f(x,y) = \sim \exists y \sim f(x,y)$	$\exists y f(x,y) = \sim \forall y \sim f(x,y)$
CNF-SAT	DNF-TAUT	can reduce problem efficiently to this form, using switching vars, <b>but</b> of course it's still hard to solve	
DNF-SAT	CNF-TAUT	problem in this form can be solved in linear time, <b>but</b> reducing to this form can blow up the size exponentially since switching variable trick can no longer be used	

# So we've reduced SAT to CNF-SAT



- Most SAT solvers are actually CNF-SAT solvers.
  - They make you enter a CNF formula.
  - To use an arbitrary formula, you have to convert it to CNF yourself.
  - Fortunately, many practical problems like the LSAT puzzles are *naturally* expressed as something close to CNF. (Convert each fact or constraint to CNF and conjoin them all: the result is still in CNF.)

# From CNF-SAT to 3-CNF-SAT



- SAT solvers could be even more annoyingly restrictive:
  - They could require each CNF clause to have at most 3 literals.
  - When converting your input formula to CNF, you'd have to get rid of “long” clauses like  $(A_1 \vee A_2 \vee A_3 \vee A_4)$ .
  - This conversion is easy, so 3-CNF-SAT is still hard (NP-complete).

# From CNF-SAT to 3-CNF-SAT

- How do we convert CNF to 3-CNF?
- Again replace “v” with a switching variable.
  - Formerly did that to fix non-CNF:  $(A_1 \wedge A_2) \vee (B_1 \wedge B_2)$
  - Now do it to fix long clause:  $A_1 \vee A_2 \vee A_3 \vee A_4$

$$A_1 \vee A_2 \vee \dots A_{n-2} \vee A_{n-1} \vee A_n$$

$$(Z \rightarrow A_1 \vee A_2 \vee \dots A_{n-2}) \wedge (\sim Z \rightarrow A_{n-1} \vee A_n)$$

$$(A_1 \vee A_2 \vee \dots A_{n-2} \vee \sim Z) \wedge (A_{n-1} \vee A_n \vee Z)$$

reduced from length n to  
length n-1; now recurse!

length 3 as desired

# From CNF-SAT to 3-CNF-SAT

- Why can't we get down to 2-CNF-SAT?
  - Actually 2-CNF-SAT can be solved in polynomial time.
  - So if we could convert any formula to 2-CNF-SAT in polynomial time, we'd have a proof that  $P = NP$ .

So it would be surprising if we could use switching variables to get to 2-CNF-SAT:

$$A_1 \vee A_2 \vee A_3$$

$$(Z \rightarrow A_1) \wedge (\sim Z \rightarrow A_2 \vee A_3)$$

$$(A_1 \vee \sim Z) \wedge (A_2 \vee A_3 \vee Z)$$



Yay, down to length 2



Oops, still has length 3



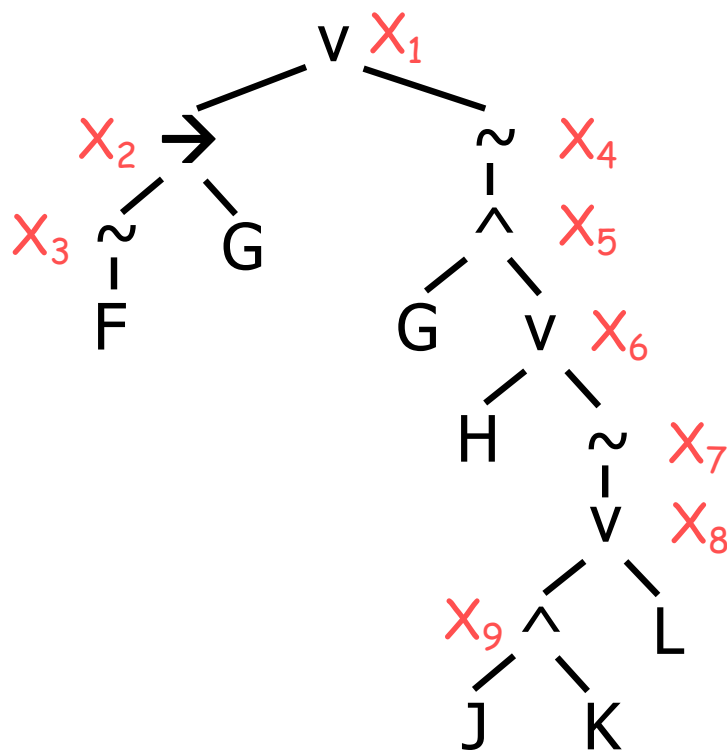
# The Tseitin Transformation (1968)

- Alternative way to convert any formula directly to 3-CNF.
- Disadvantages:
  - Introduces even more extra variables, and sometimes more clauses.
  - If the original formula is *already* in 3-CNF, will make it bigger.
- Advantages:
  - Simple, general, elegant.
  - Gets only *linear* blowup.
    - That is, the CNF formula has length  $O(k)$  where  $k$  is the length of the original formula.
  - Linear even if the original formula contains  $\leftrightarrow$ , **xor**.
    - Our old method of eliminating those could get exponential blowup.
    - Why? Converted  $F \leftrightarrow G$  to  $(F \rightarrow G) \wedge (G \rightarrow F)$ , doubling the length.
    - And  $F$ ,  $G$  could contain  $\leftrightarrow$  themselves, hence repeated doubling.

# The Tseitin Transformation (1968)

Associate a new variable with each internal node (operator) in the formula tree

Constrain each new variable to have the appropriate value given its children



$$(X_1 \leftrightarrow (X_2 \vee X_4))$$

$$\wedge (X_2 \leftrightarrow (X_3 \rightarrow G)) \quad \wedge (X_4 \leftrightarrow (\sim X_5))$$

$$\wedge (X_3 \leftrightarrow (\sim F)) \quad \wedge (X_5 \leftrightarrow (G \wedge X_6))$$

$$\wedge (X_6 \leftrightarrow (H \vee X_7))$$

$$\wedge (X_7 \leftrightarrow (\sim X_8))$$

$$\wedge (X_8 \leftrightarrow (X_9 \vee L))$$

$$\wedge (X_9 \leftrightarrow (J \wedge K))$$

$$\wedge X_1$$

Require the whole formula to have value true!

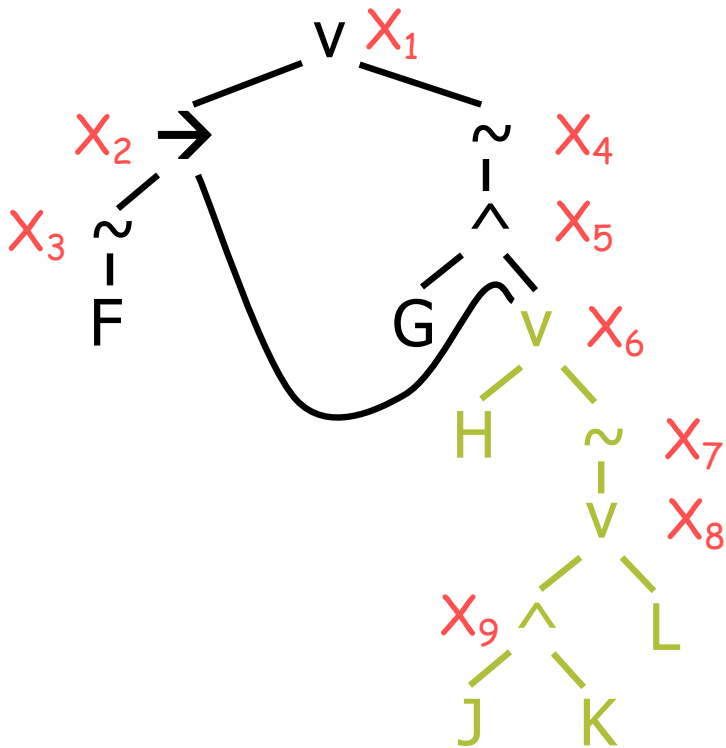
Now separately convert each of these short clauses to 3-CNF.

E.g.,  $(X_1 \leftrightarrow (X_2 \vee X_4))$  becomes  $(\sim X_1 \vee X_2 \vee X_4) \wedge (X_1 \vee X_2) \wedge (X_1 \vee X_4)$

# The Tseitin Transformation (1968)

Associate a new variable with each internal node (operator) in the formula tree

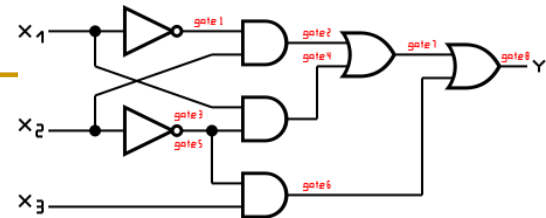
# DAG



You can build such a DAG out of logic gates. So it's called a "Boolean circuit":

$$\begin{array}{c} (\sim F \rightarrow (H \vee \sim((J \wedge K) \vee L))) \\ \vee \\ \sim(G \wedge (H \vee \sim((J \wedge K) \vee L))) \end{array}$$

- The circuit allows **shared subformulas**, so can be much smaller than the formula written out without sharing.
- Tseitin transformation will reuse the work on each subformula!
- Each node in the DAG gives rise to at most four 3-clauses.
- So the final 3-CNF formula has size that's linear in the size of the input DAG.



# Quantified Satisfiability (QSAT)

- SAT asks whether  $(\exists a) F(a)$        $a$  is an assignment to all vars
- TAUT asks whether  $(\forall a) F(a)$        $a$  is an assignment to all vars
- They're the standard NP-complete and co-NP-complete problems.
  
- QSAT lets you ask, for example, whether  
 $(\exists a) (\forall b) (\exists c) (\forall d) F(a,b,c,d)$        $a,b,c,d$  are assignments  
to non-overlapping subsets  
of the vars
- Harder! Worse than NP-complete (outside both NP and co-NP).
  
- QSAT problems are the standard complete problems for the problem classes higher up in the polynomial hierarchy.
- **Example:** Can White force a win in 4 moves? That is: Is there an opening move for you  $(\exists a)$  such that for whatever response Kasparov makes  $(\forall b)$ , you will have some reply  $(\exists c)$  so that however Kasparov moves next  $(\forall d)$ , you've checkmated him in a legal game  $(F(a,b,c,d))$ ?