Lecture 10

Pete Manolios
Northeastern

DP SAT Algorithm

- Davis Putnam (1960)
- Input: CNF formula
- Output: SAT/UNSAT
- Idea: apply three rules until
  - Derive the empty clause: UNSAT (identity of $\vee$ is false)
  - No clauses remain: SAT (identity of $\wedge$ is true)
- Three “rules”
  - Pure literal rule (affirmative-negative rule)
  - Unit resolution rule (unit propagation, BCP, 1-literal rule)
  - Resolution (Called consensus, also used for logic minimization)

Pure Literal Rule

- Given F, a set of clauses, and literal ℓ such
  - ℓ appears in F
  - $\neg\ell$ does not appear in F
  - remove all clauses containing ℓ
- Equisatisfiable because we can make ℓ true
- Notice that this always simplifies F
- Modern SAT solvers tend to not use the rule (efficiency)

Boolean Constraint Propagation

Unit resolution rule:

$$\frac{C, \neg\ell \qquad \ell}{C}$$

- BCP: given a set of clauses including $\{\ell\}$
  - remove all other clauses containing ℓ (subsumption)
  - remove all occurrences of $\neg\ell$ in clauses (unit resolution)
  - repeat until a fixpoint is reached

Resolution

Resolution rule:

$$\frac{C, v \qquad D, \neg v}{C, D} \quad \neg v, v \notin C, D$$

Resolution rule:

$$\frac{C_i, p \qquad D_i, \neg p}{C_i, D_i} \quad \neg p \notin C_i \in P, p \notin D_i \in N$$

- Soundness of rule: above line implies below line
- If below line is SAT, so is above line (w/ side conditions)
- Given literal *p*, set of clauses *S*, let *P* be the clauses in *S* that contain *p* only positively and let *N* be the clauses that contain *p* only negatively. Let *E* be the rest of the clauses. Then *S* is SAT iff *S'* is SAT, where *S'*= *E* U the set of all *p*-resolvents of *P* and *N*.
- Proof: If *A* is an assignment for *S*, then if *A*(*p*)=true, all clauses in *N*, with ¬*p* removed are satisfied, so each *p*-resolvent is satisfied. Similarly if *A*(*p*)=false. If *A* is an assignment for *S*', then it satisfies all *Ci* or all *Di*: suppose it doesn't satisfy *Ck*, then it must satisfy all *Di*. If it satisfies all *Ci*, let *A*'(*p*)=false, else *A*'(*p*)=true and *A*'(*x*)=*A*(*x*) otherwise.

Resolution Example

Resolution rule:

$$\frac{C, v \qquad D, \neg v}{C, D}$$ C, D are clauses, ¬v∉C and v∉D

Given literal *p*, set of clauses *S*, let *P* be the clauses in *S* that contain *p* only positively and let *N* be the clauses that contain *p* only negatively. Let *E* be the rest of the clauses. Then *S* is SAT iff *S'* is SAT, where *S'*= *E* U the set of all *p*-resolvents of *P* and *N*.

$$\{\{\neg p, q, r, s\}, \{p, \neg q, s\}, \{\neg p, \neg q, r, \neg s\}, \boxed{\{p, \neg r, \neg s\}}, \{\neg p, \neg q, \neg r\}, \{p, q\}, \{\neg p, \neg q, s\}\}$$

Resolve on *q*

$$\{\neg p, p, r, s\}$$

$$\{\boxed{\{p, \neg r, \neg s\}}, \{\neg p, r, s\}, \{p, s\}\}$$

Notice that clauses that contain a literal and its negation can be thrown away. Why?

Resolution Example

Resolution rule:

$$\frac{C, v \qquad D, \neg v}{C, D}$$

C, D are clauses, ¬v∉C and v∉D

Given literal *p*, set of clauses *S*, let *P* be the clauses in *S* that contain *p* only positively and let *N* be the clauses that contain *p* only negatively. Let *E* be the rest of the clauses. Then *S* is SAT iff *S'* is SAT, where *S'*= *E* U the set of all *p*-resolvents of *P* and *N*.

$\{\{\neg p, q, r, s\}, \{p, \neg q, s\}, \{\neg p, \neg q, r, \neg s\}, \{p, \neg r, \neg s\}, \{\neg p, \neg q, \neg r\}, \{p, q\}, \{\neg p, \neg q, s\}\}$

Resolve on *q* $\quad \{\neg p, p, r, s\}$

$\{\{p, \neg r, \neg s\}, \{\neg p, r, s\}, \{p, s\}\}$

Notice that clauses that contain a literal and its negation can be thrown away. Why?

Resolve on *r*

$\{\{p, s\}\}$ Sat, resolve on *p* to get {} or use pure literal rule

How do we generate a satisfying assignment? Next homework

DP SAT Algorithm

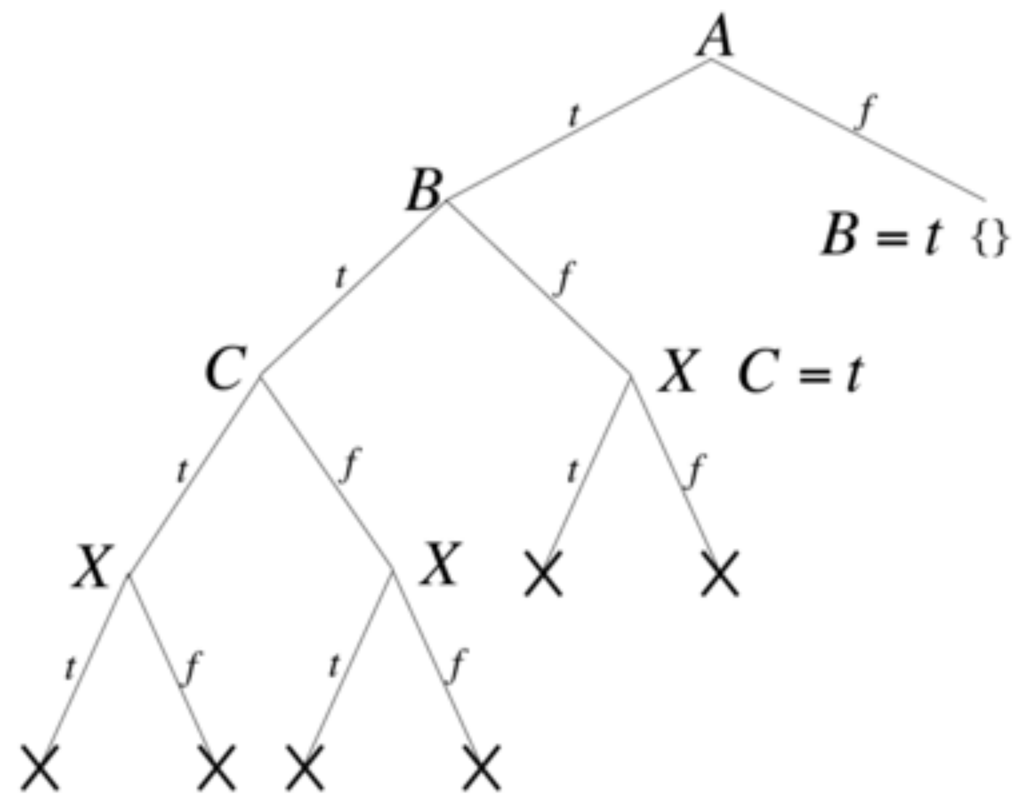
- Input: CNF formula, Output: SAT/UNSAT
- Base case: empty clause: UNSAT
- Base case: no clauses: SAT
  - Apply these two rules until fixpoint
    - Pure literal rule
    - BCP
  - Choose var, say *x*, perform all possible resolutions, remove trivial clauses and clauses containing *x*
  - Repeat
- Existentially quantify variables, one at a time
- Problem: space blow-up

DPLL SAT Algorithm

- BCP
- Base case: empty clause: UNSAT
- Remove clauses containing pure literals (modern solvers don't do this)
- Base case: no clauses: SAT
- Choose some var, say x
  (if removing pure literals, x has to appear in both phases)
  - Add {x} and recursively call DPLL
  - Add {¬x} and recursively call DPLL
  - If one of the calls returns SAT, return SAT
  - Else return UNSAT
- Correctness follows from Shannon expansion
- In contrast to DP, space is not a problem

DPLL SAT Example

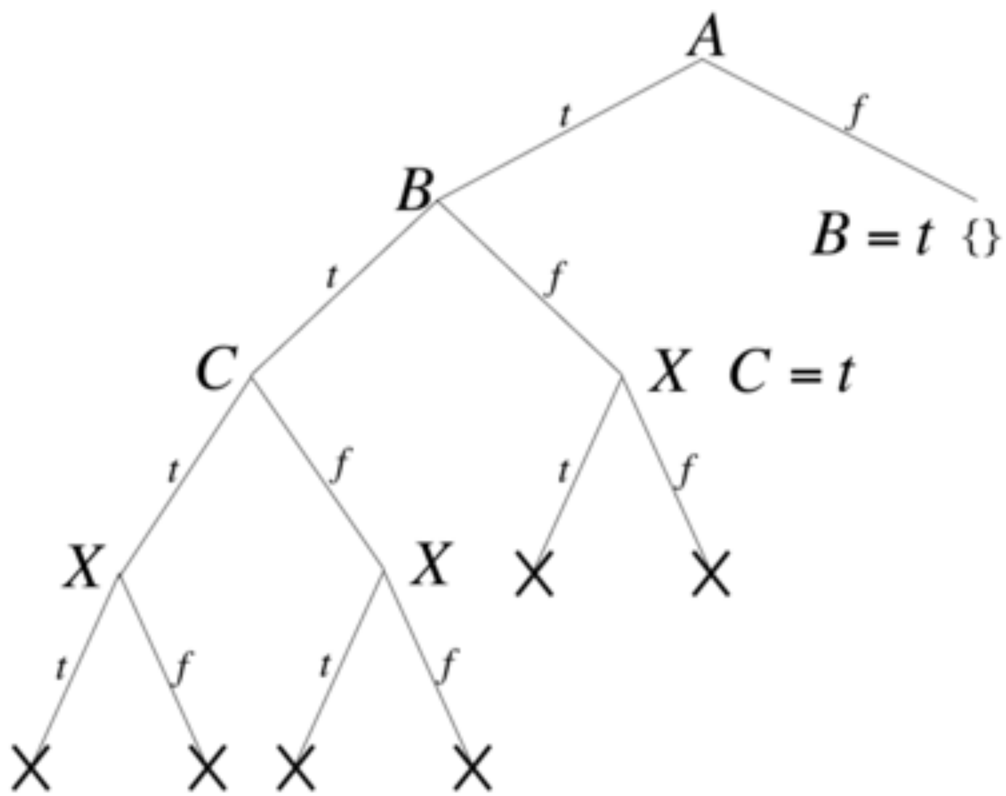
$$\Delta = \begin{array}{l} 1.\ \{A, B\} \\ 2.\ \{B, C\} \\ 3.\ \{\neg A, \neg X, Y\} \\ 4.\ \{\neg A, X, Z\} \\ 5.\ \{\neg A, \neg Y, Z\} \\ 6.\ \{\neg A, X, \neg Z\} \\ 7.\ \{\neg A, \neg Y, \neg Z\} \end{array}$$



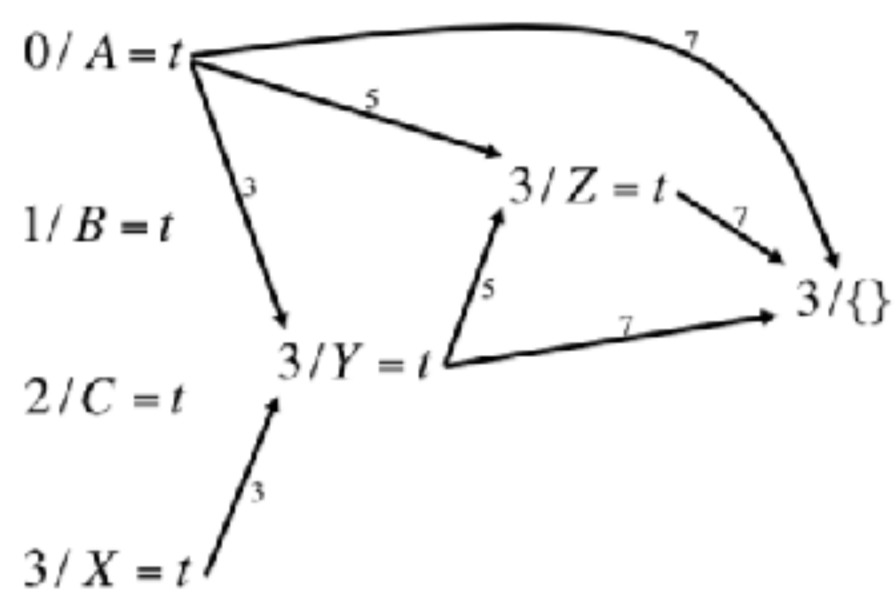
- Note that when DPLL detects contradictions it backtracks chronologically
- When we get a contradiction with X, we try ¬X, then we go back and try ¬C and X, ¬X again, …
- But the real problem was that we set A; can we avoid this exponential search?
- Yes: non-chronological backtracking, a major improvement

Examples/figures from chp. 3 SAT handbook: pure literals not removed

Implication Graphs

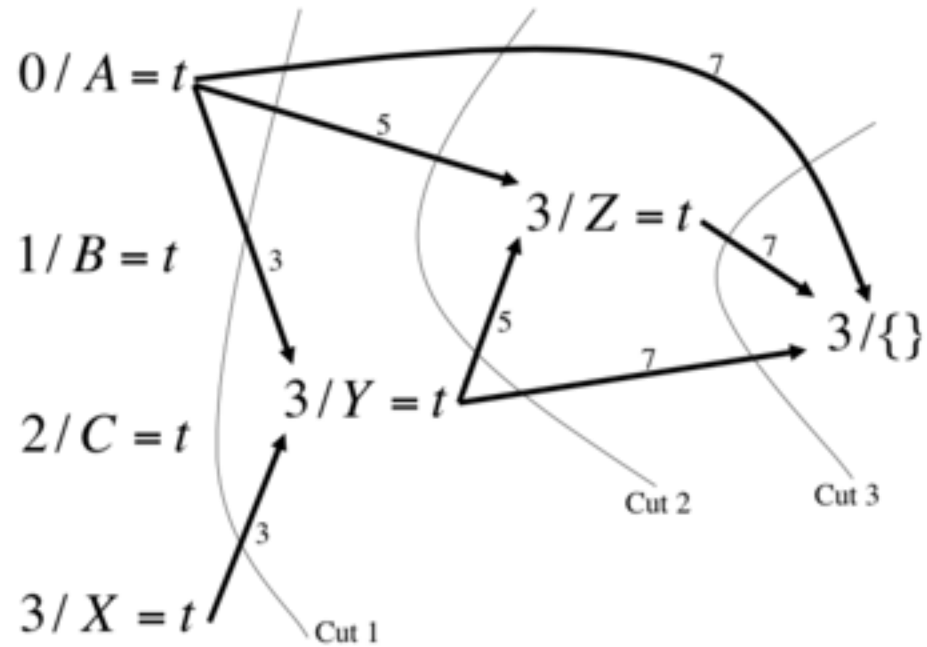


$$\Delta = \begin{array}{l} 1.\ \{A, B\} \\ 2.\ \{B, C\} \\ 3.\ \{\neg A, \neg X, Y\} \\ 4.\ \{\neg A, X, Z\} \\ 5.\ \{\neg A, \neg Y, Z\} \\ 6.\ \{\neg A, X, \neg Z\} \\ 7.\ \{\neg A, \neg Y, \neg Z\} \end{array}$$



- Nodes are l/V=v: var V set to v @ level l
- If node implied, justification recorded (clause #, edges from assignments)
- {} denotes contradiction

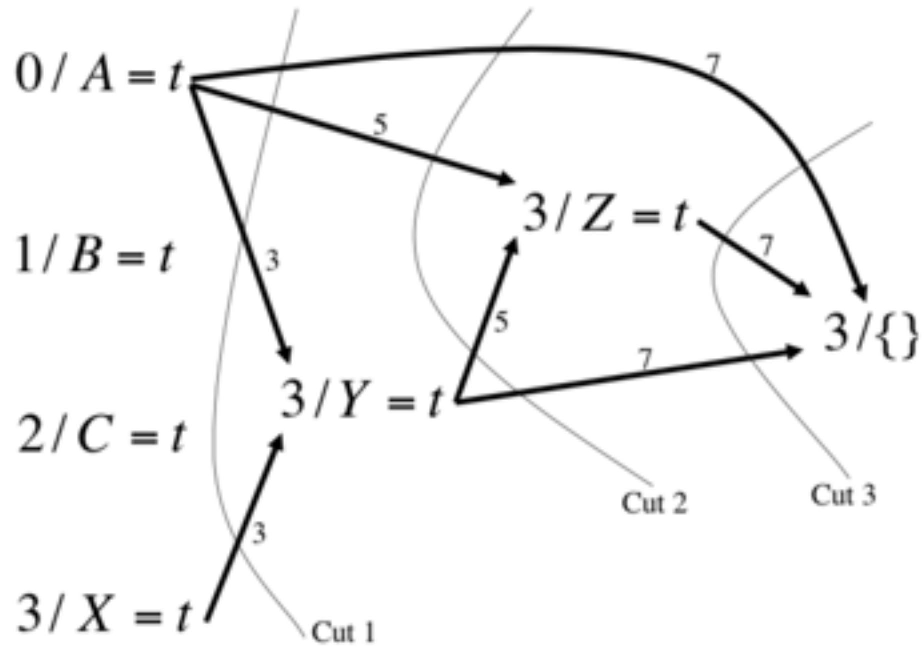
Conflict-Driven Clauses



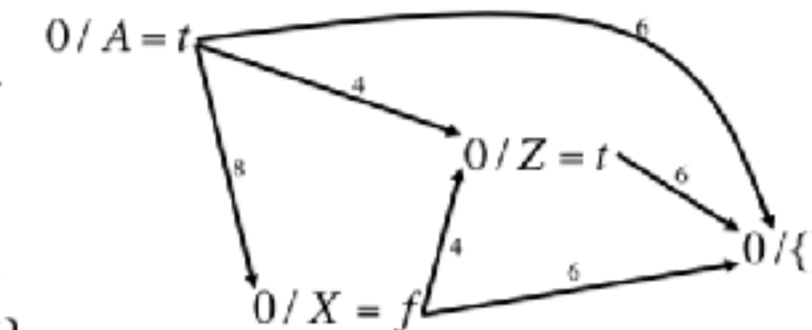
$$\Delta = \begin{array}{l} 1.\ \{A, B\} \\ 2.\ \{B, C\} \\ 3.\ \{\neg A, \neg X, Y\} \\ 4.\ \{\neg A, X, Z\} \\ 5.\ \{\neg A, \neg Y, Z\} \\ 6.\ \{\neg A, X, \neg Z\} \\ 7.\ \{\neg A, \neg Y, \neg Z\} \end{array}$$

- Consider any cut of the implication graph that separates decision vars from {}
- The nodes with an edge that crosses the cut are in conflict set
- Negate the assignments in the set to obtain a conflict-driven clause
- Conflict clauses: Cut1: {¬A,¬X}, Cut2: {¬A, ¬Y}, Cut3: {¬A, ¬Z, ¬Y}
- Conflict–driven clauses generated from cuts that contain exactly one variable assigned at the level of conflict are said to be asserting: Cut1 & Cut2 (not Cut 3)

Non-Chronological Backtracking



$$\Delta = \begin{array}{l} 1.\ \{A, B\} \\ 2.\ \{B, C\} \\ 3.\ \{\neg A, \neg X, Y\} \\ 4.\ \{\neg A, X, Z\} \\ 5.\ \{\neg A, \neg Y, Z\} \\ 6.\ \{\neg A, X, \neg Z\} \\ 7.\ \{\neg A, \neg Y, \neg Z\} \end{array}$$



- Asserting conflict clauses: Cut1: 8. {¬A,¬X}, Cut2: {¬A, ¬Y}
- Assertion level: 2nd highest level in asserting clause (0 for cuts 1, 2) or -1
- Backtrack to assertion level and add a learned clause (non-chronological!)
- We can now immediately infer (BCP) ¬X (we use Cut1), so we have A, ¬X
- Then by BCP: Z (4), ¬Z (6) so we get a new implication graph
- Asserting clauses: {¬A} at level -1, so we have ¬A, BCP: B and we're done
- Compare to previous search, where the algorithm had to go back a level at a time
- Clause learning can generate exponentially shorter proofs of unsat!

Modern CDCL Solvers

- Based on DPLL, but with conflict-driven clause learning
- Data structures to speed up BCP: 2-watched literal scheme
- Data structures for clause learning
- Decision heuristics: select recently active literals (VSIDS)
- Preprocessing: greedy variable elimination
- Inprocessing: interleave preprocessing & search
- Clause deletion: learned clauses lead to memory & efficiency problems, so delete large, inactive clauses
- Random restarts: keep learned clauses, but restart
  - avoids getting stuck in hard part of search space
  - phase saving: pick last phase of assignment

HornSAT

- A CNF formula is Horn if every clause has at most one positive literal
  - (¬a,b), (¬a,¬b,¬c,¬d), (a),(¬b,¬a,d),(¬c)
- Think of clauses as rules that “fire” under assignment A, if LHS holds
  - a⇒b, abcd⇒false, a, ba⇒d, c⇒false (or ¬c)
- HornSAT is in P
  - BCP (until fixpoint), constructing a partial assignment
  - If empty clause, return unsat
  - Else return sat (set remaining vars to false)
- Minimal assignment returned: all lits that have to be true in all sat assignments
- Note: if all clauses have 1 pos literal, then SAT (assign true to every var)
- Linear time: BCP
- Dual horn: every clause has at most one negative literal
- Same problem

Renamable Horn

- What about $\{x, y, \neg z\}, \{\neg x, y, \neg z\}$?
- Renamable horn: There is a subset of variables such that if we negate every occurrence, we have a horn formula
- Can determine if renamable horn in Ptime
- Can solve such problems in Ptime
- Lemma: F is renamable Horn iff there exists an interpretation such that at most one literal per clause is false
- So, we can test for renamability of F by using 2SAT
  - Find sat. assignment for $\wedge_{c \in F} \wedge_{u,v \in c} \ (u \vee v)$
  - Rename variables occurring positively in assignment
- Unit propagation can solve renamable horn problems (and more) in linear time (so no need to check for renamability)

SAT Solving Algorithms

- Symbolic SAT solving
  - Use BDDs, but try hard to not get blowup of intermediate BDDs
  - So, existential quantification and other techniques are used
  - The goal is to minimize the size of intermediate BDDs
- SAT by inference rules
- Stalmarck's Algorithm
  - Preprocess the formula
  - Apply simple inference rules: 0-saturation
  - Apply dilemma rule: for each variable x, 0-saturate $f|\ell$ and $f|\neg\ell$, and all common conclusions to f. Repeat until fixpoint: 1-saturation
  - n-saturation: case split over all combinations on n variables

First Order Logic

- Example: Group Theory
  - (G1) For all x, y, z: (x • y) • z = x • (y • z)
  - (G2) For all x: x • e = x
  - (G3) For all x there is a y such that: x • y = e
- Theorem: For every x, there is a y such that y • x = e
- Examples of groups: Nat, +, 0?; Int, +, 0?, Real, *, 1?
- Proof:

  By (G3) there is: a y s.t. x • y = e and a z s.t. y • z = e

  Now: y • x = y • x • e = y • x • y • z = y • e • z = y • z = e
- Is this true for all groups? Why?
- How many groups are there?
- Are there true statements about groups with no proof?

First Order Logic

- First Order Logic forms the foundation of mathematics
- We study various objects, e.g., groups
- Properties of objects captured by “non-logical” axioms
  - (G1-G3 in our example)
- Theory consists of all consequences of “non-logical” axioms
  - Derivable via logical reasoning alone
  - That’s it; no appeals to intuition
- Separation into non-logical axioms logical reasoning is astonishing: all theories use exactly same reasoning
- But, what is a proof ($\Phi \vdash \phi$)?
- Question leads to computer science
- Proof should be so clear, even a machine can check it

First Order Logic: Syntax

- Every FOL (first order language) includes
  - Variables $v_0, v_1, v_2, \ldots$
  - Boolean connectives: $\vee, \neg$
  - Equality: $=$
  - Parenthesis: (,)
  - Quantifiers: $\exists$
- The symbol set of a FOL contains (possibly empty) sets of
  - relation symbols, each with an arity > 0
  - function symbols, each with an arity > 0
  - constant symbols
- Example: groups 2-ary function symbol $\cdot$ and constant e
- Set theory: $\in$, a 2-ary relation symbol, ...

First Order Logic: Terms

- Terms denote objects of study, e.g., group elements
- The set of *S*-terms is the least set closed under:
  - Every variable is a term
  - Every constant is a term
  - If $t_1, ..., t_n$ are terms and f is an n-ary function symbol, then $f(t_1, ..., t_n)$ is a term

First Order Logic: Formulas

- Formulas: statements about the objects of study
- An atomic formula of S is
  - $t_1 = t_2$ or
  - $R(t_1, \ldots, t_n)$, where t_i is an S-term and R is an n-ary relation symbol in S
- The set of S-formulas is the least set closed under:
  - Every atomic formula is a formula
  - If ϕ, ψ are S-formulas and x is a variable, then $\neg\phi$, $(\phi \vee \psi)$, and $\exists x\phi$ are S-formulas
- All Boolean connectives can be defined in terms of $\neg$ and $\vee$
- We can define $\forall x\phi$ to be $\neg\exists x\neg\phi$

Definitions on Terms & Formulas

- Define the notion of a free variable for an S-formula
- The definition of formula depends on that of term
- So, we're going to need an auxiliary definition:

$$var(x) = \{x\}$$

$$var(c) = \{\}$$

$$var(f(t_1, \ldots, t_n)) = var(t_1) \cup \cdots \cup var(t_n)$$

- Is this a definition? (termination!)

$$free(t_1 = t_2) = var(t_1) \cup var(t_2)$$

$$free(R(t_1, \ldots, t_n)) = var(t_1) \cup \cdots \cup var(t_n)$$

$$free(\neg\phi) = free(\phi)$$

$$free((\phi \vee \psi)) = free(\phi) \cup free(\psi)$$

$$free(\exists x\phi) = free(\phi) \setminus \{x\}$$