Lecture 4: Introduction to SAT and CFG

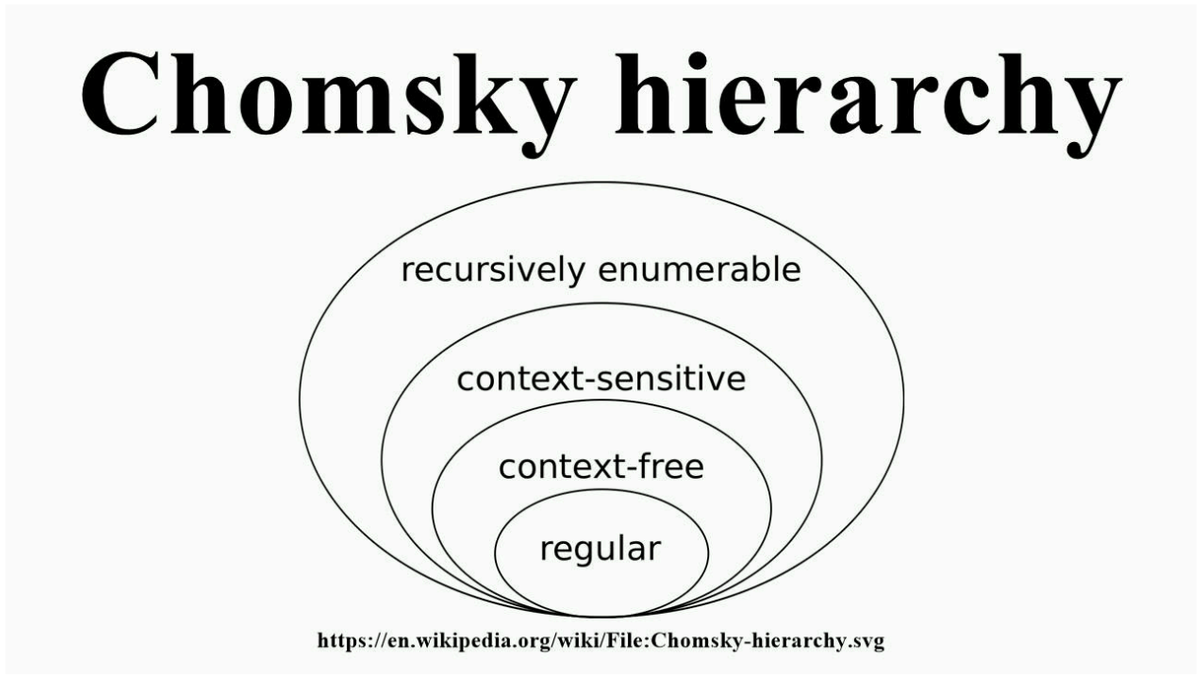
Yu Feng

Spring 2021

Summary of previous lecture

- 1st paper review was out
- HW1 is out
- The spectrum of program synthesis
- Solver-aided programming II (synthesis)

A typical flow of a compiler

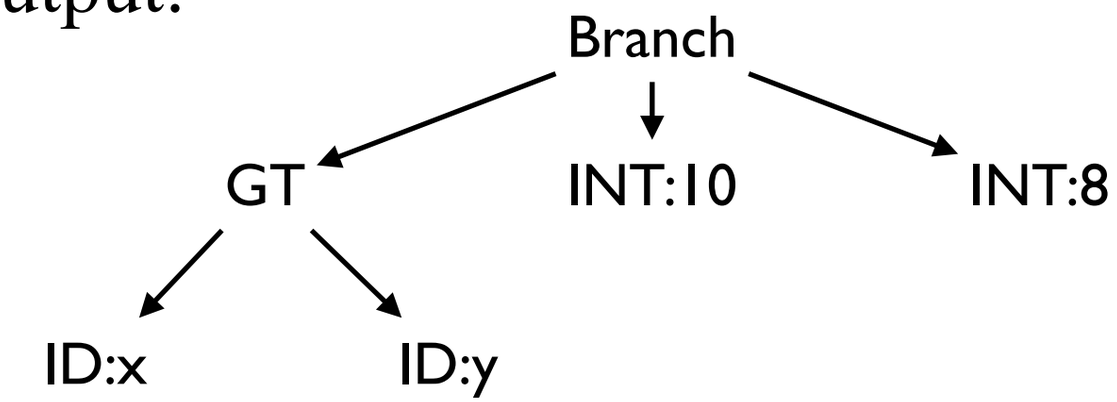


Example

- Input: Consider the following expression: **if** x>y **then** 10 **else** 8

- Parse Input: TOKEN_IF TOKEN_ID("x") TOKEN_GT TOKEN_ID("y") TOKEN_THEN TOKEN_INT(10) TOKEN_ELSE TOKEN_INT(8)

- Parser Output:



The role of a parser

- Not all strings of tokens are programs...
- Parser must distinguish between valid and invalid strings of tokens
- What we need:
  - A language for describing valid strings of tokens
  - A method for recognizing if a string of tokens is in this language or not

Context free grammar (CFGs)

- Programming language constructs have ***recursive*** structure
- Example: An expression is
  - A constant
  - *expression* + *expression*,
  - **if** *expression* **then** *expression* **else** expression, ...
- Context free grammars are a natural notation for this recursive structure

CFGs in more detail

- A CFG consists of:
  - A set of terminals T
  - A set of non-terminals N
  - A start symbol S (non-terminal)
  - A set of productions: $X \rightarrow Y_1 Y_2 \ldots Y_n$

where $X \in N$ and $Y_i \in (T \cup N \cup \{\varepsilon\})$

CFGs example

- Recall the earlier fragment:

EXPR → **if** *EXPR* **then** *EXPR* **else** *EXPR*

| *EXPR* + *EXPR*

| *ID*

- Some strings in this language:

ID

IF ID THEN ID ELSE ID

ID + ID

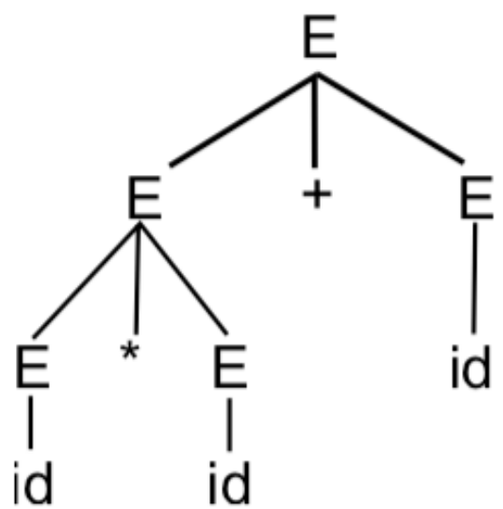
IF ID THEN ID+ID ELSE ID

IF IF ID THEN ID ELSE IF THEN ID ELSE ID

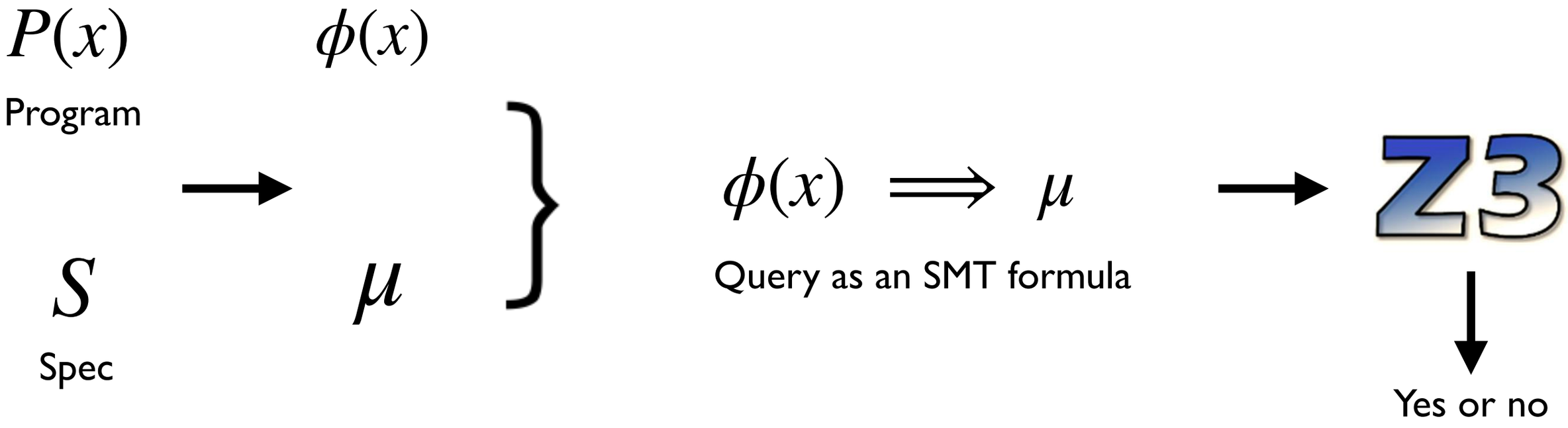
From derivations to parse trees

- A derivation is a sequence of productions: $S \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots$
- A derivation can be drawn as a tree
  - Start symbol is the tree's root
- For a production $X \rightarrow Y_1 \ldots Y_n$ add children $Y_1 \ldots Y_n$ to node X

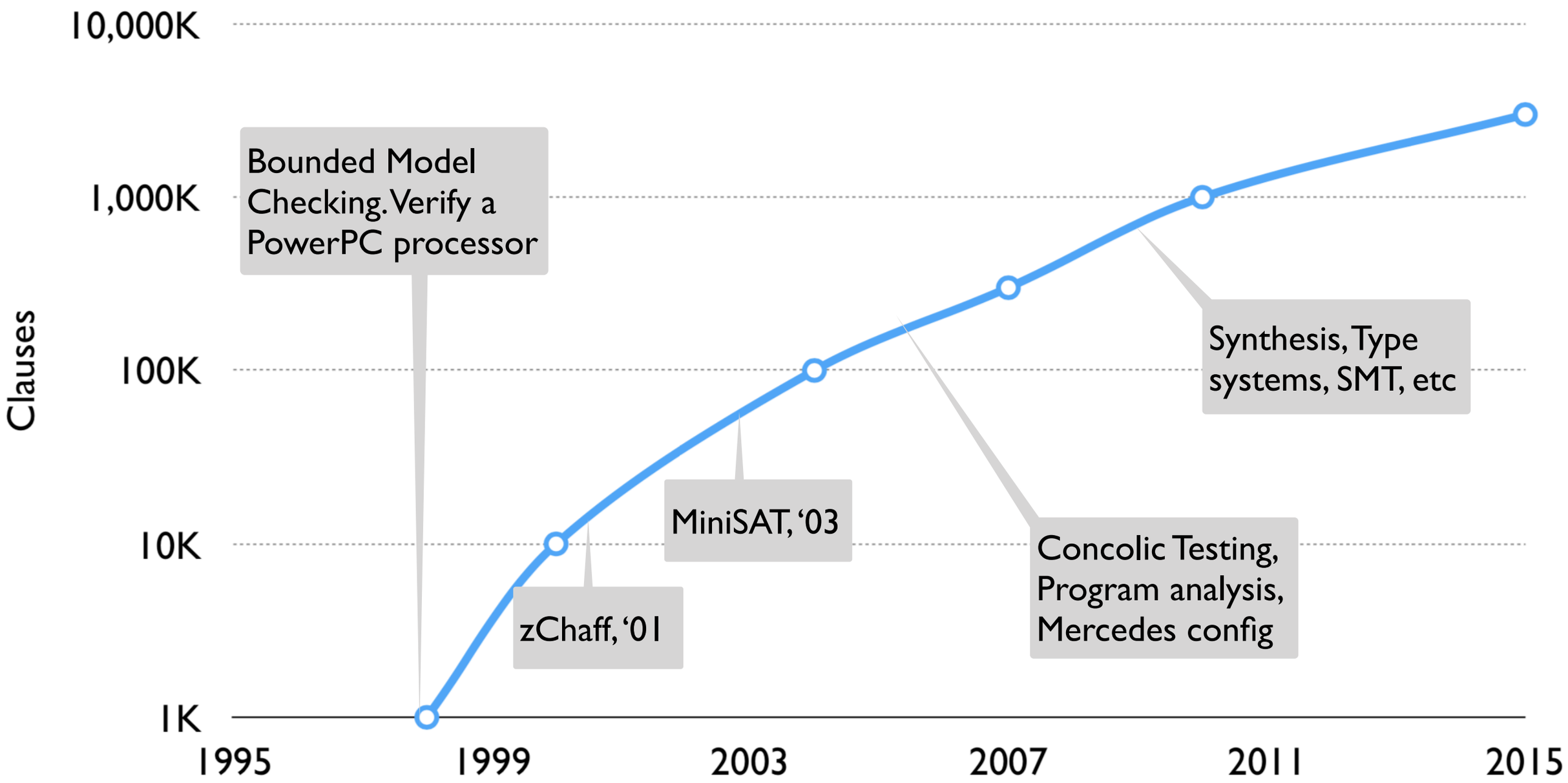
```
   E
→ E+E
→ E*E+E
→ id*E+E
→ id*id + E
→ id*id + id
```



Workhorse of formal methods

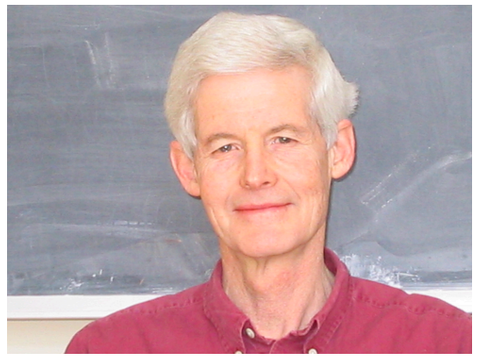


SAT solving and its applications



Why so many work on SAT

- Many interesting and computationally difficult problems can be reduced to SAT
- Boolean satisfiability is the first problem proved to be NP-complete
- Key idea: write one really good SAT solver and reduce all other NP-complete problems to SAT
-



Stephen Cook

Syntax of propositional logic

$$(\neg p \wedge \top) \vee (q \rightarrow \bot)$$

Atom

Truth symbols: $\top$ ("true"), $\bot$ ("false")
propositional variables: *p,q,r,...*

Literal

an atom α or its negation ¬α

Formula

an atom or the application of a **logical connective**

to formulas F_1, F_2:

$\neg F_1$	"not"	**(negation)**
$F_1 \wedge F_2$	"and"	**(conjunction)**
$F_1 \vee F_2$	"or"	**(disjunction)**
$F_1 \rightarrow F_2$	"implies"	**(implication)**
$F_1 \leftrightarrow F_2$	"if and only if"	**(iff)**

Semantics of propositional logic

An **interpretation** *I* for a propositional formula *F* maps every variable in *F* to a truth value:

$$I : \{ p \mapsto \text{true}, q \mapsto \text{false}, \ldots \}$$

I is a **satisfying interpretation** of *F*, written as $I \models F$, if *F* evaluates to true under *I*.

I is a **falsifying interpretation** of *F*, written as $I \not\models F$, if *F* evaluates to false under *I*.

A satisfying interpretation is also a **model**

Semantics of propositional logic

Base cases:

- $I \models \top$
- $I \not\models \bot$
- $I \models p$ iff $I[p]$ = true
- $I \not\models p$ iff $I[p]$ = false

Inductive cases:

- $I \models \neg F$ *iff* $I \not\models F$
- $I \models F_1 \wedge F_2$ *iff* $I \models F_1$ *and* $I \models F_2$
- $I \models F_1 \vee F_2$ *iff* $I \models F_1$ *or* $I \models F_2$
- $I \models F_1 \rightarrow F_2$ *iff* $I \not\models F_1$ *or* $I \models F_2$
- $I \models F_1 \leftrightarrow F_2$ *iff* $I \models F_1$ *and* $I \models F_2$, *or* $I \not\models F_1$ *and* $I \not\models F_2$

Semantics of propositional logic

F: $(p \wedge q) \rightarrow (p \vee \neg q)$

I: $\{p \mapsto \text{true}, q \mapsto \text{false}\}$

$I \models F$

Satisfiability v.s. validity

F is **satisfiable** iff $I \models F$ for some *I*.

F is **valid** iff $I \models F$ for all *I*.

Duality of satisfiability and validity:

F is valid iff $\neg F$ is unsatisfiable.

One algorithm for checking both satisfiability and validity.

SAT solving with normal forms

A **normal form** for a logic is a syntactic restriction such that every formula in the logic has an equivalent formula in the normal form.

Three important normal forms:

- Negation Normal Form (NNF)
- Disjunctive Normal Form (DNF)
- Conjunctive Normal Form (CNF)

Negation normal form

Atom := Variable | ⊤ | ⊥
Literal := Atom | ¬Atom
Formula := Literal | Formula op Formula
op := ∧ | ∨

- The only allowed connectives are ∧, ∨, and ¬.
- ¬ can appear only in literals

Conversion to NNF performed using DeMorgan's Laws:

$\neg(F \wedge G) \iff \neg F \vee \neg G$ $\neg(F \vee G) \iff \neg F \wedge \neg G$

Disjunctive normal form

Atom := Variable | $\top$ | $\bot$
Literal := Atom | ¬Atom
Formula := Clause $\vee$ Formula
Clause := Literal | Literal $\wedge$ Clause

- Disjunction of conjunction of literals
- Trivial to decide if a DNF formula is SAT, why?
- Why not modern SAT solvers use DNF?

To obtain DNF, convert to NNF and distribute $\wedge$ over $\vee$:

$$(F \wedge (G \vee H)) \iff (F \wedge G) \vee (F \wedge H)$$

$$((G \vee H) \wedge F) \iff (G \wedge F) \vee (H \wedge F)$$

Conjunctive normal form

Atom := Variable | $\top$ | $\bot$
Literal := Atom | ¬Atom
Formula := Clause $\wedge$ Formula
Clause := Literal | Literal $\vee$ Clause

- Conjunction of disjunction of literals
- Hard to decide if a CNF formula is SAT
- Default language in modern SAT solvers

To obtain CNF, convert to NNF and distribute $\vee$ over $\wedge$:

$$(F \vee (G \wedge H)) \Longleftrightarrow (F \vee G) \wedge (F \vee H)$$

$$((G \wedge H) \vee F) \Longleftrightarrow (G \vee F) \wedge (H \vee F)$$

Key feature of CNF: unit resolution

Resolution rule

$$\frac{a_1 \vee \ldots \vee a_n \vee \beta \qquad b_1 \vee \ldots \vee b_m \vee \neg\beta}{a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m}$$

Proving that a CNF formula is valid can be done using just this one proof rule!
Apply the rule until a contradiction (empty clause) is derived, or no more applications are possible.

Unit resolution rule

$$\frac{\beta \qquad b_1 \vee \ldots \vee b_m \vee \neg\beta}{b_1 \vee \ldots \vee b_m}$$

Unit resolution specializes the resolution rule to the case where one of the clauses is unit (a single literal).
SAT solvers use unit resolution in combination with backtracking search to implement a sound and complete procedure for deciding CNF formulas.

A basic SAT solver (DPLL)

```
// Returns true if the CNF formula F is
// satisfiable; otherwise returns false.
DPLL(F)
  G ← BCP(F)
  if G = ⊤ then return true
    if G = ⊥ then return false
p ← choose(vars(G))
return DPLL(G{p ↦ ⊤}) ||
            DPLL(G{p ↦ ⊥})
```

Boolean constraint propagation applies unit resolution until fixed point.

If BCP cannot reduce F to a constant, we choose an unassigned variable and recurse assuming that the variable is either true or false.

If the formula is satisfiable under either assumption, then we know that it has a satisfying assignment (expressed in the assumptions). Otherwise, the formula is unsatisfiable.

Davis-Putnam-Logemann-Loveland (1962)

TODOs by next lecture

- Start working HW1 and R1
- Form your team for the final project!
- Discuss your final project during office hour