Recall From Last Time

Definition: A **context-free grammar** (CFG) is a 4-tuple $G = (V, \Sigma, R, S)$,

- V = variables = nonterminals,
- Σ = terminals,
- R = rules = productions, $R \subseteq V \times (V \cup \Sigma)^\star$,
- $S \in V$,
- V, Σ, R are all finite.

Pumping Lemma for Regular Sets: Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Let $n = |Q|$. Let $w \in \mathcal{L}(D)$ s.t. $|w| \geq n$. Then $\exists x, y, z \in \Sigma^\star$ s.t. the following all hold:

- $xyz = w$
- $|xy| \leq n$
- $|y| > 0$, and
- $(\forall k \geq 0) xy^k z \in \mathcal{L}(D)$

CFL Pumping Lemma: Let A be a CFL. Then there is a constant n, depending only on A, such that if $z \in A$ and $|z| \geq n$, then there exist strings u, v, w, x, y such that:

- $z = uvwxy$, and
- $|vx| \geq 1$, and
- $|vwx| \leq n$, and
- for all $k \in$ **N**, $uv^k wx^k y \in A$

Prop: $P = \{a^n b^m a^n b^m \mid n, m \in \mathbf{N}\}$ is not a CFL.

Prop: $NONCFL = \{a^n b^n c^n : n \in \mathbf{N}\}$ is not a CFL.

Any CFL satisfies the conclusion of the CFL Pumping Lemma, but it is *not* true that any non-CFL must fail to satisfy it. There are other tools that can show a language to be a non-CFL. These include stronger forms of the Pumping Lemma and more *closure properties*.

If A is a CFL and R a regular language, then $A \cup R$ must be regular. Proving this, however, requires a different characterization of the CFL's.

Pushdown Automata (PDA's)

Definition: A **pushdown automaton** (PDA) is a 7-tuple, $P = (Q, \Sigma, \Gamma, \Delta, q_0, Z_0, F)$

- $Q =$ finite set of states,
- $\Sigma =$ input alphabet,
- $\Gamma =$ stack alphabet,
- $\Delta \subseteq (Q \times \Sigma^\star \times \Gamma^\star) \times (Q \times \Gamma^\star)$ finite set of transitions,
- $q_0 \in Q$ start state,
- $Z_0 \in \Gamma$ initial stack symbol,
- $F \subseteq Q$ final states.

$$\text{PDA} \quad = \quad \text{NFA} \ + \ \text{stack}$$

$$\mathcal{L}(P) \quad = \quad \{w \in \Sigma^\star \mid (q_0, Z_0) \xrightarrow[P]{w} (q, X),\ q \in F, X \in \Gamma^\star\}$$

Theorem 5.1 *Let $A \subseteq \Sigma^\star$ be any language. Then the following are equivalent:*

1. *$A = \mathcal{L}(G)$, for some CFG G.*
2. *$A = \mathcal{L}(P)$, for some PDA P.*
3. *A is a context-free language.*

Proof: We give only a sketch here – there are detailed proofs in [HMU], [LP], and [S].

To prove (1) implies (2), we can build a "bottom-up parser" or "top-down" parser, similar to those used in real-world compilers except that the latter are *deterministic*.

The top-down parser is a PDA that:

- begins by pushing "$S\$$" onto its stack
- may pop a terminal from the stack if can at the same time read a matching input letter,
- may execute a rule $A \to w$ by popping A and pushing w^R,
- ends by popping the $\$$ when done with the input

The proof that the language of any PDA is a CFL (that (2) implies (1)) is of less practical interest.

Given states i and j, let A_{ij} be the set of strings that *could* take the PDA from state i with empty stack to state j with empty stack.

We have all rules of the form $A_{pq} \to A_{pr}A_{rq}$, and a rule $A_{pq} \to aA_{rs}b$ whenever moves of the PDA warrant it.

♠

Boolean Logic: Syntax

Boolean variables: $X = \{x_1, x_2, x_3, \ldots\}$

A boolean variable represents an atomic statement that may be either true or false. There may be infinitely many of these available.

Boolean expressions:

- atomic: x_i, $\top$ ("top"), $\perp$ ("bottom")
- $(\alpha \vee \beta)$, $\neg\alpha$, $(\alpha \wedge \beta)$, $(\alpha \rightarrow \beta)$, $(\alpha \leftrightarrow \beta)$, for α, β Boolean expressions

Note that any particular expression is a finite string, and thus may use only finitely many variables.

A *literal* is an atomic expression or its negation: x_i, $\neg x_i$, $\top$, $\perp$.

As you may know, the choice of operators is somewhat arbitary as long as we have a *complete set*, one that suffices to simulate all boolean functions. On HW#1 we argued that $\{\wedge, \vee, \neg\}$ is already a complete set.

The expressions form a context-free language. As we mentioned before, we may "cheat" by omitting parentheses, etc., as long as the parsing is clear. Even with the cheating it's not too hard to tell whether a string is a valid expression.

Examples of boolean expressions:

- x_1
- $b_2 \vee \neg b_2$
- $x_1 \leftrightarrow x_2$
- $((a \rightarrow b) \wedge (b \rightarrow c)) \rightarrow (a \rightarrow c)$

Boolean Logic: Semantics

A boolean expression has a meaning, a **truth value** of true or false, once we know the truth values of all the individual variables.

A **truth assignment** is a function $T : X' \subseteq X \to \{\mathbf{true}, \mathbf{false}\}$, where X is the set of all variables. An assignment is *appropriate* to an expression φ if it assigns a value to all variables used in φ.

The double-turnstile symbol $\models$ (read as "models") denotes the relationship between a truth assignment and an expression. The statement "$T \models \varphi$" (read as "T models φ") simply says "φ is true under T".

If T is appropriate to φ, we *define* when $T \models \varphi$ is true by induction on the structure of φ:

- $\top$ is true and $\bot$ is false for any T,
- A variable x_i is true iff T says that it is,
- If $\varphi = (\alpha \wedge \beta)$, $T \models \varphi$ iff both $T \models \alpha$ and $T \models \beta$,
- If $\varphi = (\alpha \vee \beta)$, $T \models \varphi$ iff either $T \models \alpha$ or $T \models \beta$ or both,
- If $\varphi = (\alpha \rightarrow \beta)$, $T \models \varphi$ unless $T \models \alpha$ and $T \not\models \beta$,
- If $\varphi = (\alpha \leftrightarrow \beta)$, $T \models \varphi$ iff $T \models \alpha$ and $T \models \beta$ are both true or both false.

Definition 5.2 α and β are *semantically equivalent*

$$(\alpha \equiv \beta)$$

iff for all T appropriate to α and β,

$$T \models (\alpha \leftrightarrow \beta)$$

♠

Examples:

- $x_1 \;\equiv\; x_1 \vee \bot$
- $a \to a \;\equiv\; \top$
- $a \to b \;\equiv\; \neg b \to \neg a$
- $a \to b \;\equiv\; \neg a \vee b$
- $\neg(a \wedge b) \;\equiv\; \neg a \vee \neg b$
- $\neg(a \vee b) \;\equiv\; \neg a \wedge \neg b$
- $a \vee b \;\equiv\; b \vee a$
- $(a \vee b) \vee c \;\equiv\; a \vee (b \vee c)$
- $a \vee (b \wedge c) \;\equiv\; (a \vee b) \wedge (a \vee c)$
- $a \;\equiv\; \neg\neg a$

Proposition 5.3 *Every boolean expression, φ, is equivalent to one in Conjunctive Normal Form (CNF), and to one in Disjunctive Normal Form (DNF).*

Proof: DNF: look at the truth table for φ:

x	y	z	$x \leftrightarrow y$	$(x \leftrightarrow y) \leftrightarrow z$
0	0	0	1	0
0	0	1	1	1
0	1	0	0	1
0	1	1	0	0
1	0	0	0	1
1	0	1	0	0
1	1	0	1	0
1	1	1	1	1

$$(\bar{x} \wedge \bar{y} \wedge z) \ \vee \ (\bar{x} \wedge y \wedge \bar{z}) \ \vee \ (x \wedge \bar{y} \wedge \bar{z}) \ \vee \ (x \wedge y \wedge z)$$

CNF: put $\neg\varphi$ in DNF.

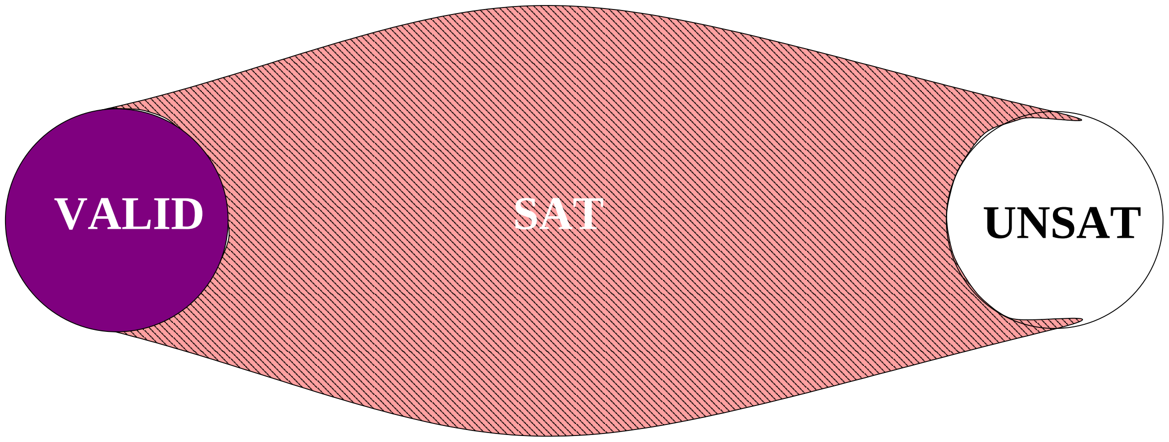
Use De Morgan's law:

$$\neg(C_1 \vee \cdots \vee C_k) \ \equiv \ (\neg C_1 \ \wedge \ \cdots \ \wedge \ \neg C_k)$$

♠

Definition 5.4 A boolean expression φ is *satisfiable* iff there exists $T \models \varphi$.

φ is *valid* iff for all T appropriate to φ, $T \models \varphi$. ♠



Proposition 5.5 *For any boolean expression* φ,

$$\varphi \in \mathit{UNSAT} \quad \Leftrightarrow \quad \neg\varphi \in \mathit{VALID}$$

$$\text{UNSAT} \leq \text{VALID}; \qquad \text{VALID} \leq \text{UNSAT}$$

Proposition 5.6 • φ *is* unsatisfiable *iff* $\varphi \equiv \bot$.

- φ *is* satisfiable *iff* $\varphi \not\equiv \bot$.
- φ *is* valid *iff* $\varphi \equiv \top$.

Relations With Other Models

A boolean expression specifies a boolean function of its variables. It's natural to ask how the computation of this function fits into our other models of computation.

A boolean expression of length ℓ can be converted into an SLP of length $\theta(\ell)$, by induction on the definition of the expression. If we already have SLP's computing α and β, we can concatenate them and then with $O(1)$ more instructions compute any of the expressions $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \rightarrow \beta)$, or $(\alpha \leftrightarrow \beta)$.

Given any SLP and any of its instructions, the output of the instruction is a boolean expression, and thus there is *some* boolean expression that computes it. Can we be sure that a short SLP leads to a short expression? We can if the SLP is **read-once**, meaning that each non-input variable occurs only once on the *right-hand side* of an instruction. You'll prove on HW#2 that the value of ℓ'th instruction of a read-once SLP has an expression of length $O(\ell)$.

It is *unknown* whether every polynomial-size SLP can be made read-once while keeping the size polynomial. As we may prove later in the course, this is equivalent to the question of whether every polynomial-size SLP can be given depth $O(\log n)$ while keeping its size polynomial. It is generally conjectured that it cannot.

What about the first-order model and boolean expressions?

We can convert a $\forall$ or $\exists$ quantifier into a boolean expression using $\wedge$ or $\vee$. For example, $\forall x : \varphi(x)$ can be equivalently written

$$\varphi(0) \wedge \varphi(1) \wedge \ldots \wedge \varphi(n-1)$$

where each of the $\varphi(i)$'s is one of the input bits.

By induction, we can prove that a first-order formula of quantifier depth d corresponds to a boolean expression (in the atomic variables) of length $O(n^d)$. (Try this!)

In this case it is *known* that there are polynomial-length boolean expressions that cannot be converted into first-order formulas, so that the boolean expresssions are strictly more powerful. We won't get to the proof of this (due to Mike Sipser and three others, in 1981) in this course.

We have defined several properties of boolean expressions: validity, satisfiability, and their negations. It is natural to ask now difficult it is for an algorithm to take an expression as input and determine whether it is valid or satisfiable.

The method of truth tables gives us a way to answer these problems, though not a very satisfactory one. Given a formula in n variables, there are 2^n different settings of the variables – 2^n rows of the truth table. We know how to evaluate the formula in each of these settings, and thus *given enough time* we know how to create the entire truth table and *see* whether there is at least one true row or at least one false row.

Of course except for very small n this running time is prohibitively large. Is there a better way to answer the questions? We don't know, and we will see that this possibility is tied to the major open questions of complexity theory.

One thing we *do* know is that *in some circumstances* we can determine the validity or satisfiability of an expression without computing its entire truth table, by giving a *formal proof* in some system. This is our next topic.

The Fitch Proof System

If two expressions are equivalent, or if one is a tautological consequence of the other, this fact can be established by truth tables.

But the 2^k lines of the truth table are far too many to work with unless k is small. We need a more efficient method to establish tautologies.

[BE] provides a proof system called **Fitch** or $\mathcal{F}$. At this point in the course we are concerned with the propositional subset $\mathcal{F}_T$ of Fitch, leaving out rules dealing with quantifiers.

A Fitch proof is a sequence of expressions, each one of which is justified in terms of previous ones. There are twelve **proof rules** that tell us when a statement is justified.

Fitch has no **axioms** (statements assumed to be true without proof) but we typically start with some **premises** and reach a **conclusion** that follows from those premises.

If from a set P of premises we can derive φ, we write $P \vdash \varphi$, read as "P proves φ". This **single turnstile** symbol $\vdash$ is not to be confused with the double turnstile symbol $\models$.

$\mathcal{F}$'s Propositional Proof Rules

These are conveniently listed on pages 557-9 of [BE].

$\wedge$ **Intro:** From $P_1, \ldots, P_n$, derive $P_1 \wedge \ldots \wedge P_n$.

$\wedge$ **Elim:** From $P_1 \wedge \ldots \wedge P_n$, derive P_i.

$\vee$ **Intro:** From P_i, derive $P_1 \vee \ldots \vee P_n$.

$\vee$ **Elim:** From $P_1 \vee \ldots \vee P_n$, if you have derived S separately from each P_i, derive S.

$\neg$ **Intro:** If you have derived $\perp$ from P, derive $\neg P$.

$\neg$ **Elim:** From $\neg\neg P$, derive P.

$\perp$ **Intro:** From P and $\neg P$, derive $\perp$.

$\perp$ **Elim:** From $\perp$, derive P (any P!)

$\rightarrow$ **Intro:** If you have derived Q from P, derive $P \rightarrow Q$.

$\rightarrow$ **Elim:** From $P \rightarrow Q$ and P, derive Q (also called **modus ponens**).

$\leftrightarrow$ **Intro:** If you have derived Q from P and have derived P from Q, derive $P \leftrightarrow Q$.

$\leftrightarrow$ **Elim:** From $P \leftrightarrow Q$ and P, derive Q.

As you practice with the Fitch software, there are two things you learn how to do. You must learn *when a move is legal*, and then *when a move is a good idea*. In chess, we call a player a beginner once they know all the moves, but after a lifetime of play they will still have more to learn about how to play well.

The first natural computational problem that Fitch creates is **determining whether a proof is valid**. The software does this for your proofs, and you learn to do it mentally as you look for proofs.

A proof is valid if each step is valid, and a step is valid if it follows from previous steps by a rule. Eight of the rules require certain other statements to be **available**, meaning that they are active premises or have been proved within the current scope. But four of the rules – $\vee$ **Elim**, $\neg$ **Intro**, $\rightarrow$ **Intro**, and $\leftrightarrow$ **Intro**, require not previous statements but **previous proofs**.

This means that Fitch proofs have a **nested structure**, with derivations nested within other derivations. We can define the **available statements** at any given point in the proof, as those that are in the current derivation or any of the derivations containing the current one.

As computer scientists we naturally think of a **stack** to maintain the nesting structure. At any given time we are engaged in a derivation that has certain active premises. Since premises are inherited by subderivations, we can keep them in a stack.

Testing validity of a step thus involves a single pass over the current premise stack and the current derivation, to see whether all the required statements or derivations for the step are there. This doesn't look to be a very difficult task.

We haven't yet formalized the notion of **polynomial time** yet in this course, but once we do we can see that checking a Fitch proof for validity takes polynomial time in the length of the proof.

What about the second problem, of finding a reasonably short valid proof when one exists? This is more difficult, of course, falling into the category of artificial intelligence. There are useful heuristics, but no reliable general method.

We'll show in the next lecture that every valid statement has a proof in Fitch. But even this **completeness theorem** won't tell us whether a short (i.e., polynomial-length) proof exists for every tautology. This latter question is open, as we'll see.