**Propositional letters (must in capital form)**: A,B,C,...Z
**connectives:** ¬, ∧, ∨, ⇒, ⇔, ⊕, **f**, **t**, (,)  //here t, f stands for true and false

A proposition is false (**f**) or true (**t**)
The "meaning" associated to a propositional formula is a truth value (0 or 1)
A **model** is a choice of truth value for each propositional letter.

**Well-formed formulas** (wffs) are generated by the grammar
wff → A | B | C | … | Z | **f** | **t**
| (¬wff) //¬: not
| (wff ∧ wff) //∧: conjunction of
| (wff ∨ wff) //∨: disjunction of
| (wff ⇒ wff) // ⇒: if P then Q, False ==> _ = True, True ==> x = x, same truth table as ¬A∨B
| (wff ⇔ wff) // X ⇔ Y: same as X == Y (a comparison), read as if and only if
| (wff ⊕ wff) // ⊕: exclusive or aka xor, x <+> y = x /= y

**Notational Conveniences**
We shall drop outermost parentheses. Binding sequence from tighter to looser: ¬, ∧, ∨, ⇒, ⇔, ⊕

e.g. (((¬Q) ∧ P) ⇒ (P ∨ (P ⇔ Q))) same as ¬Q ∧ P ⇒ P ∨ (P ⇔ Q)

We can use → instead of ⇒, ↔ instead of ⇔, and uses 0 for f and 1 for t.

**Truth table:**

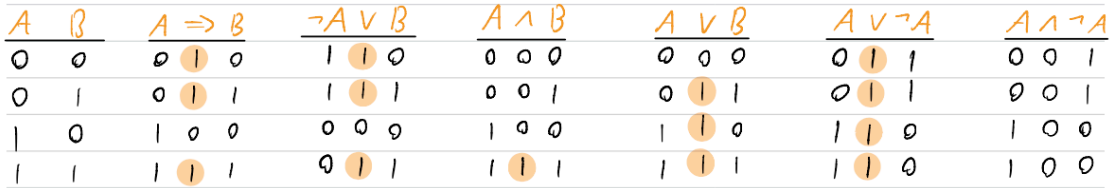
| *A* | *B* | ¬*A* | *A* ∧ *B* | *A* ∨ *B* | *A* ⇒ *B* | *A* ⇔ *B* | *A* ⊕ *B* |
|---|---|---|---|---|---|---|---|
| **f** | **f** | **t** | **f** | **f** | **t** | **t** | **f** |
| **f** | **t** | **t** | **f** | **t** | **t** | **f** | **t** |
| **t** | **f** | **f** | **f** | **t** | **f** | **f** | **t** |
| **t** | **t** | **f** | **t** | **t** | **t** | **t** | **f** |

Truth assignment: let the propositional letters be **t** / **f**
A propositional formula is valid if no truth assignment makes it false. Otherwise it is non-valid.
It is unsatisfiable if no truth assignment makes it true. Otherwise it is satisfiable.
A valid propositional formula is a tautology. An unsatisfiable propositional formula is a contradiction.

**Substitution** Preserves **Validity + unsatisfiability**
(¬P ∧ Q) ⇒ (P ⇒ R) is valid, and hence (¬(A ∨ B) ∧ (B ⇔ C)) ⇒ ((A ∨ B) ⇒ (B ⇔ C)) is valid.

(Different letters can be replaced by the same formula, but of course, all occurrences of a letter must be replaced by the same formula.)
A formula is unsatisfiable iff its negation (¬) is valid. It follows that substitution also preserves unsatisfiability.
Substitution doesn't preserve satisfiability. e.g. Take P (which is clearly satisfiable) then substitute P by Q ∧ ¬Q. All truth assignment cannot make it true.

**Models, Logical Consequence, and Equivalence**
- Let θ be a truth assignment and F be a propositional formula. If θ makes F true then θ is a **model** of F.




- G is a **logical consequence** of F iff every model of F is a **model** of G as well. In that case we write **F|=G.** In another word, every truth-assignment which make F true, also makes G true.
- If F |= G and G |= F both hold, that is, F and G have exactly the same models, then F and G are **(logically) equivalent**. In that case we write **F ≡ G.** In another word, every truth-assignment assigns the same truth-value to F and G. F ≡ G if and only if F ⇔ G
- If F ≡ G and F′ and G′ are the results of replacing each occurrence of letter P (in both) with formula H, then F′ ≡ G′

**Interchange of Equivalents**
Replacing equals by equals yields equals. If
- F is a sub-formula of H,
- F ≡ G, and
- H′ is the result of replacing F in H by G,

then H ≡ H′.
Interchange of equivalents preserves not only logical equivalence.
It also preserves logical consequence, validity, and unsatisfiability, satisfiability.

**Equivalences**

| | |
|---|---|
| Absorption: | $P \wedge P \equiv P$ |
| | $P \vee P \equiv P$ |
| Commutativity: | $P \wedge Q \equiv Q \wedge P$ |
| | $P \vee Q \equiv Q \vee P$ |
| Associativity: | $P \wedge (Q \wedge R) \equiv (P \wedge Q) \wedge R$ |
| | $P \vee (Q \vee R) \equiv (P \vee Q) \vee R$ |
| Distributivity: | $P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$ |
| | $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$ |

$\Leftrightarrow$ and $\oplus$ are also commutative and associative.

| | |
|---|---|
| Double negation: | $P \equiv \neg\neg P$ |
| De Morgan: | $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$ |
| | $\neg(P \vee Q) \equiv \neg P \wedge \neg Q$ |
| Implication: | $P \Rightarrow Q \equiv \neg P \vee Q$ |
| Contraposition: | $\neg P \Rightarrow \neg Q \equiv Q \Rightarrow P$ |
| | $P \Rightarrow \neg Q \equiv Q \Rightarrow \neg P$ |
| | $\neg P \Rightarrow Q \equiv \neg Q \Rightarrow P$ |
| Biimplication | $P \Leftrightarrow Q \equiv (P \wedge Q) \vee (\neg P \wedge \neg Q)$ |
| XOR | $A \oplus B \equiv (A \vee B) \wedge (\neg A \vee \neg B)$ |

Let $\bot$ be any unsatisfiable formula and let $\top$ be any valid formula.

| | |
|---|---|
| Duality: | $\neg\top \equiv \bot$ |
| | $\neg\bot \equiv \top$ |
| Negation from absurdity: | $P \Rightarrow \bot \equiv \neg P$ |
| Identity: | $P \vee \bot \equiv P$ |
| | $P \wedge \top \equiv P$ |
| Dominance: | $P \wedge \bot \equiv \bot$ |
| | $P \vee \top \equiv \top$ |
| Contradiction: | $P \wedge \neg P \equiv \bot$ |
| Excluded middle: | $P \vee \neg P \equiv \top$ |

**Normal Forms for Propositional Logic**
- A literal is P or ¬P where P is a propositional letter. I.e. P, ¬P, ¬Q…
- A clause is a set (disjunction) of literals, which represents their disjunction
  e.g. P —> {P}, A ∨ ¬B ∨ C —> {A, ¬B, C}

**Conjunctive normal form (CNF)** is a conjunction of disjunctions of literals (a conjunction of "clauses").
- e.g. (A ∨ ¬B) ∧ (B ∨ C ∨ D) ∧ A —> {{A, ¬B}, {B, C, D}, {A}} this is the clausal form, make no distinction between these
- Reduced CNF (RCNF) for each of its clauses, no propositional letter occurs twice.
  for example: (A ∨ ¬B ∨ ¬A) ∧ (¬C ∨ ¬B)∧(C ∨ ¬A ∨ C ∨ B) becomes (¬C ∨ ¬B) ∧ (C ∨ ¬A ∨ B)

**Disjunctive normal form (DNF)** is a disjunction of conjunctions of literals. e.g. (¬A ∧¬B)∨(¬B ∧ C)∨(A ∧¬D)
- Normal Form Does Not Mean Unique Form, e.g. (P ∧ Q ∧ R)∨(¬P ∧¬Q ∧¬R)∨(¬P ∧ R)∨(Q ∧¬R) is in reduced DNF, and so is the equivalent ¬P∨Q. So two DNF formulas may have different sizes and variables, and yet be equivalent. Similarly for (R)CNF.

**Converting a Formula to a normal form**

1 Eliminate all occurrences of $\oplus$, using $A \oplus B \equiv (A \vee B) \wedge (\neg A \vee \neg B)$.
2 Eliminate all occurrences of $\Leftrightarrow$, using $A \Leftrightarrow B \equiv (A \Rightarrow B) \wedge (B \Rightarrow A)$.
3 Eliminate all occurrences of $\Rightarrow$ using $A \Rightarrow B \equiv \neg A \vee B$.
4 Use De Morgan's Laws to push $\neg$ inward over $\wedge$ and $\vee$.
5 Eliminate double negations using $\neg\neg A \equiv A$.
6 Use the distributive laws to get the required form.

**Example Conversion to CNF**

$(\neg P \wedge(\neg Q \Rightarrow R))\Leftrightarrow S$

$\equiv ((\neg P \wedge(\neg Q \Rightarrow R))\Rightarrow S)\wedge(S \Rightarrow(\neg P \wedge(\neg Q \Rightarrow R)))$ (2)

$\equiv (\neg(\neg P \wedge(\neg Q \Rightarrow R))\vee S)\wedge(\neg S \vee(\neg P \wedge(\neg Q \Rightarrow R)))$ (3)

$\equiv (\neg(\neg P \wedge(\neg\neg Q \vee R))\vee S)\wedge(\neg S \vee(\neg P \wedge(\neg\neg Q \vee R)))$ (3)
$\equiv ((\neg\neg P \vee(\neg\neg\neg Q \wedge\neg R))\vee S)\wedge(\neg S \vee(\neg P \wedge(\neg\neg Q \vee R)))$ (4)
$\equiv ((P \vee(\neg Q \wedge\neg R))\vee S)\wedge(\neg S \vee(\neg P \wedge(Q \vee R)))$ (5)
$\equiv (((P \vee\neg Q)\wedge(P \vee\neg R))\vee S) \wedge((\neg S \vee\neg P)\wedge(\neg S \vee(Q \vee R)))$ (6)
$\equiv (P \vee\neg Q \vee S)\wedge(P \vee\neg R \vee S) \wedge(\neg S \vee\neg P)\wedge(\neg S \vee Q \vee R)$ (6)

**Empty Clauses**

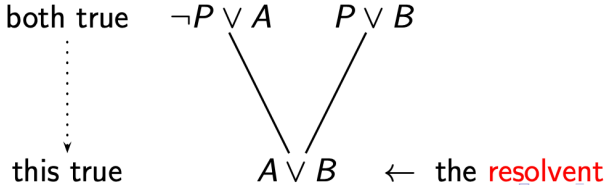
Clause {A, B} represents $A \vee B$ , and clause {A} represents A.
Empty clause is the empty set of literals, written $\varnothing$ or $\bot$, and corresponds to the formula **f**
Any set $\{\varnothing, \ldots\}$ of clauses is unsatisfiable.

**Empty CNF Formulas**

The formula $\{C_1, C_2\}$, with clauses $C_1$ and $C_2$, represents $C_1 \wedge C_2$. The formula {C} represents C.
(CNF) formula $\varnothing$: The natural reading is that it is true, C1 $\wedge$ C2 same as **t** $\wedge$ $C_1 \wedge C_2$, and C same as **t** $\wedge$ C.
The set $\varnothing$ of clauses is valid in CNF. In particular, note that $\{\varnothing\} \neq \varnothing$.

**Resolution-Based Inference**

Consider the two clauses $\neg P \vee A$ and $P \vee B$.
If P is true, they reduce to A and t. If P is false, they reduce to t and B.
There are no other possible values for P, so: if $(\neg P \vee A)$ and $(P \vee B)$ are both true, then either A is true or B is true (or both). That is, the clause $A \vee B$ is a logical consequence of the two original clauses.
You can draw the clauses {A, $\neg$P} and {P, B} and find the union and cancel out the P and negation P. We get {A,B} at last, which represents $A \vee B$. We call $A \vee B$ their resolvent.




**Propositional Resolution Generally**

Let $C_1$ and $C_2$ be clauses such that $P \in C_1$ and $\neg P \in C_2$.
$(C_1 \setminus \{P\}) \cup (C_2 \setminus \{\neg P\})$ is a resolvent of $C_1$ and $C_2$.
(Note: $A \setminus B$ is set difference: the set of elements in A but not in B.)
**Theorem**. If R is a resolvent of $C_1$ and $C_2$ then $C_1 \wedge C_2 \models R$.
I.e. for the last example, we can get $(\neg P \vee A) \wedge (P \vee B) \models A \vee B$
This generalises the well-known inference rule of modus ponens: From A and $A \Rightarrow B$ deduce B.

**Refuting a Set of Clauses**

Resolution suggests a way of **verifying** (proving) that a CNF formula is unsatisfiable.
If, through a number of resolution steps, we can derive the empty clause $\bot$, then the original set of clauses were unsatisfiable. i.e. if formula $\models$ f, the formula is unsatisfiable. We talk about a refutation proof: reduce a problem to show some set of clauses is satisfiable then show that via resolution. Theorem: $F \models G$ iff $F \wedge \neg G$ is unsatisfiable.

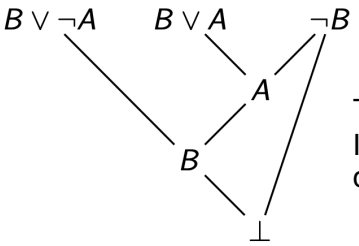
**Deductions and Refutations**

A resolution deduction of clause C from a set S of clauses is a finite sequence $C_1, C_2, \ldots, C_n$ of clauses such that $C_n = C$ and for each i, $1 \le i \le n$ , $C_i$ is either

- a member of S, or
- a resolvent of $C_j$ and $C_k$, for some $j, k < i$.

A resolution refutation of a set S of clauses is a resolution deduction of $\bot$ from S.

**An Example of a Refutation**




This shows that $(B \vee \neg A) \wedge (B \vee A) \wedge \neg B$ is unsatisfiable.
In refutation, the clauses can be used as many time as you want because it's considered to be true.

**How to Use Refutations**

To show that F is valid, first put ¬F in RCNF, yielding a set S of clauses.
Then refute S, that is, deduce $\bot$ from S.

Consider: $(P \Rightarrow R) \vee (R \Rightarrow (P \wedge Q))$

Negating yields: $\neg((\neg P \vee R) \vee (\neg R \vee (P \wedge Q)))$

Pushing negation then yields: $P \wedge \neg R \wedge R \wedge (\neg P \vee \neg Q)$

From this we can derive $\bot$ in a single resolution step.

Suppose we express a circuit design as a formula F in RCNF. Suppose we wish to show that the design satisfies some property G, that is, show F |= G.
We can exploit this fact: F |= G iff $F \wedge \neg G$ is unsatisfiable
Hence a strategy is:
1 Negate G and bring it into RCNF;
2 add those clauses to the set F; and
3 find a refutation of the resulting set of clauses.

When **Proving F ≡ G**, use resolution on ¬(F ⇔ G)

When **refuting F ≡ G**, find a truth-assignment in F or G, but not in the other
When refuting {A, B, C} |= F, finding a truth-assignment in {A, B, C} but not in F
**Proving satisfiability** always consider finding a truth value first, otherwise tempt resolution

Example Problem Prove $((A \vee B) \Rightarrow C) \wedge B \models C$

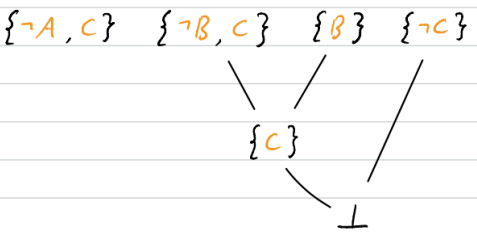
Solution

1. Invoke theorem and replace problem with showing $((A \vee B) \Rightarrow C) \wedge B \wedge \neg C$ is unsatisfiable

2. Convert formula to Conjunctive Normal Form: $(\neg A \vee C) \wedge (\neg B \vee C) \wedge B \wedge \neg C$

3. Replace CNF formula with set of clauses $\{\{\neg A, C\}, \{\neg B, C\}, \{B\}, \{\neg C\}\}$

4. Refute the set (show they are jointly-unsatisfiable) via resolution.



**Predicate Logic**
Refines the concept of an atomic proposition (P, Q, R etc) to a reusable pattern that describes a property of some number of things.

**Quantifier Syntax:**
Universal quantification, $\forall$, is generalised $\wedge$, which is used to generalize an assertion that everything satisfies some formula e.g. $\forall x$: for all x in domain, $\forall y$: for all y in domain
Existential quantification, $\exists$, is generalised $\vee$, which is used to generalize an assertion that at least one thing satisfies the formula. E.g. $\exists x$: at least one x in domain, $\exists y$: at least one y in domain.

**Terminology**
A term is a reference to some object. It is a generalization of a name for an object. A term can be a
- Variable - a term whose target object can be varied by the quantifier that captures it (x, y, z, u, v, w, …)
- constant - a fixed name for some object (a, b, c, d, e, …, $a_1$, $a_2$, $a_3$, …, 0,1, 2, …, tom, …)
- construction - $f(t_1,...,t_n)$ where n > 0, f is a function symbol of arity n, and each $t_i$ is a term, the construction refers to some object via its relation to $t_1$, ., $t_n$ via f. arity: a number that says how many arguments the function takes. Each predicate symbol similarly comes with an arity.

An atomic formula (or atom) is a construction $P(t_1,...,t_n)$ where $n \geq 0$ and P is a predicate symbol of arity n, and each $t_i$ is a term. States some property or relation that holds of the things referred to by the terms
- predicate symbols (P, Q, R, A, B, ..., <, =) predicate starts with an upper case letter; nothing else does.
- father(Ron) is a term; it denotes some object. (Most likely we intend this to mean: “the father of Ron”)
- Father(ron) is a formula; it denotes a truth value. (Most likely we intend this to mean: “Ron is a father”)

Term ⟷ Individual, object
Atom ⟷ Assertion (false or true)
A literal is an atomic formula or its negation.

**Well-formed formulas** (wffs) for predicate logic are generated by the grammar
Formula → atom | ¬wff | wff $\wedge$ wff | wff $\vee$ wff | wff $\Rightarrow$ wff | wff $\Leftrightarrow$ wff | wff $\oplus$ wff | $\forall$var(wff) | $\exists$var(wff) | (wff)

Which is a construction on atomic formulas using propositional connectives or quantifiers

**Bound and Free Variables**
A variable which is in the scope of a quantifier (binding that variable) is bound.
If it is not bound then it is free.
A variable may occur both free and bound in a formula — witness y in $\forall z\,(P(x,y,z) \wedge \forall y\,(P(f(x),z,y)))$
A formula with no free variable occurrences is closed.
It is possible for scopes to have “holes”: $\forall x \exists y\,(x < y \wedge \exists x\,(y < x))$
The last occurrence of x is bound by the closest quantifier, so the scope of $\forall x$ is not all of $\exists y(\ldots)$.

**Bound Variable Renaming and Capture**
The bound variable of a quantified formula is just a placeholder—its exact name is inessential.
$\exists x \forall y\,(x < y)$ means the same as $\exists x \forall z\,(x < z)$.
If a variable occurs bound in a certain expression then the meaning of that expression does not change when all bound occurrences of that variable are replaced by another one.
However, to avoid variable capture, we cannot change the variable bound by $\forall y$ to a variable in an enclosing scope: $\exists x \forall y\,(x \leq y)$ is very different to $\exists x \forall x\,(x \leq x)$.

**Word Order**
“There is something which is not P”: $\exists y\, \neg P(y)$
“There is not something which is P” (“nothing is P”): $\neg \exists y\, P(y)$
“All S are not P” vs “not all S are P:” $\forall x\,(S(x) \Rightarrow \neg P(x))$ vs $\neg \forall x\,(S(x) \Rightarrow P(x))$

**Quantifier Order**
$\forall x \exists y$ is not the same as $\exists y \forall x$. The former says each x has a y that satisfies P(x,y); the latter says there’s an individual y that satisfies P(x,y) for every x.
But $\forall x \forall y$ is the same as $\forall y \forall x$ and $\exists x \exists y$ is the same as $\exists y \exists x$.

**Expressiveness of Predicate Logic: Samples** (left is natural language, right is predicate logic)

No emus fly: $\forall x\ (Emu(x) \Rightarrow \neg Flies(x))$

There are black swans: $\exists x\ (Black(x) \wedge Swan(x))$

Tom found Rover and returned him to Anne: $Found(tom, rover) \wedge Gave(tom, rover, anne)$

Tom found a dog and gave it to Anne: $\exists x\ (Dog(x) \wedge Found(tom, x) \wedge Gave(tom, x, anne))$

//here dog don't have a name and it could be any name. The next expression follows the same rule

Jill inhabits the house that Jack built: $\exists x\ (House(x) \wedge Inhabits(jill, x) \wedge BuilderOf(jack, x))$

Mothers' mothers are grandmothers: $\forall x,y,z((Mother(x, y) \wedge Mother(y,z)) \Rightarrow Grandmother(x,z))$

**Introduce symbols for predicates**

Sentence: "He is a man." Predicate: is a man Predicate Symbol: M( )

"x is a man", M(x), cannot be assigned a truth value.

Kim is a man, M(kim), can be assigned a truth value.

Sentence: "Alice is taller than Kim" T(alice, kim) //here order of Alice & Kim matters! The terms may play different rolls in the formula

**Quantifier examples:**

"Every human is mortal" $\forall x\ (Human(x) \Rightarrow Mortal(x))$

"Some cat is mortal" $\exists x\ (Cat(x) \wedge Mortal(x))$ Note: Very common to use $\Rightarrow$ with $\forall$ and $\wedge$ with $\exists$.

Let L(x, y) stand for "x loves y". Let I(x, y) stand for "x is y". //here we are giving meaning to predicate L, it takes 2 variables

$\forall x\ (\neg I(x , eva) \Rightarrow L(x , eva))$ Eva is loved by everyone else

$\exists x\ (\neg I(x , bob) \wedge L(x , bob))$ Someone other than Bob loves Bob

$\forall x\ (\exists y\ L(x, y))$ Everybody loves somebody

$\exists y\ (\forall x\ L(x, y))$ Someone is loved by everybody

$\exists x\ (\forall y\ L(x, y))$ Someone loves everybody

**Interpretations (or Structures)**

The role of interpretation is to decide what denotes what.

An interpretation (or structure) consists of

1 A non-empty set D (the domain of objects, or universe, sometimes called $I_0$);
- Finite sets: {3,5,7} {a, b} {0, 1}… Infinite sets: N, Z, R, {f : N -> N} (set of functions)…
- This is collection of things that the interpretation decides the formula to be talking about

2 An assignment, to each n-ary predicate symbol P, of an n-place truth-function p : $D^n \rightarrow$ {f, t};
- P (symbol) —> I(P)(semantics) : $D^n$ —> {**f, t**}
- I(P) is a function which takes n-many objects as arguments and produce an truth value as output

3 An assignment, to each n-ary function symbol g, of an n-place function g : $D^n \rightarrow D$;
- I.e. *I*(f) is a function takes n object as arguments, and output an object.
- g is not a function, it's just a name for function
- *I* decides which g names

4 An assignment to each constant a of some fixed element of D (syntax a -> semantics *I*(a)).

*I* decides a is a name for the object *I*(a) belongs to $I_D$

**Terms and Valuations**

Valuation is a function σ(read as sigma) : *var* → *D*.

Takes a particular object in domain that the variable refers to.

It gives which object does each term denotes

The interpretation, *I*, and valuation σ collaborate to decide which object each term denotes. For any term t, write I(σ, t) for the object it denotes.

I(*σ,t)* is defined recursively on the term t
- $I(\sigma, a) = I(a)$
- $I(\sigma, x) = \sigma(x)$
- $I(\sigma, f(t_1, t_2, \ldots, t_n)) = I(f)(I(\sigma, t_1), \ldots, I(\sigma, t_n))$
- *E.g.* $I(\sigma, f(y, g(a, x)) = I(f)(I(\sigma, y), I(\sigma, g(a, x))) = I(f)(\sigma(y), I(g)()I(\sigma, a), I(\sigma, x)) = I(f)(\sigma(y), I(g)(I(a), \sigma(x)))$

But, given an interpretation *I*, we get a valuation function from terms automatically, by natural extension:

$\sigma(a) = d = I(a)$ $\sigma(g(t_1,\ldots,t_n)) = \mathbf{g}(\sigma(t_1),\ldots,\sigma(t_n))$

where *d* is the element of *D* that *I* assigns to *a*, and *g*: $D^n \rightarrow D$ is the function that *I* assigns to *g*.

Example: Consider term t = f (y , g(x , a)). Let our interpretation (with domain Z) assign to a the value 3, to f the multiplication function, and to g addition. If σ(x) = 9 and σ(y) = 5 then σ(t) = 60.

**Truth of a Formula**
The truth of a closed formula should depend only on the given interpretation.
For any valuation, *σ: vars —> D*, a variable x, and an object d belongs to D, we can define a new valuation σx→d : vars —> D defined

$$\sigma_{x\to d}(y) = \begin{cases} d & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

//Read this as "the map σ, updated to map x to d."
Where σx→d is the same as σ, except σx→d sends x to d
Chain multiple modifications x—>$d_1$, y—>$d_2$, like this $\sigma[x\text{—>}d_1, y\text{—>}d_2]$ to mean $\sigma_{(x\text{—>}d1)(y\text{—>}d2)}$

**Interpreting a quantified formulas:**
I(σ, ∀xG) = t iff I(σx→d, G) = t for every d belongs to D
I(σ, ∃xG) = t iff I(σx→d, G) = t for at least one d belongs to D
Example interpretation:
Domain = {Jemima, Thelma, Louise} //these are objects, not terms
Constants: I(a) = Jemima, I(b) = Thelma, I(c) = Louise
Function symbols: …don't care

Predicates :

| I(F) | Jemima | Thelma | Louise |
|---|---|---|---|
| Jemima | f | f | t |
| Thelma | t | t | f |
| Louise | f | t | f |

| I(M) | |
|---|---|
| Jemima | t |
| Thelma | f |
| Louise | t |

I(F)(Louise, Thelma) = t

Valuation : σ(x) = Louise , σ(y) = Thelma , σ(z) = Jemima

$$\begin{aligned} I(\sigma, F(x,a)) &= I(F)(I(\sigma,x), I(\sigma,a)) \\ &= I(F)(\sigma(x), I(a)) \\ &= I(F)(\text{Louise}, \text{Jemima}) \\ &= f \end{aligned}$$

$$\begin{aligned} I(\sigma, \exists y\, F(x,y)) &= t \quad \text{because} \\ I(\sigma_{y\mapsto \text{Thelma}}, F(x,y)) &= I(F)(\sigma_{y\mapsto\text{Thelma}}(x), \sigma_{y\mapsto\text{Thelma}}(y)) \\ &= I(F)(\text{Louise}, \text{Thelma}) \\ &= t \end{aligned}$$

**Making a Formula True**
Given an interpretation I (with domain D), and a valuation σ,
σ makes $P(t_1,...,t_n)$ true iff $p(\sigma(t_1),...,\sigma(t_n)) = t$, where p is the meaning that I gives P.
σ makes ¬F true iff σ does not make F true.
σ makes $F_1 \wedge F_2$ true iff σ makes both of $F_1$ and $F_2$ true.
σ makes ∀x F true iff $\sigma_{x\to d}$ makes F true for every d ∈ D.
If we now define ∃x F ≡ ¬∀x¬F then the meaning of every other formula follows from this.

**Rules of Passage for the Quantifiers**
∃x (¬$F_1$) ≡ ¬∀x $F_1$
∀x (¬$F_1$) ≡ ¬∃x $F_1$
∃x ($F_1 \vee F_2$) ≡ (∃x $F_1$) ∨ (∃x $F_2$)
∀x ($F_1 \wedge F_2$) ≡ (∀x $F_1$) ∧ (∀x $F_2$)
It follows that
∃x ($F_1 \Rightarrow F_2$) ≡ (∀x $F_1$) ⇒ (∃x $F_2$)

If G is a formula with no free occurrences of x, then we also get
∃x G ≡ G
∀x G ≡ G
∃x (F ∧ G) ≡ (∃x F) ∧ G
∀x (F ∨ G) ≡ (∀x F) ∨ G
∀x (F ⇒ G) ≡ (∃x F) ⇒ G

∀x (G ⇒ F) ≡ G ⇒ (∀x F)

no matter what F is. In particular F may have free occurrences of x.

**Models and Validity of Formulas**
A wff F is true in interpretation *I* iff every valuation makes F true (for *I*). If not true then it is false in interpretation *I*.
A model for F is an interpretation *I* such that F is true in *I*. We write *I* |= F.
A wff F is logically valid iff every interpretation is a model for F. In that case we write |= F.
$F_2$ is a logical consequence of $F_1$ iff *I* |= $F_2$ whenever *I* |= $F_1$. We write $F_1 \models F_2$.
$F_1$ and $F_2$ are logically equivalent iff $F_1 \models F_2$ and $F_2 \models F_1$. We write $F_1 \equiv F_2$.

**A closed, well-formed formula F is**
satisfiable iff *I* |= F for some interpretation *I*;
valid iff *I* |= F for every interpretation *I*;
unsatisfiable iff *I* /|= F (F is not logical consequence of *I*) for every interpretation *I*;
non-valid iff *I* /|= F (F is not logical consequence of *I*) for some interpretation *I*.
As in the propositional case, we have F is valid iff ¬F is unsatisfiable; F is non-valid iff ¬F is satisfiable.
**Example of Non-Validity**
Consider the formula $(\forall y \exists x\ P(x, y)) \Rightarrow (\exists x \forall y\ P(x, y))$

It is not valid. For example, consider the interpretation with domain D = Z, and the predicate P meaning "less than". Or, let D = {0, 1} and let P mean "equals".
The formula is satisfiable, as it is true, for example, in the interpretation where D = {0, 1} and P means "less than or equal".
**Example of Validity**
F = $(\exists y \forall x\ P(x, y)) \Rightarrow (\forall x \exists y\ P(x, y))$ is valid.

If we negate F (and rewrite it) we get $(\exists y \forall x\ P(x, y)) \wedge (\exists x \forall y\ \neg P(x, y))$
The right conjunct is made true only if there is some $d_0 \in D$ for which $p(d_0,d)$ is false for all $d \in D$.
But the left conjunct requires that $p(d_0,d)$ be true for at least some d.
Since F's negation is unsatisfiable, F is valid.
**Another Example of Validity**
Consider F = $(\forall x P(x)) \Rightarrow P(t)$

F is valid no matter what the term t is.
To see this, again it is easiest to consider ¬F = $(\forall x\ P(x)) \wedge \neg P(t)$
The term t denotes some element of the domain D, so ¬F cannot be satisfied.

**Resolution for Predicate Logic**
Goal: show a predicate logic formula is unsatisfiable via resolution
Definition: G is satisfiable iff there is some interpretation I such that I(v, G) = t for some valuation v(i.e. I|=G)
Obstacle one: quantifiers get in the way of CNF
Idea: construct a new equisatisfiable formula without quantifiers
Skolemization: Existential quantifiers are eliminated in a this process
Theorem (skolemization): Suppose the following is true of a formula G with no free variables
- G is free of connectives ⇒, ⇔, ⊕
- All negations in G only appear on atomic formulas
- There is an ∃z(…) inside the scope of universal quantifiers $\forall x_1$, $\forall x_2$, $\forall x_3$…
- The n-place function symbol f does not appear in anywhere of G

Then the formula H obtained by replacing each instance of z inside ∃z(…) by $f(x_1, x_2, x_3, \ldots, x_n)$, and removing the ∃z, is equisatisfiable to G, but H is not logical equivalent to G.

**From Predicate Logic Formulas to Clausal Form**
The process is similar to what we did with propositional formulas:
Replace occurrences of ⊕, ⇔, and ⇒.

Drive negation in.
Standardise bound variables apart.
Eliminate existential quantifiers (Skolemize).
Eliminate universal quantifiers (just remove them).
Bring to CNF (using the distributive laws).

**Skolemization Example**
This formula has three existential quantifiers—we remove them one by one:
If ∃ not bounded by ∀, substitute by a fresh constant (Skolem constant), in fact it is a 0-place function.
Skolem constant must be fresh, Fresh means the formula/constant cannot appear anywhere in the formula before.
∃u∀v∃x∀y∃z ((¬P(u, f(v), x, b) ∨ R(g(x, y), u)) ∧ S(y, g(a, z)))
—> ∀v∃x∀y∃z ((¬P(c, f(v), x, b) ∨ R(g(x, y), c)) ∧ S(y, g(a, z)))
—> ∀v∀y∃z ((¬P(c, f(v), h(v), b) ∨ R(g(h(v), y), c)) ∧ S(y, g(a, z)))
—> ∀v∀y ((¬P(c, f(v), h(v), b) ∨ R(g(h(v), y), c)) ∧ S(y, g(a, j(v, y))))
Instead of j(v, y) we could have chosen k(v, y), or even j(y, v)—as long as we replace each occurrence of z by the same term, of course.
∃x∀y∃z(∀w P(x, y, w, z) ∨ ∀u∃v Q(x, u, v, z))
——> ∀y(∀wP(a, y, w, f, f(y)) ∨ ∀)
Then we can simply drop the universal quantifiers by theorem:
If (I(σ, ∀xG) = t for all σ) then I(σ, G) = t for all σ

**Clausal Form: Step 1—Use Just ∨, ∧, ¬**
Let us use this running example:
∀x (P(x) ⇔ ∃y (R(x, y) ∧ ∀z R(z, y)))

First use the usual translations to eliminate ⇔ (and ⇒ and ⊕):

$$\forall x \left( \begin{array}{l} (P(x) \Rightarrow \exists y\ (R(x,y) \wedge \forall z\ R(z,y))) \wedge \\ \quad (\exists y\ (R(x,y) \wedge \forall z\ R(z,y)) \Rightarrow P(x)) \end{array} \right)$$

$$\forall x \left( \begin{array}{l} (\neg P(x) \vee \exists y\ (R(x,y) \wedge \forall z\ R(z,y))) \wedge \\ \quad (\neg \exists y\ (R(x,y) \wedge \forall z\ R(z,y)) \vee P(x)) \end{array} \right)$$

**Clausal Form: Step 2—Push Negation**
Next drive negation in.

$$\forall x \left( \begin{array}{l} (\neg P(x) \vee \exists y\ (R(x,y) \wedge \forall z\ R(z,y))) \wedge \\ \quad (\neg \exists y\ (R(x,y) \wedge \forall z\ R(z,y)) \vee P(x)) \end{array} \right)$$

$$\forall x \left( \begin{array}{l} (\neg P(x) \vee \exists y\ (R(x,y) \wedge \forall z\ R(z,y))) \wedge \\ \quad (\forall y\ (\neg R(x,y) \vee \exists z\ \neg R(z,y)) \vee P(x)) \end{array} \right)$$

**Clausal Form: Step 3—Standardize Apart**
Now rename variables so that no two quantifiers use the same variable name. With that,

$$\forall x \left( \begin{array}{l} (\neg P(x) \vee \exists y\ (R(x,y) \wedge \forall z\ R(z,y))) \wedge \\ \quad (\forall y\ (\neg R(x,y) \vee \exists z\ \neg R(z,y)) \vee P(x)) \end{array} \right)$$

$$\forall x \left( \begin{array}{l} (\neg P(x) \vee \exists y\ (R(x,y) \wedge \forall z\ R(z,y))) \wedge \\ \quad (\forall u\ (\neg R(x,u) \vee \exists v\ \neg R(v,u)) \vee P(x)) \end{array} \right)$$

**Clausal Form: Step 4—Skolemize**
Let us highlight the existentially quantified variables:

$$\forall x \left( \begin{array}{l} (\neg P(x) \vee \exists y\ (R(x,y) \wedge \forall z\ R(z,y))) \wedge \\ \quad (\forall u\ (\neg R(x,u) \vee \exists v\ \neg R(v,u)) \vee P(x)) \end{array} \right)$$

The existentially quantified y is in the scope of ∀x—so replace it by f(x).
The existentially quantified v is in the scope of ∀x, as well as of ∀u. So we replace it by g(u,x).

$$\forall x \left( \begin{array}{l} (\neg P(x) \vee (R(x, f(x)) \wedge \forall z\ R(z, f(x)))) \wedge \\ \quad (\forall u\ (\neg R(x,u) \vee \neg R(g(u,x), u)) \vee P(x)) \end{array} \right)$$

**Clausal Form: Step 5—Drop Universal Quantifiers**
Eliminating universal quantifiers is easy:

$$\forall x \left( \begin{array}{l} (\neg P(x) \vee (R(x, f(x)) \wedge \forall z\ R(z, f(x)))) \wedge \\ \quad (\forall u\ (\neg R(x,u) \vee \neg R(g(u,x), u)) \vee P(x)) \end{array} \right)$$

becomes
(¬P(x)∨(R(x, f (x))∧R(z, f (x))))∧(¬R(x, u)∨¬R(g(u, x), u)∨P(x))

**Clausal Form: Step 6—Convert to CNF**
becomes, using distribution:
$(\neg P(x) \vee R(x, f(x))) \wedge (\neg P(x) \vee R(z, f(x))) \wedge (\neg R(x,u) \vee \neg R(g(u,x),u) \vee P(x))$
or, written as a set of sets of literals:

$$\left\{ \begin{array}{c} \{\neg P(x), R(x, f(x))\}, \\ \{(\neg P(x), R(z, f(x))\}, \\ \{(\neg R(x, u), \neg R(g(u, x), u), P(x))\} \end{array} \right\}$$

**Substitutions**
A substitution is a finite set of replacements of variables by terms, that is, a set θ of the form $\{x_1 \rightarrow t_1, x_2 \rightarrow t_2, \ldots, x_n \rightarrow t_n\}$, where the $x_i$ are variables and the $t_i$ are terms.

We can also think of θ as a function from terms to terms, or from atomic formulas to atomic formulas.
θ(F) is the result of simultaneously replacing each occurrence of $x_i$ in F by $t_i$ .

Example: If F is P(f(x), g(y, y, b)), and θ = {x → h(u), y → a, z → c} then θ(F) is P(f(h(u)), g(a, a, b)).
Note: Substitution maps a variable to a term, and, by natural extension, terms to terms.

**Most General Unifiers**
A unifier of two terms s and t is a substitution θ such that θ(s) = θ(t).
The terms s and t are unifiable iff there exists a unifier for s and t. A most general unifier (mgu) for s and t is a substitution θ such that

θ is a unifier for s and t, and

every other unifier σ of s and t can be expressed as τ ◦ θ for some substitution τ.

The composition τ◦θ is the substitution we get by first using θ, and then using τ on the result produced by θ.
**Theorem**. If s and t are unifiable, they have a most general unifier.

**Unifier Examples**

P(x, a) and P(b, c) are not unifiable.

P(f(x), y) and P(a, w) are not unifiable.

P(x, c) and P(a, y) are unifiable using {x → a, y → c}.

P(f(x), c)and P(f(a), y)also unifiable using {x→a, y→c}.

Note: P(x) and P(f(x)) are not unifiable.

The last case relies on a principle that a variable (such as x) is not unifiable with any term containing x (apart from x itself).
If we were allowed to have a substitution {x → f (f (f (. . .)))}, that would be a unifier for the last example. But we cannot have that, as terms must be finite.

**More Unifier Examples**
Now consider P(f(x), g(y, a)) and P(f(a), g(z, a)). The following are all unifiers, so which is “best”?
{x → a, y → z}
{x → a, y → a, z → a}
{x → a, y → g(b, f(u)), z → g(b, f(u))} {x → a, z → y}
The first and the last are mgus. They avoid making unnecessary commitments. The second commits y and z to take the value a, which was not really needed in order to unify the two formulas.
Note that {x → a, y → a, z → a} = {z → a} ◦ {x → a, y → z}.

**A Unification Algorithm**
In the following, let x be a variable, let F and G be function or predicate names, and let s and t be arbitrary terms.
Input: Two terms s and t.
Output: If they are unifiable: a most general unifier for s and t; otherwise ‘failure’.
Algorithm: Start with the set of equations {s = t}. (This is a singleton set: it has one element.)

**Unification: Solving Term Equations**
1. $F(s_1,\ldots,s_n) = F(t_1,\ldots,t_n)$: • Replace the equation by the n equations $s_1 = t_1,\ldots,s_n = t_n$.
2. $F(s_1,\ldots,s_n) = G(t_1,\ldots,t_m)$ where F /= G or n /= m: • Halt, returning ‘failure’.
3. x = x: • Delete the equation.
4. t = x where t is not a variable: • Replace the equation by x = t.
5. x = t where t is not x but x occurs in t: • Halt, returning ‘failure’.
6. x = t where t contains no x but x occurs in other equations: • Replace x by t in those other equations.

**Solving Term Equations: Example 1**
Starting from f (h(y), g(y, a), z) = f(x, g(v, v), b)

$$\overset{(1,4)}{\Longrightarrow} \begin{array}{rcl} x & = & h(y) \\ g(y,a) & = & g(v,v) \\ z & = & b \end{array} \overset{(1,4)}{\Longrightarrow} \begin{array}{rcl} x & = & h(y) \\ y & = & v \\ v & = & a \\ z & = & b \end{array} \overset{(6,6)}{\Longrightarrow} \begin{array}{rcl} x & = & h(a) \\ y & = & a \\ v & = & a \\ z & = & b \end{array}$$

The last set is in normal form and corresponds to the substitution
θ = {x → h(a), y → a, v → a, z → b} which indeed unifies the two original terms.

**Solving Term Equations: Example 2**
Starting from: f(x, a, x) = f(h(z , b), y, h(z, y))

$$\overset{(1,4)}{\Longrightarrow} \begin{array}{rcl} x & = & h(z,b) \\ y & = & a \\ x & = & h(z,y) \end{array} \overset{(6)}{\Longrightarrow} \begin{array}{rcl} x & = & h(z,b) \\ y & = & a \\ h(z,b) & = & h(z,y) \end{array} \overset{(1,4)}{\Longrightarrow} \begin{array}{rcl} x & = & h(z,b) \\ y & = & a \\ z & = & z \\ y & = & b \end{array}$$

$$\overset{(3)}{\Longrightarrow} \begin{array}{rcl} x & = & h(z,b) \\ y & = & a \\ y & = & b \end{array} \overset{(6)}{\Longrightarrow} \begin{array}{rcl} x & = & h(z,b) \\ y & = & a \\ a & = & b \end{array} \overset{(2)}{\Longrightarrow} \text{failure}$$

So the two original terms are not unifiable.

**Solving Term Equations: Example 3**
Starting from f (x, g (v, v), x ) = f (h(y), g(y, z), z)

$$\overset{(1)}{\Longrightarrow} \begin{array}{rcl} x & = & h(y) \\ v & = & y \\ v & = & z \\ z & = & h(y) \end{array} \overset{(6)}{\Longrightarrow} \begin{array}{rcl} x & = & h(y) \\ z & = & y \\ v & = & z \\ z & = & h(y) \end{array} \overset{(6)}{\Longrightarrow} \begin{array}{rcl} x & = & h(y) \\ z & = & y \\ v & = & y \\ y & = & h(y) \end{array} \overset{(5)}{\Longrightarrow} \text{failure}$$

This is “failure by occurs check”: The algorithm fails, as soon as we discover the equation y = h(y).

**Term Equations as Substitutions**
The process of solving term equations always halts.
When it halts without reporting ‘failure’, the term equation system is left in a normal form: On the left-hand sides we have variables only, and they are all different. Moreover, these variables occur nowhere in the right-hand sides.
If the normal form is $\{x_1 = t_1,...,x_n = t_n\}$ then $\{x_1 \rightarrow t_1,...,x_n \rightarrow t_n\}$ is a most general unifier for the input terms s and t. If the result is ‘failure’, no unifier exists.

**Resolvents**
Two literals L and ¬L′ are complementary if {L, L′} is unifiable.
Let $C_1$ and $C_2$ be clauses, renamed apart (fresh variable names). Let θ be an mgu of complementary literals {L, ¬L′} with L a literal in $C_1$ and ¬L′ a literal in $C_2$. Then the resolvent of $C_1$ and $C_2$ is the union $\theta(C_1 \setminus \{L\}) \cup \theta(C_2 \setminus \{\neg L'\})$.

**Automated Inference with Predicate Logic**

| | |
|---|---|
| Every shark eats a tadpole | $\forall x(S(x) \Rightarrow \exists y(T(y) \wedge E(x, y)))$ |
| All large white fish are sharks | $\forall x(W(x) \Rightarrow S(x))$ |
| Colin is a large white fish living in deep water | $W(colin) \wedge D(colin)$ |
| Any tadpole eaten by a deep water fish is miserable | $\forall z((T(z) \wedge \exists y(D(y) \wedge E(y, z))) \Rightarrow M(z))$ |
| Therefore some tadpole is miserable | $\therefore \exists z(T(z) \wedge M(z))$ |

**Tadpoles in Clausal Form**

Every shark eats a tadpole //idea: replace y by f(x)

• ∀x (S(x) ⇒ ∃y (T(y) ∧ E(x, y))) {¬S(x), T(f(x))}, {¬S(x), E(x, f(x))}

All large white fish are sharks

• ∀x (W(x) ⇒ S(x)) {¬W(x), S(x)}

Colin is a large white fish living in deep water

• W(colin) ∧ D(colin) {W(colin)}, {D(colin)}

Any tadpole eaten by a deep water fish is miserable

• ∀z((T(z) ∧ ∃y (D(y) ∧ E(y, z))) ⇒ M(z)) {¬T(z), ¬D(y), ¬E(y, z), M(z)}

Negation of: Some tadpole is miserable

• ¬∃z(T(z) ∧ M(z)) {¬T(z), ¬M(z)}

**A Refutation**

Let us find a refutation of the set of seven clauses.

To save space, we leave out braces and some parentheses, for example, we write ¬Wu, Su for clause {¬W(u), S(u)}.

¬Wu,Su ¬Sv,T(fv) ¬Tz,¬Dy,¬E(y,z),Mz

Wc ¬Sx,E(x,fx) Dc ¬Tw,¬Mw

Many different resolution proofs are possible

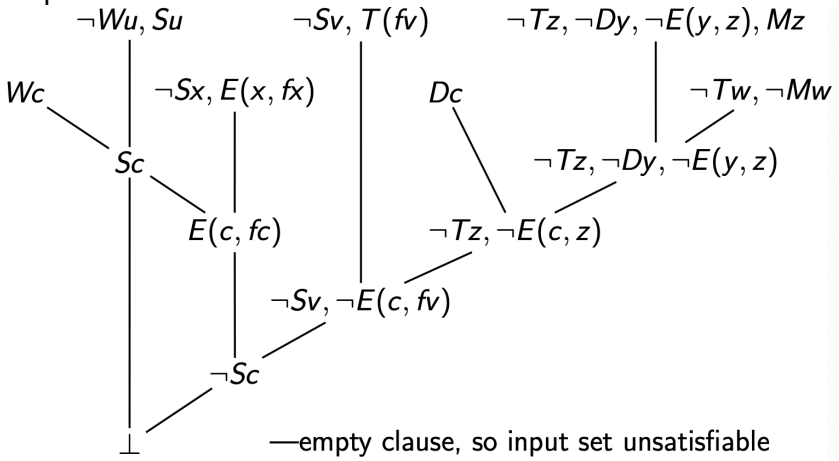
**A Refutation for the Tadpole Example**

Get u -> c cancel out Wc with not Wc

map w -> z

Replace y by c

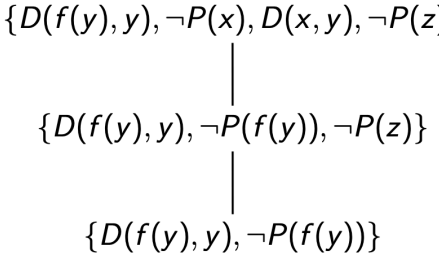
Map z -> fv




**Factoring**

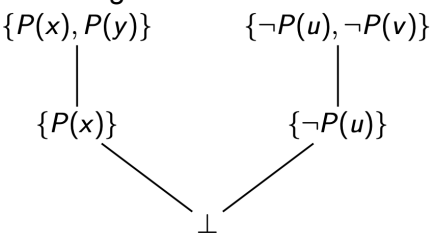
In addition to resolution, there is one more valid rewriting of clauses, called factoring.

Let C be a clause and let $A_1, A_2 \in C$. If $A_1$ and $A_2$ are unifiable with mgu θ, add the clause θ(C).




**The Need for Factoring**

Factoring is sometimes crucial:



**How to Use Clauses**
A resolution step uses two clauses (or two “copies” of the same clause). A factoring step uses one clause.
A given clause can be used many times in a refutation, taking part in many different resolution/factoring steps.
But recall that each clause is implicitly universally quantified.
Hence we really should rename all variables in a clause every time we use the clause, using fresh variable names.
Sometimes this renaming is essential for correctness, especially when resolution uses two “copies” of the same clause.

**The Resolution Method**
Start with collection C of clauses While $\perp \not\in C$ do
add to C a factor of some $C \in C$ or a resolvent of some $C_1, C_2 \in C$

**The Power of resolution**
**Theorem**. C is unsatisfiable iff the resolution method can add $\perp$ after a finite number of steps.
We say that resolution is sound and complete.
It gives us a proof procedure for unsatisfiability (and validity).
Note, however, that resolution does not give a decision procedure for unsatisfiability (or validity) of first-order predicate logic formulas.
Indeed, it can be shown that there are no such proof procedures.
We say that validity and unsatisfiability are semi-decidable) properties.

**Set Theory**
"Definition": (Georg Cantor) A set is a collection into a whole of definite, distinct objects of our intuition or of our thought. The objects are called the elements (members) of the set.
**Notation**: We write $a \in A$ to express that a is a member of set A. Examples: $42 \in \mathbb{N}$ and $\pi$ not $\in \mathbb{Q}$.
Order and repetition in set are irrelevant, e.g. {{1, 2, 2}, {1}, {2, 1}} = {{1}, {1, 2}}
{} often write as $\varnothing$.
If P is a property of objects x then the abstraction $\{x \mid P(x)\}$ denotes the set of things x that have the property P. Hence $a \in \{x \mid P(x)\}$ is equivalent to P(a). $\{x \in A \mid P(A)\} = \{x \mid x \in A \wedge P(A)\}$
Principle of Extensionality: For all sets A and B we have $A = B \Leftrightarrow \forall x\ (x \in A \Leftrightarrow x \in B)$

**The Subset Relation**
A is a subset of B iff $\forall x\ (x \in A \Rightarrow x \in B)$. We write this as $A \subseteq B$.

If $A \subseteq B$ and $A \neq B$, we say that A is a proper subset of B, and write this $A \subset B$.
Do not confuse $\subseteq$ with $\in$. e.g. We have $\{1\} \subseteq \{1, 2\}$, but $\{1\} \notin \{1, 2\}$. (not hold as '{1}' set is not in set {1, 2})

**Special Sets**
The empty set satisfies $\varnothing \subseteq A$ for every set A. i.e. $\Phi := \{x \mid \mathbf{f}\}$
Singleton. $\{a\} := \{x \mid x = a\}$ e.g. {{1, 2}} is a singleton (its only element is a set).
The set {a} should not be confused with its element a.

**Notation of ordered pairs**:
Defining $(a, b) = \{\{a\}, \{a, b\}\}$, Hence we can freely use the notation (a, b) with the intuitive meaning.

**Powersets**
The powerset P(X) of the set X is the set $\{A \mid A \subseteq X\}$ of all subsets of X.
In particular $\varnothing$ and X are elements of P(X). If X is finite, of cardinality n, then P(X) is of cardinality $2^n$.

**Cartesian Product and Tuples**
The Cartesian product of A and B is defined $A \times B = \{(a, b) \mid a \in A \wedge b \in B\}$
We define the set $A^n$ of n-tuples over A as follows: $A^0 = \{\varnothing\}$; $A^{n+1} = A \times A^n$

**Algebra of Sets** Let A and B be sets. Then

| | | |
|---|---|---|
| $A \cap B$ | $= \{x \mid x \in A \wedge x \in B\}$ | is the intersection of A and B; |
| $A \cup B$ | $= \{x \mid x \in A \vee x \in B\}$ | is their union; |
| $A \setminus B$ | $= \{x \mid x \in A \wedge x \notin B\}$ | is their difference; and |
| $A \oplus B$ | $= (A \setminus B) \cup (B \setminus A)$ | is their symmetric difference. |
| $A \times B$ | $= \{(x, y) \mid x \in A \wedge y \in B\}$ | is the product |
| $P(A)$ | $= \{Y \mid Y \subseteq A\}$ | is the Powerset |

$A^c = X \setminus A$ is the complement of A. Where x is the universal

| | | |
|---|---|---|
| $A = B$ | means $\forall x\ (x \in A \Leftrightarrow x \in B)$ | is equality |
| $A \subseteq B$ | means $\forall x\ (x \in A \Rightarrow x \in B)$ | subset |

**Laws**

| | |
|---|---|
| Absorption: | $A \cap A = A$ $\quad A \cup A = A$ |
| Commutativity: | $A \cap B = B \cap A$ $\quad A \cup B = B \cup A$ |
| Associativity: | $A \cap (B \cap C) = (A \cap B) \cap C$ |
| | $A \cup (B \cup C) = (A \cup B) \cup C$ |
| Distributivity: | $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ |
| | $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ |
| Double complement: | $A = (A^c)^c$ |
| DeMorgan: | $(A \cap B)^c = A^c \cup B^c$ |
| | $(A \cup B)^c = A^c \cap B^c$ |
| Duality: | $X^c = \varnothing$ and $\varnothing^c = X$ |
| Identity: | $A \cup \varnothing = A$ and $A \cap X = A$ |
| Dominance: | $A \cap \varnothing = \varnothing$ and $A \cup X = X$ |
| Complementation: | $A \cap A^c = \varnothing$ and $A \cup A^c = X$ |

**Subset Equivalences**
Subset characterization: $A \subseteq B \equiv A = A \cap B \equiv B = A \cup B$
Contraposition: $A^c \subseteq B^c \equiv B \subseteq A$
$A \subseteq B^c \equiv B \subseteq A^c$
$A^c \subseteq B \equiv B^c \subseteq A$

**Some Laws Involving Cartesian Product**
$(A \times B) \cap (C \times D) = (A \times D) \cap (C \times B)$
$(A \cap B) \times C = (A \times C) \cap (B \times C)$
$(A \cup B) \times C = (A \times C) \cup (B \times C)$
$(A \cap B) \times (C \cap D) = (A \times C) \cap (B \times D)$
$(A \cup B) \times (C \cup D) = (A \times C) \cup (A \times D) \cup (B \times C) \cup (B \times D)$

**Generalised Union and Intersection**
Suppose we have a collection of sets $A_i$ , one for each i in some (index) set i.
The union of the collection is $\bigcup_{i \in I} A_i = \{x \mid \exists i\ (i \in I\ \wedge x \in A_i)\}$
The intersection of the sets is $\bigcap_{i \in I} A_i = \{x \mid \forall i\ (i \in I\ \Rightarrow x \in A_i)\}$

**Relations**
An n-ary relation is a set of n-tuples. e.g. 4-nary relation
{(MY255,Lagos,Lusaka,1755), (ZA942 , Lima, London, 1015), (BB114, Lyon, Lodz, 2220)}
That is, the relation is a subset of some Cartesian product $A_1 \times A_2 \times \cdots \times A_n$.

**Binary Relations**
A binary relation is a set of pairs, or 2-tuples. “Being unifiable”, “<”, “$\subseteq$”, “divides” are all binary relations.

**Domain and Range of a Relation**
The domain of R is dom(R) = {x | ∃y R(x, y)}. The range of R is ran(R) = {y | ∃x R(x, y)}.
We say that R is a relation from A to B if dom(R) ⊆ A and ran(R) ⊆ B. Or, R is a relation between A and B.
A relation from A to A is a relation on A.
“Being unifiable” is a relation on Term. “<” is a relation on integers. “$\subseteq$” is a relation on P(A).
A × B is a relation — the full relation from A to B. ∅ is a relation.
$\Delta_A = \{(x, x) \mid x \in A\}$ is a relation on A — the identity relation.

If R is a relation from A to B then $R^{-1} = \{(b, a) \mid (a, b) \in R\}$ is a relation from B to A, called the inverse of R.

Clearly $(R^{-1})^{-1} = R$.
Since relations are sets, all the set operations, such as ∩ and ∪, are applicable to relations.

**Composing Relations**
Let $R_1$ and $R_2$ be relations on A. The composition $R_1 \circ R_2$ is the relation on A defined by
$(x, z) \in (R_1 \circ R_2)$ iff $\exists y\ (R_1(x, y) \wedge R_2(y, z))$
e.g. $R_1$ contains (0, 2), $R_2$ contains (2, 4) then $R_1 \circ R_2$ will contain (0,4) here 0 is x, 2 is y and 4 is z
The n-fold composition $R^n$ is defined by $R^1 = R$; $R^{n+1} = R^n \circ R$

**Properties of Relations**
Let A be a non-empty set and let R be a relation on A.
R is reflexive iff R(x, x) for all x in A.
R is irreflexive iff R(x, x) holds for no x in A.
R is symmetric iff $R(x, y) \Rightarrow R(y, x)$ for all x, y in A.

R is asymmetric iff $R(x, y) \Rightarrow \neg R(y, x)$ for all x, y in A.

R is antisymmetric iff $R(x, y) \wedge R(y, x) \Rightarrow x = y$ for all x , y in A.

R is transitive iff $R(x, y) \wedge R(y, z) \Rightarrow R(x, z)$ for all x, y, z in A.

**Equivalence Relations**
A binary relation which is reflexive, symmetric and transitive is an equivalence relation.
The identity relation $\Delta_A$ is the smallest equivalence relation on a set A. The full relation $A^2$ is the largest equivalence relation on A.

**Reflexive, Symmetric, Transitive Closures**

1 The full relation is transitive.

2 Transitive relations are closed under intersection, that is, if $R_1$ and $R_2$ are transitive then so is $R_1 \cap R_2$.

Together, these two properties tell us that for any binary relation R, there is a unique smallest transitive relation $R^+$ which includes R, we call $R^+$ the transitive closure of R.

Similarly we have the (unique) reflexive closure and the (unique) symmetric closure of R.

The transitive closure of R can be defined in terms of union and composition: $R^+ = \bigcup_{n \geq 1} R^n$

The reflexive, transitive closure is $R^* = \bigcup_{n \geq 0} R^n = R^+ \cup \Delta_A$

(x, y), (y, z)∈ R, and hence (x, y), (y, z) ∈ $R^+$, but since $R^+$ is transitive, (x, z) ∈ $R^+$ ($R^2$ gives us all such pairs)

(x, z) ∈ $R^2$, (z, w) ∈ R, and hence (x, z), (z, w) ∈ $R^+$, but since $R^+$ is transitive, (x, w) ∈ $R^+$ ($R^3$ gives us all such pairs)

The reflexive, transitive closure is $R^* = R^+ \cup \Delta_A$

What is the reflexive, transitive closure of R = {(n, n + 1) | n ∈ N}? <= relation

What is the symmetric closure of < on Z? /= relation

**Partial Orders**

R is a pre-order iff R is transitive and reflexive.

R is a strict partial order iff R is transitive and irreflexive.

R is a partial order iff R is an antisymmetric preorder.

R is linear iff R (x , y ) ∨ R (y , x ) ∨ x = y for all x , y in A.

A linear partial order is also called a total order.

In a total order, every two elements from A are comparable.

**Functions**

**Mathematically**: A function f is a relation with the property that (x, y) ∈ f ∧ (x, z) ∈ f ⇒ y = z. That is, for x ∈ dom(f), there is exactly one y ∈ ran(f) such that (x, y) ∈ f. We write this: f(x) = y.

**Computationally**: A prescription (an algorithm) for how to calculate output values from input values.

Note that a function-as-a-relation may be infinite, but we assume that an “algorithm” is finite.

**Domains and Co-Domains**

We say that the function f is from X to Y, or f : X → Y

if dom(f) = X and ran(f) ⊆ Y. We call Y the co-domain of f.

Example: The range of the factorial function is {1, 2, 6, 24, 120, . . .}, but we normally define it as having co-domain N.

The domain/co-domain specification is integral to the function definition, as we define functions f : X → Y and f′ : X′ → Y′ to be equal iff X = X′ , Y = Y′ , and for all x ∈ X , f(x) = f′(x).

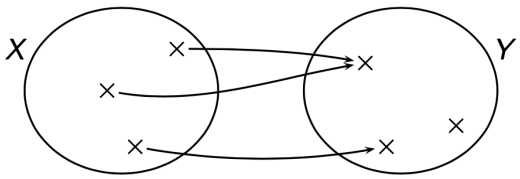
**Injections, Surjections and Bijections**

A function f : X → Y is

surjective (or onto) iff f [X] = Y. that is, every value in co-domain get hit by some value in domain

injective (or one-to-one) iff f (x) = f (y) ⇒ x = y.

bijective iff it is both surjective and injective.




**Examples**

f : Z → Z defined by f (n) = $n^2$ is neither surjective nor injective.

g : Z → N defined by g(n) = |n| is surjective but not injective.

s : N → N defined by s(n) = n + 1 is injective but not surjective.

d : Z → N defined by d(n)= 2n−1 if n>0; −2n if n ≤ 0 is bijective. It establishes a one-to-one mapping between Z and N.

**Function Composition**

The composition of f : X → Y and g : Y → Z is the function g ◦ f : X → Z defined by (g ◦ f)(x) = g(f(x))

We assume that g’s domain coincides with f ’s co-domain, although the composition makes sense as long as ran(f) ⊆ dom(g).

Note the unfortunate inconsistency with the use of ◦ for composing relations. For functions, g ◦ f is best read as “g after f .”

◦ is associative ,and for f :X → Y, f ◦ $1_X$ = $1_Y$ ◦ f = f, where $1_X$ : X → X is the identity function on X.

**Alphabets**
- Σ is a finite alphabet, the elements of Σ are the symbols of the alphabet
- e.g. {0, 1}, {a, b, c}, {a, b, c, …, x, y, z}

**Strings**
A string over Σ is a finite sequence of symbols contended together from Σ. e.g. '0100' 'abbbab' 'duck'
We write the concatenation of a string y to a string x as xy.
The empty string is denoted by ε. Think ""

**Kleene star**
$\Sigma^*$ denotes the set of all finite strings over Σ.

$$\Sigma^* := \bigcup_{n \in \mathbb{N}} \{a_1 a_2 \dots a_n \mid \forall 1 \le i \le n \ (a_i \in \Sigma)\}$$

Union over all $n \in \mathbb{N}$

e.g. {0,1}* = {ε, 0, 1, 00, 01, 10, 11, …, 10010101, …}

the set is infinite, but not any of its strings

**Languages**
A language (over alphabet Σ) is a (finite or infinite) set of finite strings over Σ.
It's a subset of $\Sigma^*$
e.g. ∅, $\Sigma^*$, {ε}, {010}, ab*a, {ww | w ∈ $\Sigma^*$}, {F | F is a valid Haskell expression}
{<p> |p is a traveling salesman problem whose solution length is less than 100}
{F | F is a valid Haskell expression which executes in finite time}

**Class of languages**
we will distinguish between primitive and sophisticated languages by dividing them into classes:
Regular context free context sensitive Turing recognizable Decidable

**Closure**
Given any operation on a language a class of language is closed under <that operation>
If, applying it to language is in that class can never produce a language outside that class

**DFA (deterministic finite-state automaton) formal definition**
A finite automaton is a 5-tuple (Q, Σ, δ, $q_0$, F), where
- Q is a finite set of states
- Σ is a finite alphabet
- δ : Q × Σ → Q is the transition function
- $q_0$ ∈ Q is the start state,
- F ⊆ Q are the accept states.

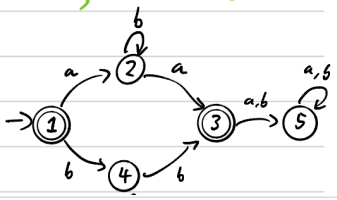
Here δ is a total function, that is, δ must be defined for all possible inputs.

**DFA informally,**
1. Begin in the start state
2. Read the next symbol in the string, and follow the corresponding transition to the next state
3. Repeat 2 until no symbols remain
4. If current state is an accept state, accept the string and consider to be valid

Example $(Q, \Sigma, \delta, q_0, F)$ $\delta((1,a)) = 2$

$Q = \{1,2,3,4,5\}$, $\Sigma = \{a,b\}$, $q_0 = 1$, $F = \{1,3\}$

$$\delta = \begin{cases} ((1,a),2), ((1,b),4), \\ ((2,a),3), ((2,b),2), \\ ((3,a),5), ((3,b),5), \\ ((4,a),4), ((4,b),3), \end{cases}$$



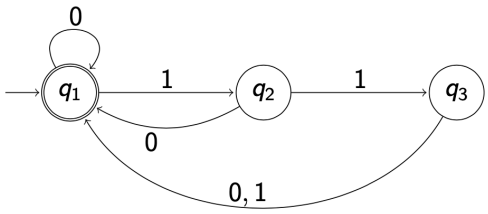

The example DFA above accepts aba because the sequence of states 1, 2, 2, 3 satisfies $1 = q_0$, $3 \in F$ and $2 = \delta(1,a)$ $2 = \delta(2,b)$ $3 = \delta(2,a)$. We could package this information like this: $q_0 = 1 \xrightarrow{a} 2 \xrightarrow{b} 2 \xrightarrow{a} 3 \in F$

**Rejection (for DFAs)**
If the final state in this sequence is not in F we can say that the DFA rejects that string
**Acceptance and Recognition**
We will define several models of computation, which are able to accept and sometimes reject, strings over some alphabet. We say that such a machine recognizes a language g if it accepts **all** strings in the language, and **does not** accept others.
In exam, a DFA will accept w as an empty string if there is nothing to check. E.g. ε belongs to {w | every odd position of w is a 1}.
In exam, 0 should be considered as multiple of all number, e.g. ε belongs to {w | the length of w is a multiple of 3}

Let M = (Q, Σ, δ, $q_0$, F) and let w = $v_1v_2 \cdots v_n$ be a string from $\Sigma^*$.
M accepts w iff there is a sequence of states $r_0, r_1, \ldots, r_n$, with each $r_i \in Q$, such that
1. $r_0 = q_0$
2. $\delta(r_i, v_{i+1}) = r_{i+1}$ for i = 0, …, n − 1
3. $r_n \in F$

M recognises language A iff A = {w | M accepts w}.

**Example**




The automaton $M_1$ (above) can be described precisely as

$$M_1 = (\{q_1, q_2, q_3\}, \{0,1\}, \delta, q_1, \{q_1\}) \quad \text{with}$$

| $\delta$ | 0 | 1 |
|---|---|---|
| $q_1$ | $q_1$ | $q_2$ |
| $q_2$ | $q_1$ | $q_3$ |
| $q_3$ | $q_1$ | $q_1$ |

$$L(M_1) = \left\{ w \;\middle|\; \begin{array}{l} w \text{ is } \epsilon, \textit{ or} \text{ ends with '0', } \textit{or} \text{ the number of} \\ \text{'1' symbols ending } w \text{ is a multiple of 3} \end{array} \right\}$$

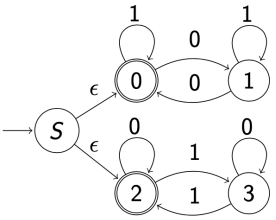
is the language recognised by $M_1$.

**Regular Languages**
A language is regular iff there is a finite automaton (DFA) that recognises it.
Examples: ∅, $\Sigma^*$, {ε}, {010}, {a}* U {b} + any finite language

**Regular Operations**
A language is simply a set of strings. Let A and B be languages. The regular operations are:
Union: A ∪ B
Concatenation: A ◦ B ={xy |x ∈ A, y ∈ B}
Kleene star: A* ={$x_1x_2\cdots x_k$ | k ≥ 0, each $x_i \in$A}
Note that the empty string, ε, is always in A*.
The regular languages are **closed** under the regular operations. Which means if A, B are regular languages, then A ∪ B, A ◦ B, A* are regular languages
Regular languages have several other closure properties. They are closed under
- intersection,
- complement, $A^c$
- difference (this follows, as A \ B = A ∩ $B^c$ )
- reversal.

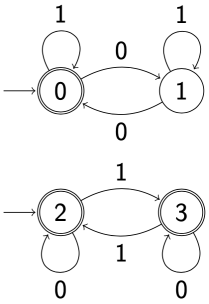
**Example**
Let A = {aa, abba} and B = {a, ba, bba, bbba, . . .}.
A ∪ B = {a, aa, abba, ba, bba, bbba, . . .}.
A ◦ B = {aaa, abbaa, aaba, abbaba, aabba, abbabba, . . .}.
A* = {ε, aa, abba, aaaa, aaabba, abbaaa, abbaabba, aaaaaa, aaaaabba, aaabbaaa, aaabbaabba, …}.

**Nondeterminism finite automata (NFA)**
**Nondeterminism** being able to make a correct sequence of choices to achieve the goal
In a non-deterministic finite-state automata, the non-determinism appears via epsilon transitions and multiple choice transitions.

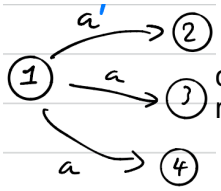



**Epsilon Transitions**
ε transition: NFAs be allowed to move from one state to another without consuming input.
Among other things, this gives us an easy way to construct a machine to recognise the union of two languages:

**Multiple Possible Start States**
Epsilon transitions are often useful, but in the previous example we actually did not need them, because an NFA is also allowed to have multiple start states.
This NFA is equivalent to the previous one, but it has only four states:

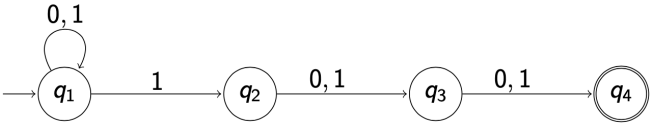



**Multiple choice transitions**

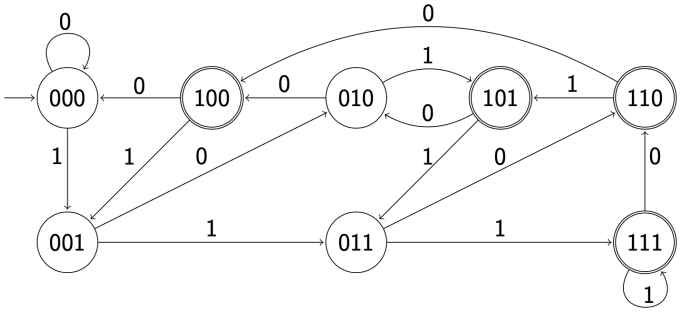



can read "a" and transition to either 2, 3, or 4.
represented δ(1, a) = {2, 3, 4}

**Example:** Here is an NFA that recognises the language $\left\{ w \;\middle|\; \begin{array}{l} w \in \{0,1\}^* \text{ has length 3 or more,} \\ \text{and the third last symbol in } w \text{ is 1} \end{array} \right\}$




Note: No transitions from $q_4$, and two possible transitions when we meet a 1 in state $q_1$.
The NFA is more intelligible than a DFA for the same language:




**Formal Definition of NFA**
For any alphabet Σ let $\Sigma_\varepsilon$ denote Σ ∪ {ε}.
An NFA is a 5-tuple (Q, Σ, δ, I, F), where
- Q is a finite set of states,
- Σ is a finite alphabet,
- δ : Q × $\Sigma_\varepsilon$ → P(Q) is the transition function,
- I ⊆ Q are the start states, and
- F ⊆ Q are the accept states.

Note that, unlike a DFA, an NFA can have several "start" states—it can start executing from any one of those.

**Difference between DFAs & NFAs**:
DFA δ : Q × Σ → Q I.e. (q, a) —> q'
NFA δ : Q × $\Sigma_\varepsilon$ → P(Q) I.e. (q, a) —> <some subset of Q>
⬆️ a either a ∈ Σ or a = ε x

**NFA Acceptance and Recognition, Formally**

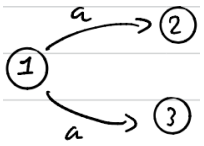
The definition of what it means for an NFA N to accept a string says that it has to be possible to make the necessary transitions.

Let $N = (Q, \Sigma, \delta, I, F)$ be an NFA and let $w = v_1 v_2 \cdots v_n$ where each $v_i$ is a member of $\Sigma_\varepsilon$.

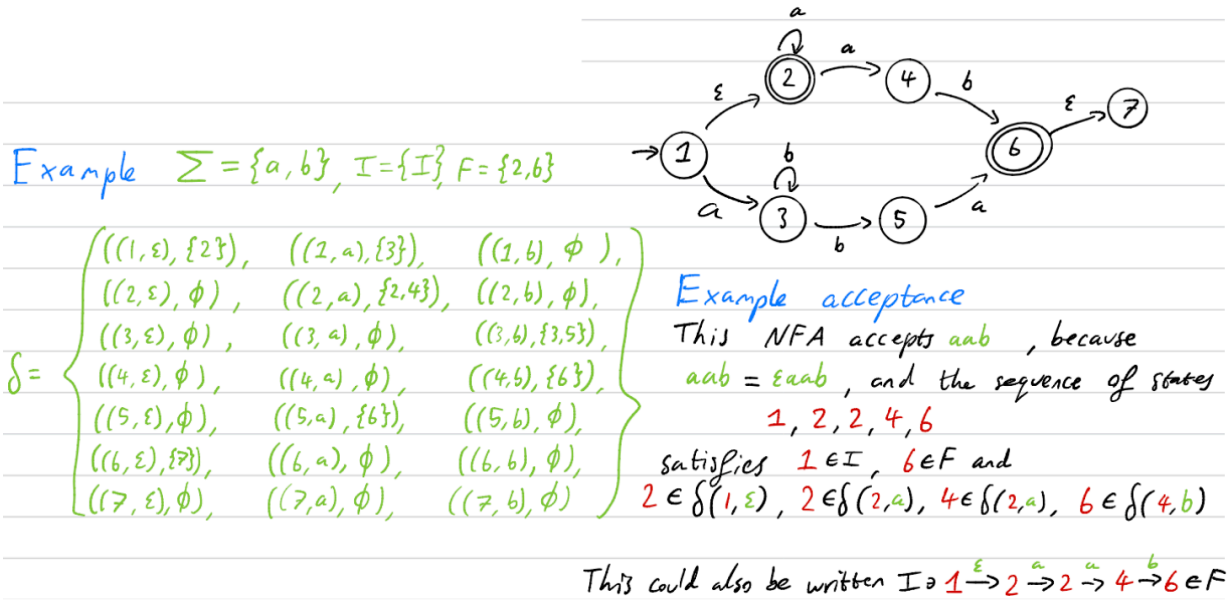
N accepts w iff there is a sequence of states $r_0, r_1, \ldots, r_n$, with each $r_i \in Q$, such that

1. $r_0 \in I$
2. $r_{i+1} \in \delta(r_i, v_{i+1})$ for $i = 0, \ldots, n-1$
3. $r_n \in F$

N recognises language A iff $A = \{w \mid N \text{ accepts } w\}$.




Note in this situation, we cannot say $2 = \delta(1, a)$,
we must say $2 \in \delta(1, a)$.
It's also wrong to say $\{2\} = \delta(1, a)$




**DFAs vs NFAs**

The class of languages recognised by NFAs is exactly the class of regular languages.

Theorem: Every NFA has an equivalent DFA.

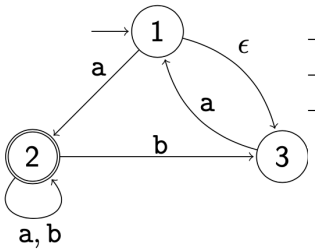
The proof rests on the so-called subset construction.

Given NFA N, we construct DFA M, each of whose states corresponds to a set of N-states.

If N has k states then M may have up to $2^k$ states (but it will often have far fewer than that).

Consider the NFA on the right. We can systematically construct an equivalent DFA from the NFA.

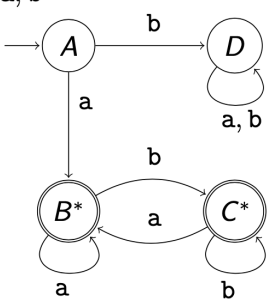
The DFA's start state is {1, 3}. From {1, 3} a takes us to {1, 2, 3}. From {1, 2, 3}, a takes us back to {1, 2, 3}, b takes us to {2, 3}.




| | a | b |
|---|---|---|
| A={1,3} | B* | – |
| B*={1,2,3} | B* | C* |
| C*={2,3} | B* | C* |

Any state S which contains an accept state from the NFA will be an accept state for the DFA. Here we mark accept states with a star.

Here is the equivalent DFA that we derive.

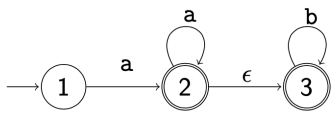


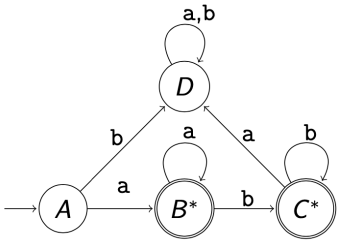

| | a | b |
|---|---|---|
| A={1,3} | B* | D |
| B*={1,2,3} | B* | C* |
| C*={2,3} | B* | C* |
| D=∅ | D | D |

Any state S which contains an accept state from the NFA (in this case the NFA has just one, namely state 2) becomes an accept state for the DFA.

We add (dead) state D that corresponds to the empty set.

## One more example




Adding new state to DFA:

1. Step 1: Move on a symbol
2. Step 2: Build $\epsilon$-closure

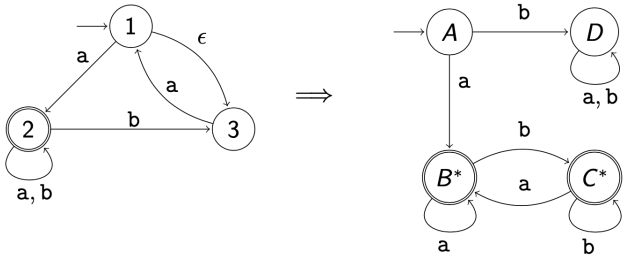
| | a | b |
|---|---|---|
| A={1} | B* | D |
| B*={2,3} | B* | C* |
| C*={3} | D | C* |
| D=∅ | D | D |

## Equivalence of DFAs

We can always find a minimal DFA for a given regular language
(by minimal we mean having the smallest possible number of states).
Since a DFA has a unique start state and the transition function is total and deterministic, we can test two DFAs for equivalence (modulo the names used for their states) by minimizing them.

## Minimizing DFAs

There is no guarantee that DFAs that are produced by the various algorithms, such as the subset construction method, will be minimal.
A={1, 3}, B* = {1,2,3}, C* = {2, 3}, and D = ∅.

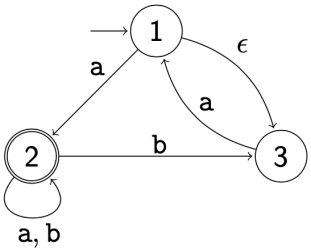


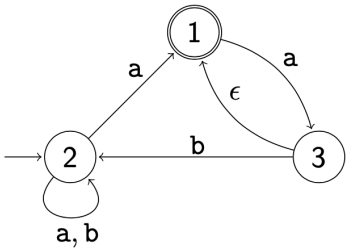

## Generating a Minimal DFA

The following algorithm takes an NFA and produces an equivalent minimal DFA. Of course the input can also be a DFA.
1 Reverse the NFA;
2 Determinize the result;
3 Reverse again;
4 Determinize.
To reverse an NFA A with start states I and accept states F , F /= ∅: simply reverse every transition in A and swap I and F.

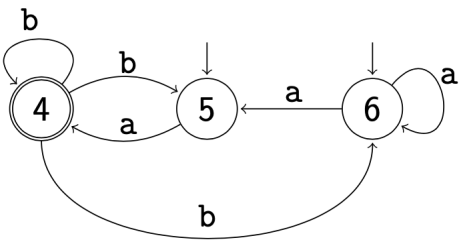
## Example:

Here it is on the left, with its reversal on the right:

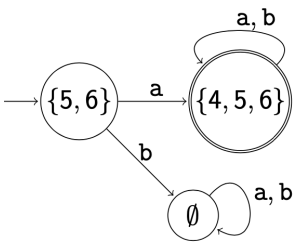


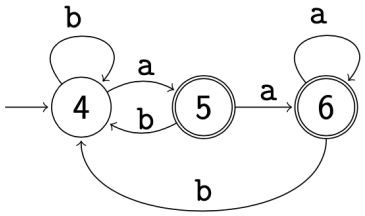




Now make the reversed NFA deterministic (we have renamed the states to avoid later confusion: 4 corresponds to {2}, 5 to {1, 2}, and 6 to {1, 2, 3}).
Now reverse the result:




Finally make the result deterministic:



**Regular Expressions**
Regular expressions is a notation for languages.
Example: (0 ∪ 1)(0 ∪ 1)(0 ∪ 1)((0 ∪ 1)(0 ∪ 1)(0 ∪ 1))* denotes the set of non-empty strings with the lengths that are multiple of 3.
The star binds tighter than concatenation, which in turn binds tighter than union.

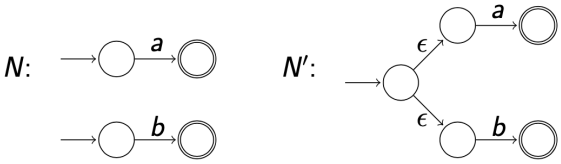
**Syntax:**
The regular expressions over an alphabet $\Sigma = \{a_1, \ldots, a_n\}$ are given by the grammar
regexp → $a_1$ | ⋯ | $a_n$ | ε | ∅ | regexp ∪ regexp | regexp regexp | regexp*
**Semantics:**
L(a) = {a}
L(ε) = {ε}
L(∅) =∅
$L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$
$L(R_1\, R_2) = L(R_1) \circ L(R_2)$
L(R*) = L(R)*

**Regular Expressions – Examples**
ε : {ε}  1 : {1}  110 : {110}
((0 ∪ 1)(0 ∪ 1))* : all binary strings of even length
(0 ∪ ε)(ε ∪ 1) : {ε, 0, 1, 01}
1* : all finite sequences of 1s
ε ∪ 1 ∪ (ε ∪ 1) * (ε ∪ 1) : all finite sequences of 1s
(1*0*)* : 1 or 0 appear any number of times

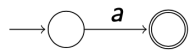
**Regular Expressions vs Automata**
Theorem: L is regular iff L can be described by a regular expression.
First note that, given NFA N = (Q, Σ, δ, I, F), we can build an equivalent NFA with at most one start state, like so: If | I | ≤ 1, do nothing. Otherwise transform N to N′ = (Q ∪ {$q_i$}, Σ, δ′, {$q_i$}, F) by adding a new state $q_i$ , with ε transitions from $q_i$ to each state in I :

$$\delta'(q, v) = \begin{cases} I & \text{if } q = q_i \text{ and } v = \epsilon \\ \delta(q, v) & \text{otherwise} \end{cases}$$

**Example:**




**NFAs from Regular Expressions**
We now show the 'if' direction of the theorem, by showing how to convert a regular expression R into an NFA that recognises L(R). The proof is by structural induction over the form of R.

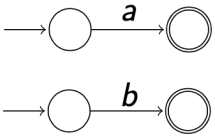
Case $R = a$: Construct 


Case $R = \epsilon$: Construct 

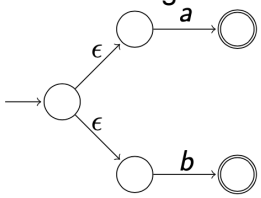
Case $R = \emptyset$: Construct 

Case R = $R_1$ ∪ $R_2$, R = $R_1$ $R_2$, or R = $R_1$*:

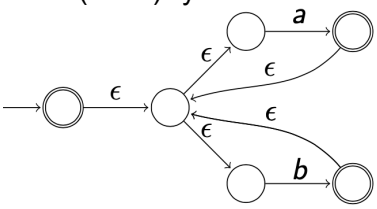
**NFAs from Regular Expressions: Example**
Let us construct, in the proposed systematic way, an NFA for (a ∪ b)*bc.
Start from innermost expressions and work out:



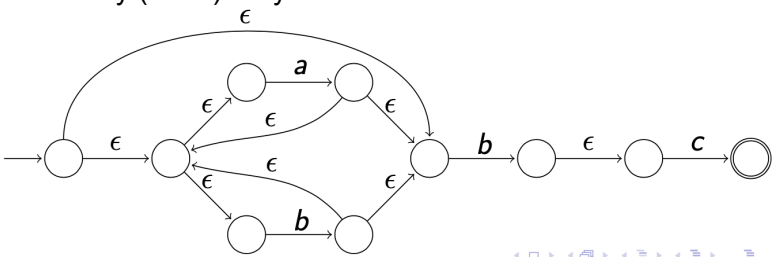

So a single-start state NFA for a ∪ b is:



Then (a ∪ b)* yields:




Finally (a ∪ b)*bc yields:




**Regular Expressions from NFAs**

We now show the 'only if' direction of the theorem.

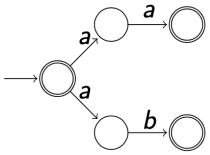
First note that, given an NFA N, we can build an equivalent NFA with at most one accept state.

We transform N = (Q, Σ, δ, I, F) to N′ =(Q ∪ {$q_f$}, Σ, δ′, I, {$q_f$}) like so: If |F| ≤ 1, do nothing.

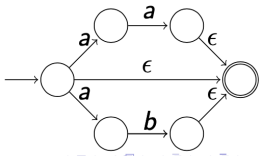
Otherwise add a new $q_f$ and ε transitions to $q_f$ from each state in F. $q_f$ becomes the only accept state:

$$\delta'(q,v) = \begin{cases} \delta(q,v) \cup \{q_f\} & \text{if } q \in F \text{ and } v = \epsilon \\ \delta(q,v) & \text{otherwise} \end{cases}$$

N:




N′:



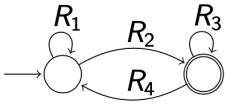

We sketch how an NFA can be turned into a regular expression in a systematic process of "state elimination".

In the process, arcs get labelled with regular expressions.

Start by making sure the NFA has a single accept state and a single start state.

Repeatedly eliminate states that are neither start nor accept states.

The process produces either

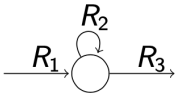



or


We get ($R_1 \cup R_2 R_3^* R_4)^* R_2 R_3^*$ in the first case; $R^*$ in the second.

**The State Elimination Process**

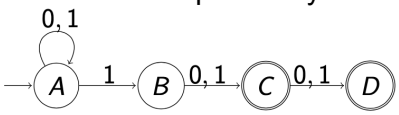
Consider a node



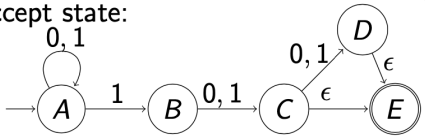

Any such pair of incoming/outgoing arcs get replaced by a single arc that bypasses the node. The new arc gets the label $R_1R_2^*R_3$.

If there are m incoming and n outgoing arcs, these arcs are replaced by m × n bypassing arcs when the node is removed.

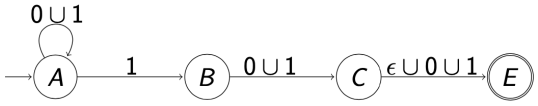
Let us illustrate the process on this example:



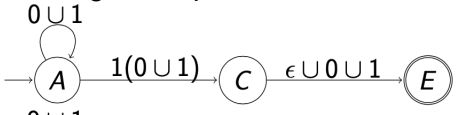

Create a single accept state:



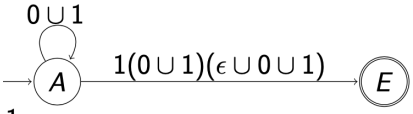

Eliminate D (and use regular expressions with all arcs):



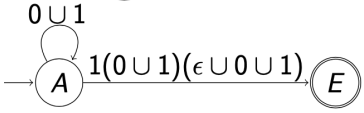

Now eliminate B:



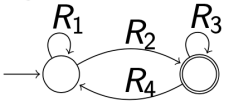

and then C:




Note that




is




with $R_1 = 0 \cup 1$, $R_2 = 1(0 \cup 1)(\varepsilon \cup 0 \cup 1)$ , $R_3 = R_4 = \varnothing$

Hence the instance of the general "recipe" $(R_1 \cup R_2R_3^*R_4)^*R_2R_3^*$ is $(0 \cup 1)^*1(0 \cup 1)(\varepsilon \cup 0 \cup 1)$

**Some Useful Laws for Regular Expressions**

$A \cup A = A$

$A \cup B = B \cup A$

$(A \cup B) \cup C = A \cup (B \cup C) = A \cup B \cup C$

$(A\,B)\,C = A\,(B\,C) = A\,B\,C$

$\varnothing \cup A = A \cup \varnothing = A$

$\varepsilon\,A = A\,\varepsilon = A$

$\varnothing\,A = A\,\varnothing = \varnothing$

$(A \cup B)\,C = A\,C \cup B\,C$

$A\,(B \cup C\,) = A\,B \cup A\,C$

$(A^*)^* = A^*$

$\varnothing^* = \varepsilon^* = \varepsilon$

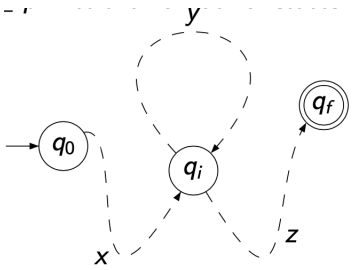
$(\varepsilon \cup A)^* = A^*$

$(A \cup B)^* = (A^*\,B^*)^*$

**The Pumping Lemma for Regular Languages**
This is the standard tool for proving languages non-regular.
Loosely, it says that if we have a regular language A and consider a sufficiently long string s ∈ A, then a recogniser for A must traverse some loop to accept s. So A must contain infinitely many strings exhibiting repetition of some substring in s.
**Pumping Lemma**: If A is regular then there is a number p such that for any string s ∈ A with |s| ≥ p (|s| means the length of s), s can be written as s = xyz, satisfying
$xy^iz \in A$ for all i ≥ 0 ($xy^iz$ means x(y repeated i times)z); y /= ε; |xy| ≤ p

**Proving the Pumping Lemma**
Let DFA M = (Q, Σ , δ , $q_0$, F) recognise A.
Let p = |Q| and consider s with |s| ≥ p. Let the number of states of M be p, let |s| ≥ p.

In an accepting run for s, some state must be re-visited.Let $q_i$ be the first such state. $q_0$ At the first visit, x has been consumed, at the second, xy, (strictly longer than x).




Consider the first time a state ($q_i$) is re-visited. This suggests a way of splitting s into x, y and z such that xz, xyz, xyyz,... are all in A. Notice that y/= ε. Let m + 1 be the number of state visits when reading xy, then |xy| = m ≤ p, because m + 1 is the number of state visits with only one repetition.

**Using the Pumping Lemma**
The pumping lemma says:

$$A \text{ regular} \Rightarrow \exists p \forall s \in A : \left\{ \begin{array}{l} s \text{ can be written} \\ xyz \text{ such that } \ldots \end{array} \right.$$

We can use its contrapositive to show that a language is non-regular:

$$\forall p \exists s \in A : \left\{ \begin{array}{l} s \text{ can't be written} \\ xyz \text{ such that } \ldots \end{array} \right\} \Rightarrow A \text{ not regular}$$

Coming up with such an s is sometimes easy, sometimes difficult.

**Pumping Example 1**
We show that B = $\{0^n1^n \mid n \geq 0\}$ is not regular.
Assume it is, and let p be the pumping length.
Consider $0^p1^p \in B$ with length greater than p.
By the pumping lemma, $0^p1^p = xyz$, with $xy^iz$ in B for all i ≥ 0, y /= ε, and |xy| ≤ p.
Since |xy| ≤ p, y consists entirely of 0s. But then xyyz /∈ B, a contradiction.
So we inevitably arrive at a contradiction if we assume that B is regular.

**Pumping Example 2**
C = {w | w has an equal number of 0s and 1s} is not regular.
A simple proof: If C were regular then also B from before would be regular, since B = C ∩ 0*1* and regular languages are closed under intersection.

**Pumping Example 3**
We show that D = {ww | w ∈ {0,1}*} is not regular.
Assume it is, and let p be the pumping length.
Consider $0^p10^p1 \in D$ with length greater than p.
By the pumping lemma, $0^p10^p1 = xyz$, with $xy^iz$ in D for all i ≥ 0, y/= ε, and |xy| ≤ p.
Since |xy| ≤ p, y consists entirely of 0s. But then xyyz/∈ D, a contradiction.

**Example 4 – Pumping Down**
We show that E = $\{0^i1^j \mid i > j\}$ is not regular.
Assume it is, and let p be the pumping length.
Consider $0^{p+1}1^p \in E$.
By the pumping lemma, $0^{p+1}1^p = xyz$, with $xy^iz$ in E for all i ≥ 0, y /= ε, and |xy| ≤ p.
Since |xy| ≤ p, y consists entirely of 0s. But then xz/∈ E , a contradiction.

**Context-Free Grammars**
A grammar is a set of substitution rules, or productions. We have the shorthand notation
R → 0 | 1 | eps | empty | R ∪ R | R R | R*

**Derivations, Sentences and Sentential Forms**
A simpler example is this grammar G:
A → 0 A 0;  A → 1 A 1;  A → ε
Using the two rules as a rewrite system, we get derivations such as
A ⇒ 0A0 ⇒ 00A00 ⇒ 001A100 ⇒ 00100100

A is called a variable. Other symbols (here 0 and 1) are terminals. We refer to a valid string of terminals (such as 00100100) as a sentence. The intermediate strings that mix variables and terminals are sentential forms.

**Context-Free Languages**
The language of G is written L(G). L(G) = {$ww^R$ |w ∈ {0, 1}*}
A language which can be generated by some context-free grammar is a context-free language (CFL).
It should be clear that some of the languages that we found not to be regular are context-free, for example {$0^n1^n$ |n≥1}

**Context-Free Grammars Formally**
A context-free grammar (CFG) G is a 4-tuple (V, Σ, R, S), where
- V is a finite set of variables,
- Σ is a finite set of terminals,
- R is a finite set of rules, each consisting of a variable (the left-hand side) and a string in (V ∪ Σ)* (the right-hand side),
- S is the start variable.

The binary relation ⇒ on (V ∪ Σ)* is defined as follows.

Let u, v, w ∈ (V ∪ Σ)*. Then uAw ⇒ uvw iff A → v is a rule in R. That is, ⇒ captures a single derivation step.

Let $\overset{*}{\Rightarrow}$ be the reflexive transitive closure of ⇒. L(G) = {s ∈ Σ* | $\overset{*}{\Rightarrow}$S s}

**A Context-Free Grammar for Numeric Expressions**
Here is a grammar with three variables, 14 terminals, and 15 rules:
E → T | T + E (this means from E you can create T or create T + E)
T → F | F * T
F → 0 | 1 | … | 9 | (E)
When the start variable is unspecified, it is assumed to be the variable of the first rule.
An example sentence in the language is (3 + 7) * 2
The grammar ensures that * binds tighter than +.

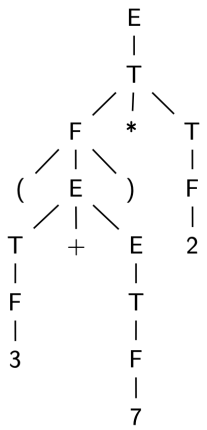
**Parse Trees**
There are different derivations leading to the sentence (3 + 7) * 2, all corresponding to the parse tree above. They differ in the order in which we choose to replace variables. Here is the leftmost derivation:

E ⇒ T ⇒ F * T ⇒ (E) * T ⇒ (T + E) * T ⇒ (F + E) * T  //E → T | T + E

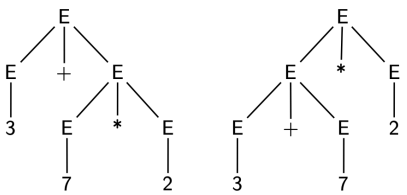
⇒ (3 + E) * T  //T → F | F * T

⇒ (3 + T) * T  //F → 0 | 1 | … | 9 | (E) ⇒ (3 + F) * T

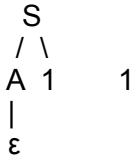
⇒ (3 + 7) * T ⇒ (3 + 7) * F ⇒ (3 + 7) * 2




**Accidental vs Inherent Ambiguity**
A grammar that has different parse trees for some sentence is ambiguous.
Sometimes we can find a better grammar (as in our example) which is not ambiguous, and which generates the same language.



**Example**: prove the language A = {$0^i1^j$ | 0 ≤ i < j} is context-free & not-regular
Proof: Let G = ({S, A}, {0, 1}, R, S) where R is
S —> A1
A —> A1 | 0A1 | ε




We claim that L(G) = A so A is context-free
Let p ≥ 1. Take $0^p1^{p+1} \in A$, then $|0^p1^{p+1}| = 2p + 1 \geq p$
Let x, y, z ∈ Σ*, be such that s = xyz, y /= ε, |xy| ≤ p
Since |xy| ≤ p, xy consists entirely of 0's, so $y = 0^m$, 1 ≤ m ≤ p
The xyyz has p + m many zeros and p + 1 ones, but p + m ≥ p + 1, since m ≥ 1, so xyyz /∈ A
Therefore by pumping lemma, A is not regular

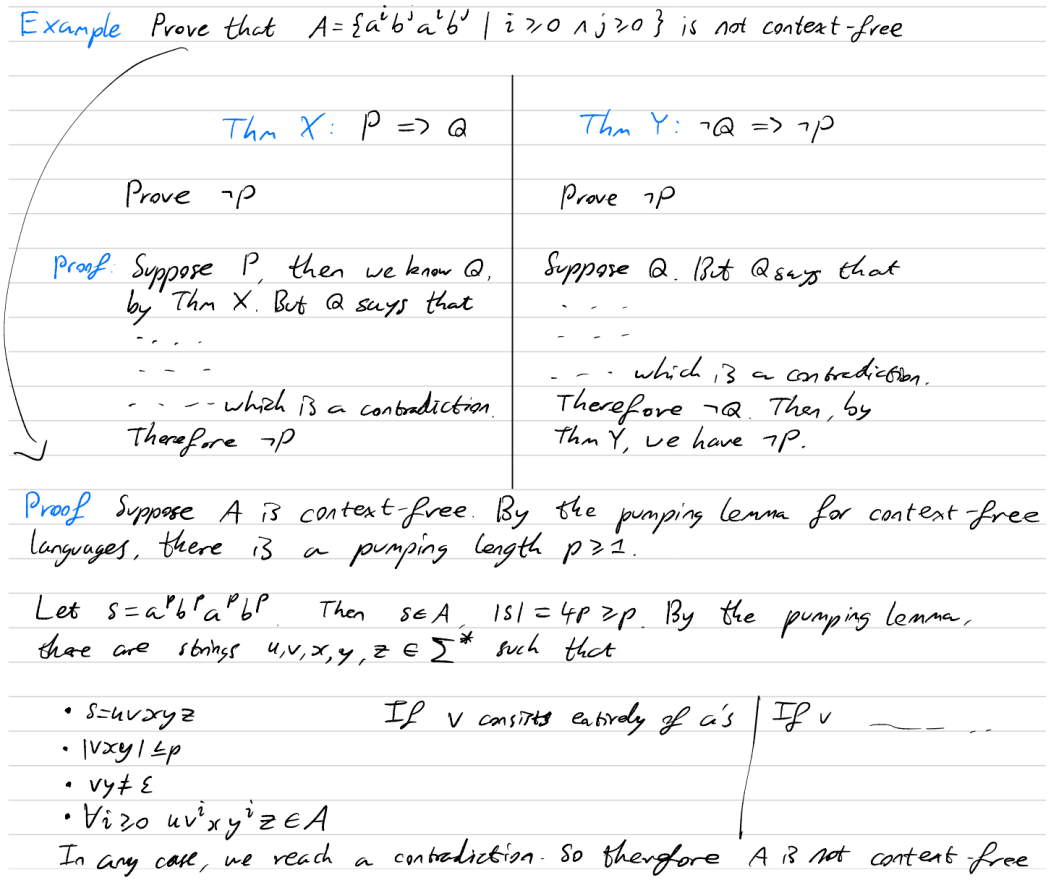
**Pumping Lemma for CFLs**
There are languages that are not context-free, and again there is a pumping lemma that can be used to show (some) languages non-context-free:
If A is context-free then there is a number p such that for any string s ∈ A with |s| ≥ p, s can be written as s = uvxyz, satisfying
- $uv^ixy^iz \in A$ for all i ≥ 0
- |vy| > 0
- |vxy| ≤ p

The Pumping Lemma for CFLs For any language $A$,
$A$ is Context-Free $\Rightarrow \exists p \geq 1\ \forall s \in A \left[ |s| \geq p \Rightarrow \exists u,v,x,y,z \left\{ \begin{array}{l} s = uvxyz \wedge |vxy| \leq p \wedge \ldots \\ vy \neq \varepsilon \wedge \forall i \geq 0\ uv^ixy^iz \in A \end{array} \right\} \right]$

**Step 1: Take any p**
**Step 2: design an s at least as long as p**
**Step 3: show <u>any</u> way of breaking down s into uvxyz cannot make all four statements true**

**Pumping Example 1**
A = {ww | w ∈ {0, 1}*} is not context-free.
Assume it is, let p be the pumping length, take $0^p1^p0^p1^p$.
By the pumping lemma, $0^p1^p0^p1^p$ = uvxyz, with $uv^ixy^iz$ in A for all i ≥ 0, and |vxy| ≤ p.
There are three ways that vxy can be part of 00. . .0011. . .1100. . .0011. . .11
If it straddles the midpoint, it has form $1^n0^m$, so pumping down, we are left with $0^p1^i0^j1^p$, with i < p, or j < p, or both.
If it is in the first half, $uv^2xy^2z$ will have pushed a 1 into the first position of the second half.
Similarly if vxy is in the second half.

**Pumping Example 2**
B = {$a^n b^n c^n$ | n ∈ N} is not context-free.
Assume it is, let p be the pumping length, and take $a^p b^p c^p \in B$.
By the pumping lemma, $a^p b^p c^p$ = uvxyz, with $uv^i xy^i z$ in B for all i.
Either v or y is non-empty (or both are).
If one of them contains two different symbols from {a, b, c} then $uv^2xy^2z$ has symbols in the wrong order, and so cannot be in B.
So both v and y must contain only one kind of symbol. But then $uv^2xy^2z$ can't have the same number of as, bs, and cs.
In all cases we have a contradiction.



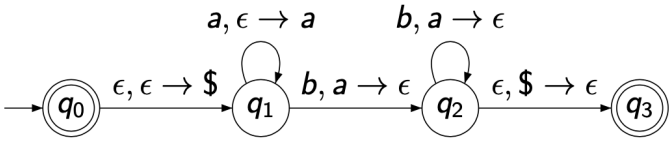

**Closure Properties for CFLs**
The class of context-free languages is closed under union, concatenation, Kleene star, reversal.
The class of context-free languages is not closed under intersection!
Hence it is not closed under complement either
Consider these two CFLs:
C = {$a^m b^n c^n$ | m, n ∈ N}
D = {$a^n b^n c^m$|m, n ∈ N}
But C∩D is the language B = {$a^n b^n c^n$ | n ∈ N} which we just showed is not context-free.
However, we do have: If A is context-free and R is regular then A ∩ R is context-free.

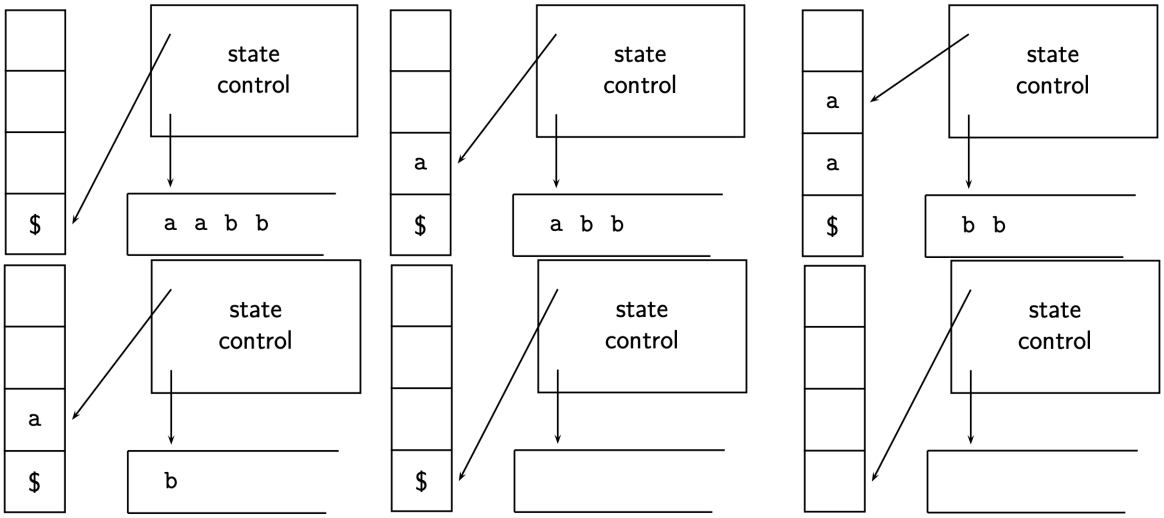
**Pushdown Automata Formally**
A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where
- $Q$ is a finite set of states,
- $\Sigma$ is the finite input alphabet,
- $\Gamma$ is the finite stack alphabet,
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow P(Q \times \Gamma_\varepsilon)$ is the transition function,
- $q_0 \in Q$ is the start state, and
- $F \subseteq Q$ are the accept states.

$\delta(q_5, a, b) = \{(q_7, \varepsilon)\}$ means: If in state $q_5$, when reading input symbol a, provided the top of the stack holds 'b', consume the a, pop the b, and go to state $q_7$.
$\delta(q_5, \varepsilon, b) = \{(q_6, a), (q_7, b)\}$ means: If in state $q_5$, and if the top of the stack holds 'b', either replace that b by a and go to state $q_6$, or leave the stack as is and go to state $q_7$. In either case do not consume an input symbol.

**Epsilon transitions**: consider transaction a, y -> x
If a is ε, don't consume any input
If x is ε, don't take anything off the stack (jus add y)
If y is ε, don't replace x with anything (just take off x)
If x is ε & y is ε, don't change the stack

**PDA Example 1**



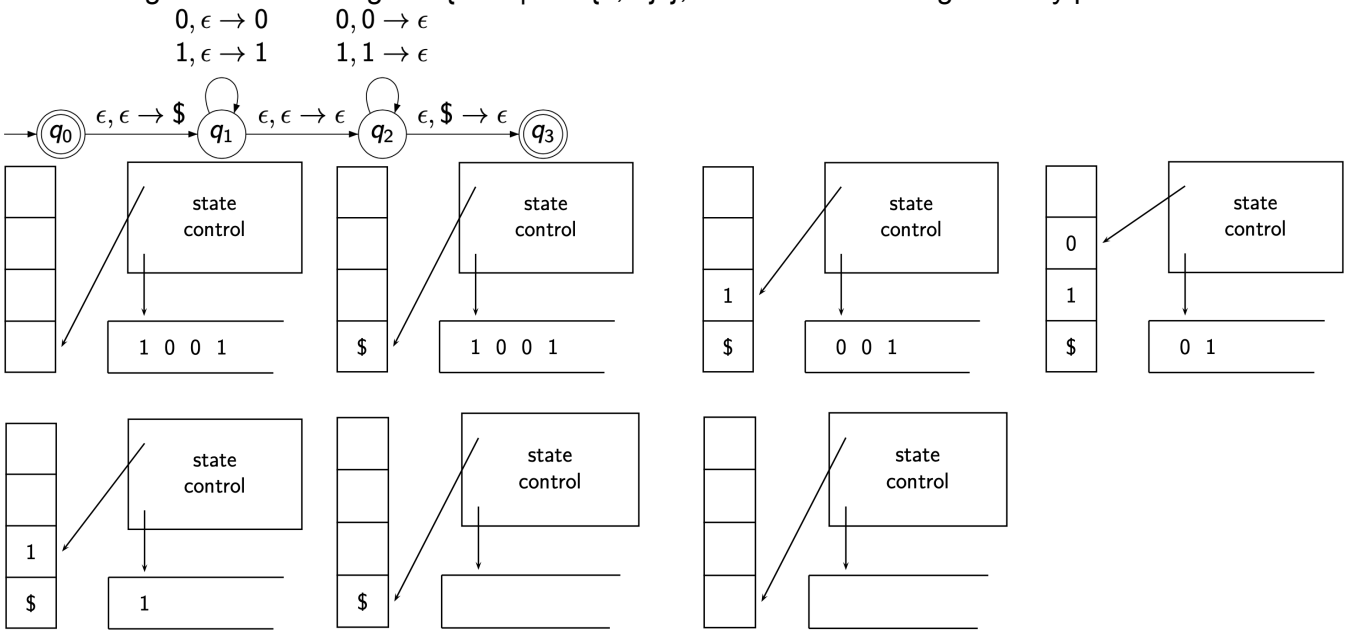

**This PDA recognises $\{a^n b^n \mid n \geq 0\}$:**
$Q = \{q_0, q_1, q_2, q_3\}$; $\Sigma = \{a, b\}$; $\Gamma = \{a, \$\}$;
$\delta(q_0, \varepsilon, \varepsilon) = \{(q_1, \$)\}$, $\delta(q_1, a, \varepsilon) = \{(q_1, a)\}$,
$\delta(q_1, b, a) = \{(q_2, \varepsilon)\}$, $\delta(q_2, b, a) = \{(q_2, \varepsilon)\}$,
$\delta(q_2, \varepsilon, \$) = \{(q_3, \varepsilon)\}$, for other inputs δ returns ∅;
$q_0 = q_0$; $F = \{q_0, q_3\}$.

**Acceptance Precisely**
The PDA $(Q, \Sigma, \Gamma, \delta, q_0, F)$ accepts input w iff $w = v_1 v_2 \cdots v_n$ with each $v_i \in \Sigma_\varepsilon$, and there are states $r_0, r_1, \ldots, r_n \in Q$ and strings $s_0, s_1, \ldots, s_n \in \Gamma^*$ such that
1 $r_0 = q_0$ and $s_0 = \varepsilon$.
2 $(r_{i+1}, b) \in \delta(r_i, v_{i+1}, a)$, $s_i = at$, $s_{i+1} = bt$ with $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*_\epsilon$.
3 $r_n \in F$.
Note 1: There is no requirement that $s_n = \varepsilon$, so the stack may be non-empty when the machine stops (even when it accepts).
Note 2: Trying to pop an empty stack leads to rejection of input, rather than "runtime error".
**Intuitively**
There is some state path you can take from $q_0$ to some accept state which
Consumes all of the input
Always has x on the top of the stack when a transition along the path want to replace x with something (no invalid transitions. e.g. a, x -> y if you don't have x on the top of stack you cannot transform it)

**PDA Example 2**

Let $w^R$ denote the string w reversed.

Let us design a PDA to recognise $\{ww^R \mid w \in \{0, 1\}^*\}$, the set of even-length binary palindromes:




**CFLs Have PDAs as Recognisers**

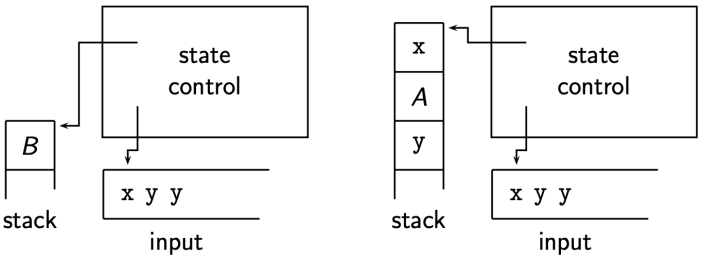
Given a context-free language L (in the form of a grammar), we can find a PDA which recognises L.

And, every PDA recognises a context-free language.

Namely, given a CFG G, we show how to construct a PDA P such that L(P) = L(G).

The idea is to let the PDA use its stack to store a list of "pending" recogniser tasks.

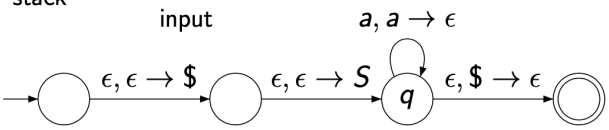
The construction does not give the cleverest PDA, but it always works.




Say B → xAy is a rule in G, and the PDA finds the symbol B on top of its stack, it may pop B and push y, A, and x, in that order.

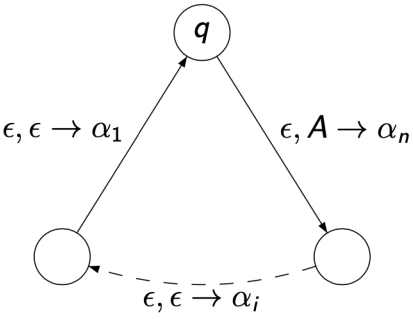
If it finds the terminal x on top of the stack, and x is the next input symbol, it may consume the input and pop x.

Construct the PDA like this:



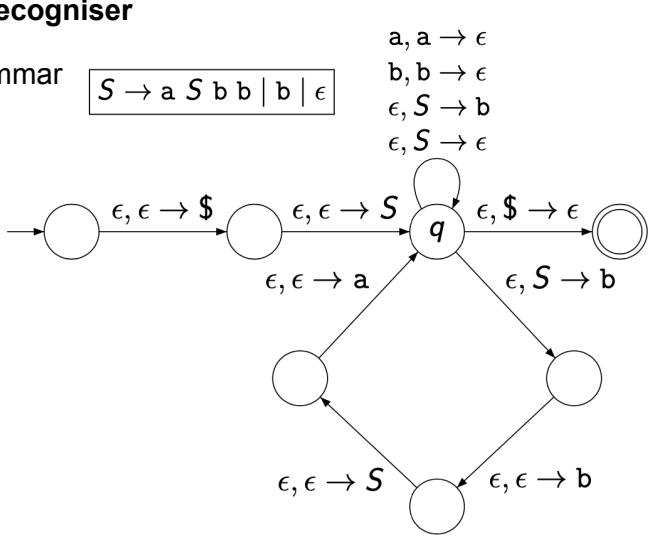

with a self-loop from q for each terminal a (S is the grammar's start symbol).

For each rule A → $\alpha_1 \ldots \alpha_n$, add this loop from q to q:



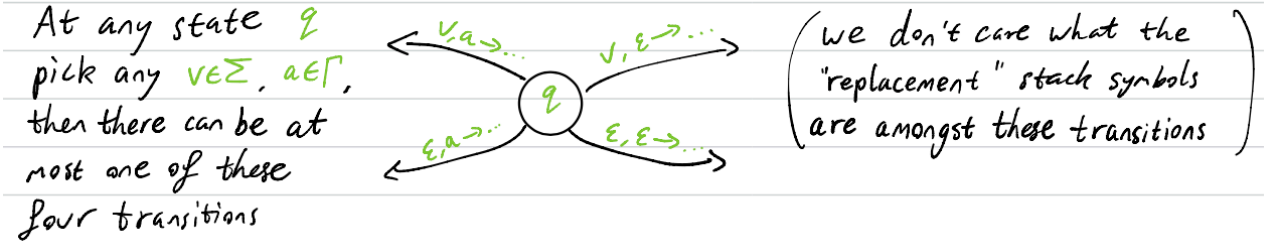

**Example Recogniser**

For the grammar $S \to \mathtt{a}\ S\ \mathtt{b}\ \mathtt{b} \mid \mathtt{b} \mid \epsilon$



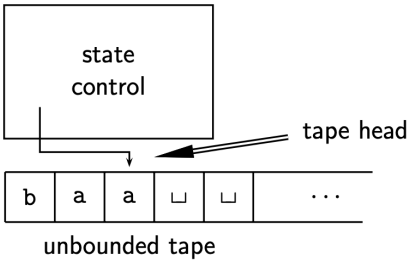
**Progressive PDAs**
A pushdown automaton $(Q, \Sigma, \Gamma, \delta, q_0, F)$ is progressive iff $\forall q \in Q, \forall a \in \Gamma_\varepsilon : \delta(q, \varepsilon, a) = \varnothing$.
A pushdown automaton is progressive if and only if each transition step consumes exactly one input symbol. i.e. no epsilon symbol

**Deterministic PDAs**
A pushdown automaton $(Q, \Sigma, \Gamma, \delta, q_0, F)$ is deterministic iff $\forall q \in Q, \forall v \in \Sigma, \forall a \in \Gamma$ it holds that:
$|\delta(q, v, a) + \delta(q, \varepsilon, a) + \delta(q, v, \varepsilon) + \delta(q, \varepsilon, \varepsilon)| \leq 1$.  // here $|x|$ is the size of set
For any configuration there can be at most one of the four transitions.
A deterministic pushdown automaton (DPDA) never finds itself in a position where it can make two different transition steps.



## Turing Machines

A Turing machine has an unbounded tape through which it takes its input and performs its computations.
Unlike our previous automata it can:
both read from and write to the tape, and
move either left or right over the tape.
The machine has distinct accept and reject states, in which it accepts/ rejects irrespective of where its tape head is.

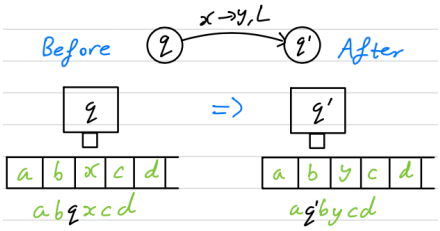


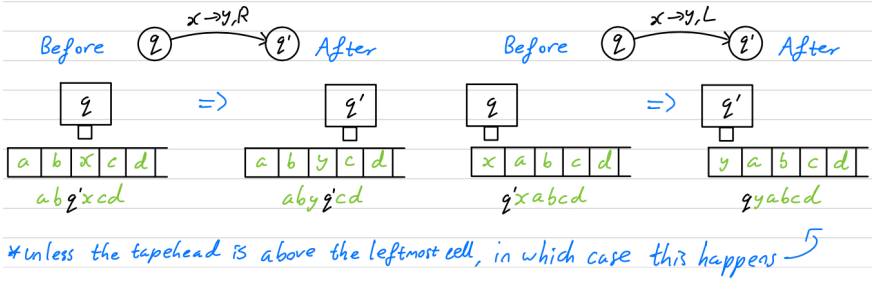

## Turing Machines Formally

A Turing machine is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ where
$Q$ is a finite set of states,
$\Gamma$ is a finite tape alphabet, which includes the blank character, _,
$\Sigma \subseteq \Gamma \setminus \{ \_ \}$ is the input alphabet, (which means no blanks in input)
$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,

- A transition $\delta(q_i, x) = (q_j, y, d)$ depends on two things
  - current state $q_i$, and
  - current symbol x under the tape head
- It consists of three actions
  - change state to $q_j$ ,
  - over-write tape symbol x by y, and
  - move the tape head in direction d.

$q_0 \in Q$ is the initial state
$q_a \in Q$ is the accept state, and
$q_r \in Q$ is the reject state. ($q_r \neq q_a$)

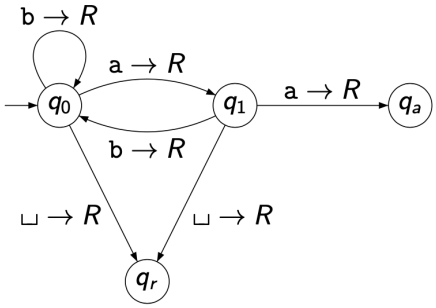



## Drawing Turing Machines

On an arrow from $q_i$ to $q_j$ we write
$x \to d$ when $\delta(q_i, x) = (q_j, x, d)$, and
$x \to y, d$ when $\delta(q_i, x) = (q_j, y, d)$, $y \neq x$.

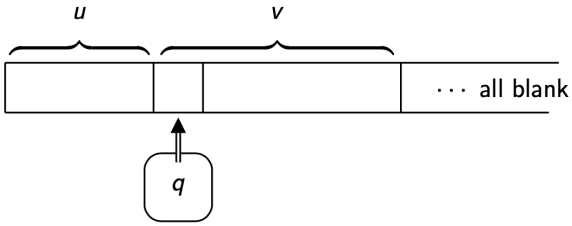
## Turing Machine Example 1




This machine recognises the regular language (a ∪ b)*aa(a ∪ b)*.
We can leave the reject state $q_r$ out with the understanding that transitions that are not specified go to $q_r$.

## Turing Machine Configurations

A configuration involves:
Which state the Turing machine is in
What is written on the tap
Which tape-cell the tapered is looking at
Notation: if u, v ∈ Γ*, x ∈ Γ, q ∈ Q, uqxv denotes the configuration where the state is q, the tape is ixv followed by infinitely many blanks and the tapered is over the first cell with an x after u
We write uqv for this configuration: uv
On input aba, the example machine goes through 5 configurations:




$\varepsilon q_0 aba$ (or just $q_0 aba$)
$\Rightarrow a q_1 ba$
$\Rightarrow ab q_0 a$
$\Rightarrow aba q_1 \_$
$\Rightarrow aba \_ q_r$

**Computations Formally**

For all $q_i, q_j \in Q$, a, b, c $\in \Gamma$, and u, v $\in \Gamma^*$, we have

$uq_ibv \Rightarrow ucq_jv$ if $\delta(q_i, b) = (q_j, c, R)$

$q_ibv \Rightarrow q_jcv$ if $\delta(q_i, b) = (q_j, c, L)$

$uaq_ibv \Rightarrow uq_jacv$ if $\delta(q_i, b) = (q_j, c, L)$

**Turing recognizability and decidability**

Let M be a Turing machine

M accepts a string w if, starting in configuration $q_0w$, it reaches the accept state after finitely many transitions

M rejects a string w if, starting in configuration $q_0w$ , it reaches the reject state after finitely many transitions.

A TM halts on input w if one of the above occurs. Otherwise it does not halt.

The language of M is L(M)= {w $\in \Sigma^*$|M accepts w}. Note that M does not accept w if M does not halt input w.

M is a decider if M halts on input w , for every w $\in \Sigma^*$

M decides A if L(M) = A and M is a decider

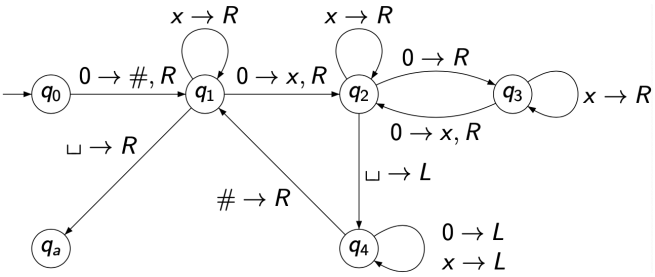
A language, A, is Turing recognisable if there is a TM, N, such that L(N) = A

A language, A. is decidable if there is a decider, N. such that L(N) = A

**Turing Machine Example 2**

This machine decides the language $\{0^{2^n} \mid n \geq 0\}$.

It has input alphabet {0} and tape alphabet { _, 0, x , #}.




Running the machine on input 000:

$q_0000$

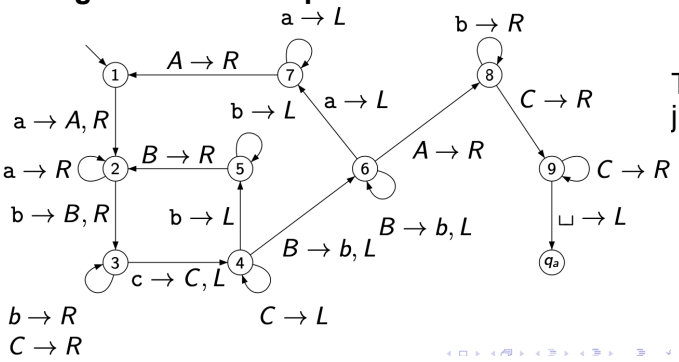
$\Rightarrow \#q_100$

$\Rightarrow \#xq_20$

$\Rightarrow \#x0q_3\_$

$\Rightarrow \#x0\_q_r$

**Turing Machine Example 3**




This machine decides the language $\{a^ib^jc^k \mid k = I \cdot j, I, j > 0\}$. Its tape alphabet is {_, a, b, c, A, B, C}.

**Partial Functions**

$f : X \hookrightarrow Y$ represent f has a domain which is a subset of X, but f(x) may be undefined for some $x \in X$.

Note that a total function $f : X \to Y$ is by definition also partial: $f : X \hookrightarrow Y$ .

**Partial Functions: Example 1**

The function f defined by

$$f(n) = \begin{cases} 42 & \text{if } n = 0 \\ f(n-2) & \text{if } n \neq 0 \end{cases}$$

is a partial function $f : Z \hookrightarrow Z$.

In a natural interpretation, it is undefined if n is odd and/or negative. Its range is {42}.

In this case, it is not too hard to determine the set of values for which f is defined. So we could also choose to say that f is a total function $X \to Z$, where $X = \{n \in Z \mid n \geq 0 \wedge n \text{ is even}\}$.

However, it is not always easy, or even possible, to determine a function's domain.

**Termination:**

Goal: to show a looping program will terminate, e.g. while(condition): <do stuff>

Idea: Associate each state of the program with a value which will strictly decrease on every loop, but cannot decrease infinitely

**Proving Termination**

Suppose that, for each loop in a program, we can find some "measure" (a function of the program variables) such that

1 the measure is a natural number, and

2 the measure gets smaller with each loop iteration.

Then the program must terminate for all input, because a natural number cannot be made smaller indefinitely.

**Well-Founded Orderings**

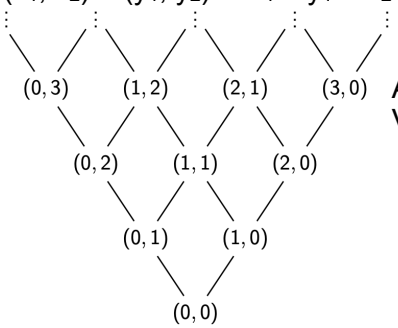
The binary relation $\prec$ over some set X is well-founded iff there is no infinite sequence of X-elements $x_1, x_2, x_3, \ldots$ such that $x_1 \succ x_2 \succ x_3 \succ \cdots$, that is $x_{i+1}$ strictly smaller than $x_i$ for $i \in N$

We say that $(X, \prec)$ is a well-founded **structure**. For example, (N, <) is a well-founded structure.

Given a finite number of well-founded structures $(X_1, \prec_1), \ldots, (X_n, \prec_n)$, we can obtain well-founded orderings of $X = X_1 \times \cdots \times X_n$ in a number of different ways.

**Ordering Pairs: Component-Wise Ordering**

$(x_1, x_2) \preccurlyeq (y_1, y_2)$ iff $x_1 \leq y_1 \wedge x_2 \leq y_2$ $\quad p \prec q$ iff $p \preccurlyeq q \wedge p \neq q$




A Hasse diagram for component-wise ordering of N×N.
Values increase as you travel up along edges.

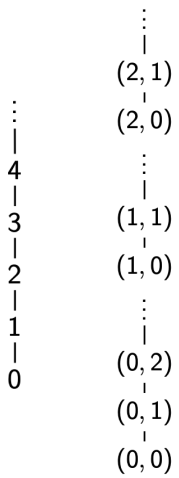
**Ordering Pairs: Lexicographic Ordering**

On the left: N with the usual ordering.

On the right: $(N \times N, \preccurlyeq)$, where $\preccurlyeq$ is lexicographic ordering:

$(m, n) \preccurlyeq (m', n')$ iff $m \leq m' \wedge (m = m' \Rightarrow n \leq n')$.

Define again $p \prec q$ iff $p \preccurlyeq q \wedge p \neq q$.




**Well-Founded Orderings on Tuples**

Theorem: If $\prec$ is well-founded then so is its component-wise extension to tuples.

Theorem: If $\prec$ is well-founded then so is its lexicographic extension to tuples.

**Component-Wise Ordering of Tuples**
Ordering $X = X_1 \times \cdots \times X_n$ component-wise:

$$(x_1, \ldots, x_n) \preceq (y_1, \ldots, y_n) \text{ iff } \bigwedge_{i=1}^{n} x_i \preceq_i y_i$$

If each $(X_i, <_i)$ is well-founded, then so is $(X, <)$.
Example using component-wise ordering: (2,2,2) > (2,2,1) > (2,1,1) > (2,0,1) > (2,0,0) > (0,0,0)

**Lexicographic Ordering of Tuples**
Ordering $X = X_1 \times \cdots \times X_n$ lexicographically:

$$(x_1, \ldots, x_n) \prec (y_1, \ldots, y_n) \text{ iff } \bigvee_{i=1}^{n} \left( x_i \prec_i y_i \wedge \bigwedge_{j=1}^{i-1} x_j = y_j \right)$$

I = 1 j = 1 If each $(X_i, <_i)$ is well-founded, then so is $(X, <)$.
Example using lexicographic ordering: (2,2,2) > (2,1,42) > (1,3,1000) > (1,3,999) > (1,3,0) > (0,0,15)

**Well-Founded Induction**
There is an induction principle to go with well-founded relations. Given a well-founded structure $(X, <)$ we can prove a statement "for all $x \in X$, S(x)" as follows:
We proceed in one step: Assume that S(x′) holds for all $x' < x$, and use that to establish S(x).

**Example Termination proof:**
The state of this algorithm at each loop can be described by a well-founded structure $(A, <)$. Description of A & $<$, and proof of well-foundedness is necessary
After each loop, the associated element $a \in A$ becomes a' and we know that $a' < a$, because…
Since $(A, <)$ is well-founded, the loop cannot be executed infinitely many times.

**Example 2: Ackerman's Function**
The following is a definition of Ackerman's function:
ack(0, y) = y + 1
ack(x + 1, 0) = ack(x, 1)
ack(x + 1, y + 1) = ack(x, ack(x +1, y))
It grows incredibly fast.
For example, ack(3, 3) = 61 and ack(4, 4) = $2^{2^{2^{2^{16}}}} - 3$.
However, lexicographic well-founded induction allows us to conclude that the function is total—as a Haskell program it will terminate for all input (possibly after a very long time).
In each recursive call, the argument (x,y) decreases strictly:

$$\begin{aligned} ack(x+1,0) &= \overbrace{ack(x,1)}^{x \text{ smaller}} \\ ack(x+1,y+1) &= \underbrace{ack(x, \overbrace{ack(x+1,y)}^{x \text{ same, } y \text{ smaller}})}_{x \text{ smaller}} \end{aligned}$$

**Decidability & Undecidability Proofs**
To prove that A is decidable we must prove the existence of a TM which decides A. This can be done by ...
- Specifying a TM which decides A
- Describing a decider for A in pseudocode which could be implemented by a TM, e.g. tutorial 11.3
  - Correctly recognize if the string is in the language or not
  - Terminate on all input
  - Reasonably detailed
- Describing a decider for A in pseudocode, which makes use of a decider for another language, B.

We call this third method a reduction from A to B, since we were tasked with deciding A, and reduced this task to deciding B.

**DFA Acceptance Is Decidable**
Theorem: $A_{DFA}$ is a decidable language.
Proof sketch: The crucial point is that it is possible for a Turing machine M to simulate a DFA D.
M finds on its tape, say

$$\underbrace{1\ldots n}_{Q}\#\#\underbrace{ab\ldots z}_{\Sigma}\#\#\underbrace{1a2\#\ldots\#nbn}_{\delta}\#\#\underbrace{1}_{q_0}\#\#\underbrace{3\ 7}_{F}\#\#\underbrace{baa\ldots}_{w}\$$$

First M checks that the first five components represent a valid DFA, and if not, rejects.
Then M simulates the moves of D, keeping track of D's state and the current position in w, by writing these details on its tape, after $.
When the last symbol in w has been processed, M accepts if D is in a state in F, and rejects otherwise.

**NFA Acceptance Is Decidable**
**Theorem**: $A_{NFA}$ = {⟨N, w⟩ | N is an NFA that accepts w} is a decidable language. Where ⟨N,w⟩ is encoding.
**Proof sketch**: The procedure we gave for translating an NFA to an equivalent DFA was mechanistic and terminating, so a halting Turing machine can do that job.
Having written the encoding of the DFA on its tape, the Turing machine can then "run" the machine M from the previous proof.

**DFA Equivalence Is Decidable**
**Theorem**: $EQ_{DFA}$ = {⟨A, B⟩ | A and B are DFAs and L(A) = L(B)} is decidable.
**Proof sketch**: We previously saw how it is possible to construct, from DFAs A and B, DFAs for $A \cap B$, $A \cup B$, and $A^c$.
These procedures are mechanistic and finite — a halting Turing machine M can perform them.
Hence from A and B, M can produce a DFA C to recognise $L(C) = (L(A) \cap L(B)^c) \cup (L(A)^c \cap L(B))$
Note that $L(C) = \varnothing$ iff L(A) = L(B).
So M just needs to use the emptiness checker on C.

**Generation by CFGs Is Decidable**
**Theorem**: $A_{CFG}$ = {⟨G, w⟩ | G is a CFG that generates w} is decidable.
The proof relies on the fact that we can rewrite any CFG to a particular equivalent form, Chomsky Normal Form.
In Chomsky Normal Form, each production takes one of two forms: $A \to B\ C$ or $A \to a$
(With one exception: We also allow $S \to \varepsilon$, where S is the grammar's start variable.)
For every grammar in Chomsky Normal Form form, if string w can be derived then its derivation has exactly $2|w| - 1$ steps.
So to decide $A_{CFG}$ , we can simply try out all possible derivations of that length, in finite time, and see if one generates w.

**Every CFL Is Decidable**
The decider, call it S, took ⟨G, w⟩ as input.
Now we are saying that any particular CFL $L_0$ is decidable:
Theorem: Every context-free language $L_0$ is decidable.
Proof: This is just saying that we can specialise the decider S.
Let $G_0$ be a CFG for $L_0$. The decider for $L_0$ simply takes input w and runs S on $\langle G_0,w\rangle$.

**An Undecidable Language**
We start by showing that it is undecidable whether a Turing machine accepts a given input string. That is, $A_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ is undecidable.
The main difference from the case of $A_{CFG}$ , for example, is that a Turing machine may fail to halt.

**TM Acceptance Is Undecidable**
Theorem: $A_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ is undecidable.
Proof: Assume (for contradiction) that $A_{TM}$ is decided by a TM H:
$H\langle M, w\rangle =$ ⌈ accept if M accepts w
⌊ reject if M does not accept w
Using H we can construct a Turing machine D which decides whether a given machine M fails to accept its own encoding $\langle M\rangle$:
1. Input is $\langle M\rangle$, where M is some Turing machine.
2. Run H on $\langle M,\langle M\rangle\rangle$.
3. If H accepts, reject. If H rejects, accept.
In summary:
$D(\langle M\rangle) =$ ⌈ accept if M does not accept $\langle M\rangle$
⌊ reject if M accepts $\langle M\rangle$
But no machine can satisfy that specification!
Why? Because we obtain an absurdity when we investigate D's behaviour when we run it on its own encoding:
$D(\langle D\rangle) =$ ⌈ accept if D does not accept $\langle D\rangle$
⌊ reject if D accepts $\langle D\rangle$
Hence neither D nor H can exist.

**Comparing Sizes of Sets: Cantor's Criterion**
So what does 'equals' and 'less' mean for infinite cardinality?
How do we compare the "sizes" of infinite sets? Cantor's criterion:
$\text{card}(X) \leq \text{card}(Y)$ iff there is a total, injective $f : X \to Y$.
$\text{card}(X) = \text{card}(Y)$ iff $\text{card}(X) \leq \text{card}(Y)$ and $\text{card}(Y) \leq \text{card}(X)$.
As a consequence, there are (infinitely) many degrees of infinity.

**To Infinity and Beyond**
X is countable iff $\text{card}(X) \leq \text{card}(\mathbb{N})$.
X is countably infinite iff $\text{card}(X) = \text{card}(\mathbb{N})$.
Examples: $\mathbb{Z}$, $\mathbb{N}^k$ , and $\mathbb{N}^*$ (the set of all finite sequences of natural numbers) are all countably infinite.
Importantly, $\Sigma^*$ is countable for all finite alphabets $\Sigma$, including the alphabet of printable characters on your keyboard.
$\mathcal{P}(\mathbb{N})$, $\mathbb{N} \to \mathbb{N}$, and $\mathbb{Z} \to \mathbb{Z}$ are uncountable, as can be shown by diagonalisation.

**Diagonalisation Showing $\mathbb{Z} \to \mathbb{Z}$ Is Uncountable**
Theorem: There is no bijection $h : \mathbb{N} \to (\mathbb{Z} \to \mathbb{Z})$. Proof: Assume h exists. Then
$h(0), h(1), h(2), \ldots, h(n), \ldots$ contains every function in $\mathbb{Z} \to \mathbb{Z}$, without duplicates.
Now construct $f : \mathbb{Z} \to \mathbb{Z}$ as follows:
$f(n) = h(n)(n) + 1$
Then $f \neq h(n)$ for all n, so we have a contradiction.

**Why This Is Called Diagonalisation**
Here is some hypothetical listing of all the functions h(0), h(1), . . . that make up $\mathbb{Z} \to \mathbb{Z}$:

| | 0 | 1 | 2 | 3 | 4 | 5 | ... |
|---|---|---|---|---|---|---|---|
| h(0) | 19 | 3 | 42 | 0 | 7 | 9 | ... |
| h(1) | 42 | 42 | 42 | 42 | 42 | 42 | ... |
| h(2) | 42 | 43 | 44 | 45 | 46 | 47 | ... |
| h(3) | 6 | 93 | 17 | 84 | 6 | 93 | ... |
| h(4) | 45 | 18 | -8 | -5 | 63 | -9 | ... |
| ⋮ | | | | | | | |

f is defined in such a way that it cannot possibly be in the listing:

| | 0 | 1 | 2 | 3 | 4 | 5 | ... |
|---|---|---|---|---|---|---|---|
| f | 20 | 43 | 45 | 85 | 64 | ... | ... |

**At Least $A_{TM}$ Is Recognisable**
Note that $A_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and M accepts w}\}$ is Turing recognisable.
The reason is that it is possible to construct a universal Turing machine U which is able to simulate any Turing machine.
On input $\langle M, w\rangle$, U simulates M on input w. If M enters its accept state, U accepts.
If M enters its reject state, U rejects. If M never halts, neither does U.

**Hilbert's Tenth Problem**
Find an algorithm that determines whether a polynomial (in many variables but with integer coefficients) has an integral root.
There is no algorithm for it. {p | p is a polynomial with integral root} is undecidable.
However, this language does have a recogniser.
If the input polynomial p has k variables $x_1, \ldots, x_k$ then we can simply enumerate all integer k-tuples $(v_1, ..., v_k)$ and evaluate $p(v_1, \ldots, v_k)$, one by one. If $p(v_1, \ldots, v_k) = 0$, accept. We refer to this type of problem as semi-decidable.

**Closure Properties**
The set of Turing recognisable languages is closed under the regular operations, and intersection. It is not closed under complement.
The set of decidable languages is closed under the same operations, and also under complement.

**Relating Decidability and Recognisability**
Theorem: A language L is decidable iff both L and its complement $L^c$ are Turing recognisable.
Proof: If L is decidable, clearly L and also $L^c$ are recognisable. Assume both L and $L^c$ are recognisable. That is, there are recognisers $M_1$ and $M_2$ for L and $L^c$, respectively. A Turing machine M can then take input w and run $M_1$ and $M_2$ on w in parallel. If $M_1$ accepts, so does M. If $M_2$ accepts, M rejects.
Note that at least one of $M_1$ and $M_2$ is guaranteed to accept. Hence M decides L.

**A Non-Turing Recognisable Language**
This gives us an example of a language which is not Turing recognisable: $(A_{TM})^c$, that is, the complement of $A_{TM}$.
Namely, we know that $A_{TM}$ is recognisable.
If $(A_{TM})^c$ was also Turing recognisable, $A_{TM}$ would be decidable.
But we have shown that it isn't.

**Too Many Languages**
As we have seen, the set {0, 1}* of finite binary strings is a countable set.
Problem set exercise P12.3 asks you to show that the set B of all infinite binary strings is uncountable.
That is interesting, because it follows that the set of all languages over any finite non-empty alphabet Σ is also uncountable.
Namely, the set of all languages over Σ is in a one-to-one correspondence with B, as we now show.
Let $s_1, s_2, s_3, \ldots$ be the standard enumeration of Σ*.
For each language A over Σ, there is a unique characteristic sequence $\chi_A \in \mathbf{B}$, whose *ith* bit is1iff $s_i \in A$:

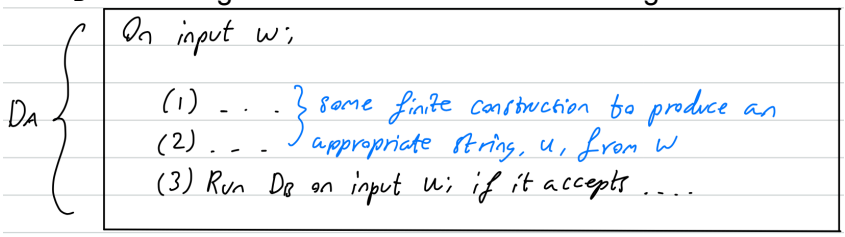
| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $\Sigma^*$ : | { $\epsilon$, | a, | b, | aa, | ab, | ba, | bb, | aaa, | aab, | ...} |
| $A$ : | { | a, | | aa, | ab, | | | aaa, | | ...} |
| $\chi_A$ : | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | ... |

Hence we cannot put the set of all languages into a one-to-one correspondence with the set of all Turing machines.
That is, we could never hope to have a recogniser for each possible language.

**Reducibility**
Let P and P′ be decision problems with instances $p_i$ and $p_i'$ (problem with particular input).
P can be reduced to P′ iff there is a Turing machine M: $\langle p_i\rangle \rightarrow$ M $\rightarrow \langle p_i'\rangle$
such that answer($p_i$) can be obtained from answer(pi′).
- P reducible to P′ and P′ decidable ⇒ P decidable.
- P reducible to P′ and P undecidable ⇒ P′ undecidable.

So reducibility is useful for proving decidability and undecidability results.

**Reducibility example**
Let $D_B$ be a recogniser for B. We construct a recogniser for A

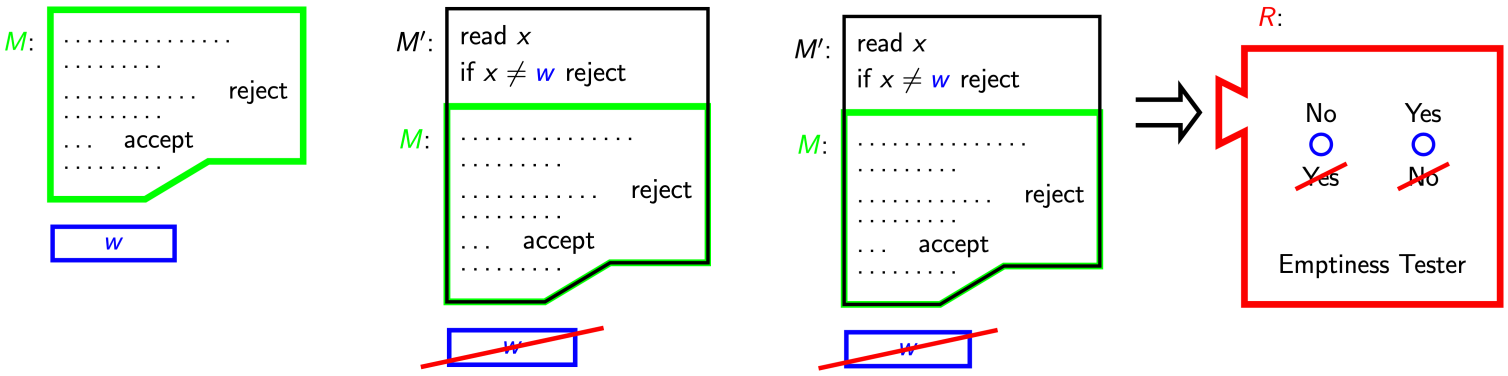



Then B decidable -> A decidable

**Prove B is undecidable.**
Proof: We know A is undecidable from a previous proof.
Suppose B is decidable, and let $M_B$ be a decider for B.
Then the following describes a decider for A (insert reduction from above)
But this contradicts the fact that A is undecidable.
Hence B cannot be decidable - B is undecidable

**TM Emptiness Is Undecidable**
**Theorem**: $E_{TM} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \varnothing\}$ is undecidable.
**Proof:** $A_{TM}$ is reducible to $E_{TM}$. First notice that, given ⟨M, w⟩, a Turing machine can modify the encoding of M, so as to turn M into M′ which recognises L(M) ∩ {w}.

Here is what the new machine M' does:
If input x is not w, reject.
Otherwise run M on w and accept if M does.
Notice how w has been "hard-wired" into M′: This machine is like M, but it has extra states added to perform the test against w.
Also note that $L(M') = \begin{cases} \{w\} & \text{if M accepts w} \\ \varnothing & \text{otherwise} \end{cases}$
Here then is a decider for $A_{TM}$, using a decider R for $E_{TM}$:
1 From input ⟨M, w⟩ construct ⟨M′⟩.
2 Run R on ⟨M′⟩.
3 If R rejects, accept; if it accepts, reject.




**TM Equivalence Is Undecidable**
Theorem: $EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1, M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$ is undecidable.
Proof: $E_{TM}$ is reducible to $EQ_{TM}$.
Assume that S decides $EQ_{TM}$. Here is a decider for $E_{TM}$:
1 Input is ⟨M⟩.
2 Construct a Turing machine $M_\varnothing$ which rejects all input.
3 Run S on $\langle\langle M\rangle, \langle M_\varnothing\rangle\rangle$.
4 If S accepts, accept; if it rejects, reject.
But we know that $E_{TM}$ is undecidable. So $EQ_{TM}$ is undecidable.

**Valid and Invalid Computations**
Recall how we captured a Turing machine configuration as a string (such as $baq_5bb$).
A valid computation for Turing machine $M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r \rangle$ (on input w) is a string of form
$C_1\#C_2\#\cdots\#C_k\#$
where
1 $C_1$ is M's start configuration,
2 $C_k$ is an accepting configuration,
3 for each even $i \in \{2, \ldots, k\}$, $C_{i-1} \Rightarrow C_i^R$ where R means reversed string
4 for each odd $i \in \{2, \ldots, k\}$, $(C_{i-1})^R \Rightarrow C_i$

A string (over the same alphabet) which is not a valid computation is an invalid computation.
Rephrasing: A string w is an invalid computation iff one or more of the following properties hold:
0. w is not of the form $C_1\#\cdots\#C_k\#$ with $C_i$ in $\Gamma^*Q\Gamma^*$.
1. $C_1$ is not a start configuration for M , that is, not in $q_0\Sigma^*$ .
2. $C_k$ is not accepting, that is, not in $\Gamma^*q_a\Gamma^*$.
3. $C_{i-1} \not\Rightarrow C_i^R$ for some even i.
4. $(C_{i-1})^R \not\Rightarrow C_i$ for some odd i.

The set of strings satisfying (0) – (2) are regular languages.
We claim the set of strings satisfying (3) is context-free (and so is the set satisfying (4)).

Here is how to build a PDA P for the set of strings for which $C_{i-1} \Rightarrow C_i^R$ is false for some even i.

- P non-deterministically skips past an even number of # symbols.
- While reading through $C_{i-1}$, P pushes onto its stack the symbols of the configuration C obtained by $C_{i-1} \Rightarrow C$.
- After skipping the next #, P compares C (backwards) against the string found until the following #; if they are different, P scans over the remaining input and accepts.

Conclusion: The language of invalid computations is context-free!

**CFG Exhaustiveness is Undecidable**
Theorem: $ALL_{CFG} = \{\langle G \rangle \mid G \text{ is a CFG and } L(G) = \Sigma^*\}$ is undecidable.
Proof: We just saw that, for any Turing machine M, we can construct a CFG G that generates all the invalid computations for M.
Clearly $L(G) = \Sigma^*$ iff $L(M) = \varnothing$.
Hence if $ALL_{CFG}$ is decidable, then so is $E_{TM}$.
Since we proved the latter undecidable, $ALL_{CFG}$ must be undecidable as well.

**CFG Equivalence is Undecidable**
Our final language undecidability result is an easy reduction from $ALL_{CFG}$ .
Theorem: $EQ_{CFG} = \{\langle G, G' \rangle \mid G, G' \text{ are CFGs and } L(G) = L(G')\}$ is undecidable.
Proof: Assume we have a decider E for $EQ_{CFG}$. Then we can easily build a decider for $ALL_{CFG}$. Namely, given input grammar G over alphabet $\Sigma$, construct a CFG Gall for $\Sigma^*$. Then run E on $\langle G, G_{all} \rangle$.

**Undecidability in Logic**
Validity in propositional logic is decidable.
Validity in first-order predicate logic is undecidable, however, it is semi-decidable.
However, it is not the quantifiers per se that cause that problem. We cross the boundary to the undecidable only when the quantifiers appear together with predicates of arity greater than 1.
Monadic logic, that is, the fragment of predicate logic which does not allow predicates of arity 2 and above is decidable.

**Rice's Theorem**
We have this rather sweeping result:
Rice's Theorem: Every interesting semantic Turing machine property is undecidable!
A property P is interesting iff $P(M_1)$ and not $P(M_2)$ for some Turing machines $M_1$ and $M_2$.
It is semantic iff $P(M_1)$ iff $P(M_2)$ for all Turing machines $M_1$ and $M_2$ such that $L(M_1) = L(M_2)$.

## Closure Properties

| | $\cup$ | $\circ$ | $^*$ | $R\cap$ | $\cap$ | compl |
|---|---|---|---|---|---|---|
| Reg | Y | Y | Y | Y | Y | Y |
| DCFL | N | N | N | Y | N | Y |
| CFL | Y | Y | Y | Y | N | N |
| Decidable | Y | Y | Y | Y | Y | Y |
| Recognisable | Y | Y | Y | Y | Y | N |

Here '$\circ$' means concatenation, '$^*$' means Kleene star, and '$R\cap$' means "intersection with a regular language". DCFL is the class of languages that can be recognised by deterministic PDAs (DPDAs).

## Decidability of Language Properties

| Question | Reg | DCFL | CFL | Decidable | Recognisable |
|---|---|---|---|---|---|
| $w \in L$ | D | D | D | D | U |
| $L = \emptyset$ | D | D | D | U | U |
| $L = \Sigma^*$ | D | D | U | U | U |
| $L_1 = L_2$ | D | D | U | U | U |
| $L =$ given $R$ | D | D | U | U | U |
| $L$ regular | D | D | U | U | U |
| $L_1 \subseteq L_2$ | D | U | U | U | U |

Here 'D' = decidable; 'U' = undecidable.