Comprehensive CS

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
United States

Contents

Theory of Computation

1.1 Natural Languages, Formal languages: Definitions and theorems

- **Gödel's incompleteness theorem**: Gödel's Incompleteness Theorems are two fundamental results in mathematical logic that state:
  - Proved that for some axiomatic systems that there is no algorithm that will generate all true statements from those axioms.
  - No such system can prove its own consistency.

  This was the first indication that there are inherent limts on algorithms
- **Turing**: Alan Turing later provided formalism to the concepts of an "algorithm" and "computation", he Invented definition for an abstract machine called the "universal algorithm machine", he Provided means to formally (i.e., with mathematical rigor) explore the boundaries of what algorithms could, and could not, accomplish. Turing's model for a universal abstract machine was the basis for the first computer – in fact, Turing was involved in the construction of the first computer.
- **Natural languages**: We communicate via a *natural language*, Although we don't often think about it, our language is guided by rules; spelling, grammar, punctuation
- **Formal language**: Formal languages, which are not intended for human-to- human communication, are similar to natural languages in that they too have rules that define "correct" words and statements, but they are also different than natural languages in two key ways;
  - The rules that define a formal language are strictly enforced. There is no tolerance for misspellings, bad grammar, etc.
  - For the purpose of determining if a word or statement is acceptable in a formal language, meaning is ignored. Determining if something is (or is not) part of a language is determined by the language's defining rules which do not attach meaning (i.e., no definitions of words like in natural languages)

  In short, formal languages is a game of symbols, not meaning
- **Formal Languge terminology**:
  - **Symbol**: it is an abstract entity that is not formally defined – like a point or a line in geometry – but think of it as a single character like a letter, numerical digit, punctuation mark, or emoticon
  - **String (or Word)**: A finite sequence (i.e., order matters) of zero or more symbols
  - **Length**: The length of a string w is denoted by $length(w)$ or $|w|$ and is the number of symbols composing the string. Because strings, by definition, are finite then a string's lengths is always defined (sometimes zero).
  - **Prefix, suffix**: Any number of leading/trailing symbols of the string.
  - **Concatenation**: The concatenation of two strings w and x is formed by writing the first string w then the second string x

Note: For any string w, $\Lambda w = w\Lambda = w$

- **Alphabet**: A finite set of symbols, typically denoted by the Greek capital letter sigma Σ, for example

$$\Sigma = \{a, b, c\} \quad \Sigma = \{0, 1\} \quad \Sigma = \varnothing \quad \text{(special case)}.$$

- **The empty string**: A string with zero symbols is called the empty string and is denoted by the capital Greek letter lambda Λ, or sometimes lower case Greek letter epsilon ϵ, where Λ and ϵ are **not** symbols

  Thus,

$$|\Lambda| = 0.$$

- **Formal language definition**: A formal language is a et of strings from some **one** alphabet. Given an alphabet we generally define a formal language over that alphabet by specifying rules that either;

  1. Tell us how to test a candidate word, or
  2. Tell us how to construct all words.

  For example, Given $\Sigma_1 = \{x\}$, we can define languages

  $L_1 =$ any non empty string $= \{x, xx, xxx, ...\}$
  $L_2 = \{X^n : \ x = 2k + 1, \ k \in \mathbb{Z}\} = \{x, xxx, xxxxx, xxxxxxx, ...\}$ $L_3 \quad = \{x, xxxxxxxx\}$.

- **The empty language**: The empty language $L = \varnothing$ is typically denoted with the capital greek letter phi Φ. Thus, $L = \varnothing = \Phi$

- **Notes on formal languages**:

  - All languages are defined over some alphabet; cannot define a language without an alphabet.
  - Some languages are finite, some languages are infinite (remember, alphabets are always finite).
  - Some languages include the empty string Λ, some do not.
  - Some languages are defined by rules, some are simply written completely (e.g., $\Sigma_1 = \{x\}$, $L_3 = \{x, \text{xxxxxxxxxx}\}$).
  - No matter what the alphabet Σ (even $\Sigma = \emptyset$), you can always define at least two languages; $L_1 = \{\Lambda\}$ and $L_2 = \emptyset$.

- **Closure of an alphabet (closure of Σ) (Kleene closure)**: The language defined by the set of all strings (including the empty string Λ) over a fixed alphabet Σ.

  - **Examples:**

$$\begin{aligned} \Sigma &= \{a\} & \Sigma^* &= \{\Lambda, a, aa, aaa, aaaa, \dots\} \\ \Sigma &= \{0, 1\} & \Sigma^* &= \{\Lambda, 0, 1, 00, 01, 10, 11, 000, \dots\} \\ \Sigma &= \emptyset & \Sigma^* &= \{\Lambda\} \end{aligned}$$

    Note: If $\Sigma = \emptyset$ then Σ^* is finite and $\Sigma^* = \{\Lambda\}$, otherwise Σ^* is infinite.

- **Positive closure**: $\Sigma^+ = \Sigma^* - \{\Lambda\}$, you just take the empty string out of the kleene closure

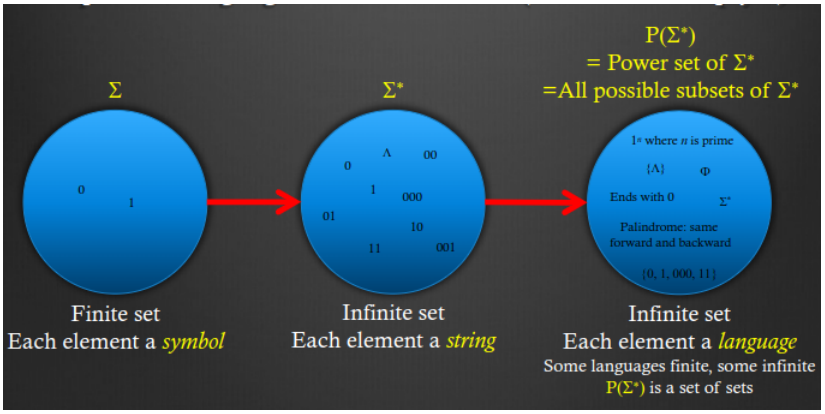
- **Recall: Power set**: The power set of any set S, written $\mathcal{P}(S)$ is the set of all subsets of S, including the empty set and the set S itself.

In other words, given a set S, then its power set $\mathcal{P}(S)$ is a set of sets

- **Note:**
  - If $S = \emptyset$, then then $\mathcal{P}(S) = \mathcal{P}(\emptyset) = \{\emptyset\} = \{\emptyset\}$ = a set with one element = $\emptyset$.
  - If S is non-empty and finite with n elements, then $\mathcal{P}(S)$ will be finite with 2^n elements.
  - If S is infinite, then $\mathcal{P}(S)$ will be infinite.
- **Example:**
  If $S = \{x, y, z\}$, then $\mathcal{P}(S)$ will have the following $2^3 = 8$ elements (each a set):

$$\mathcal{P}(S) = \{\emptyset, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$$

- **Power set of the kleene closure** $\mathcal{P}(\Sigma^\star)$: Given some alphabet Σ we can construct the set of all possible languages from Σ as follows (assume non-empty Σ):



- **From formal languages to computers**:
  - Given an alphabet Σ we can define many formal languages – the range of which is captured by $\mathcal{P}(\Sigma^*)$.
  - We can define many formal languages verbally, but is there a way to define/express every language in any $\mathcal{P}(\Sigma^*)$ with some formal system or abstract machine?
  - We search for a formal system or abstract machine with enough "power" to define any language in any $\mathcal{P}(\Sigma^*)$.
  - **KEY POINT**
    The abstract machines we discover along our search to cover $\mathcal{P}(\Sigma^*)$ turn out to be *the theoretical basis for all computing.*
  - In other words, by understanding the power (and limitations) of abstract machines that cover $\mathcal{P}(\Sigma^*)$, we are simultaneously discovering the same limits about all computing.

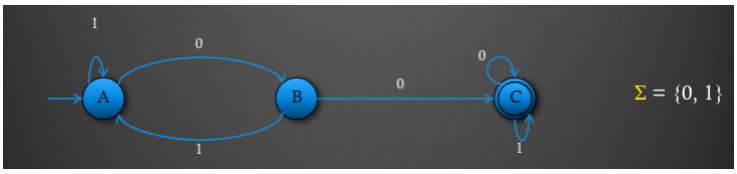
1.2 Regular languages

Preface. The first few subsubsections will be in the world of regular languages. In the context of computation theory, regular languages are a class of formal languages that can be recognized by finite automata. These languages are important because they are the simplest class of languages that can be described by a computational model. The characteristics of regular languages are as follows,

- **Finite Automata:** Regular languages can be recognized by deterministic or nondeterministic finite automata (DFA or NFA).
- **Regular Expressions:** Regular languages can be described using regular expressions.
- **Closure Properties:** Regular languages are closed under several operations, including:
  - **Union:** The union of two regular languages is also regular.
  - **Concatenation:** The concatenation of two regular languages is also regular.
  - **Kleene Star:** The Kleene star operation, which involves repeating a regular language any number of times (including zero), results in a regular language.
  - **Intersection and Difference:** Regular languages are also closed under intersection and difference.
- **Decision Problems:** Certain decision problems are decidable for regular languages. For example, it is possible to determine whether a given string belongs to a regular language (membership problem), whether two regular languages are equivalent, or whether a regular language is empty.

1.2.1 Finite Automata

- **Informal definition**: Described informally, a finite automaton (FA) is always associated with some alphabet Σ and is an abstract machine which has
  1. A non-empty finite number of states, exactly one of which is designated as the "start state" and some number (possibly zero) of which are designated as "accepting states".
  2. A transition table that shows how to move from one state to another based on symbols in the alphabet Σ
- **A simple example of a FA**:



  - Defined over alphabet $\Sigma = \{0, 1\}$.
  - States are circles; transitions are directed edges (i.e., arrows) between states.
  - Has exactly three states; **A**, **B**, and **C**.
  - Every FA must have exactly one start state. In this example, the start state is **A** and denoted as the only state that has an edge coming to it from no other state.

- There is only one accepting state, **C**, and it is denoted by its *double circle*. (We could have more than one but in this case we only have one)
- **Very important:**
  - Each symbol in the alphabet has exactly one associated edge leaving every state.
  - In other words, every state must have exactly one edge leaving it for each symbol in the alphabet.

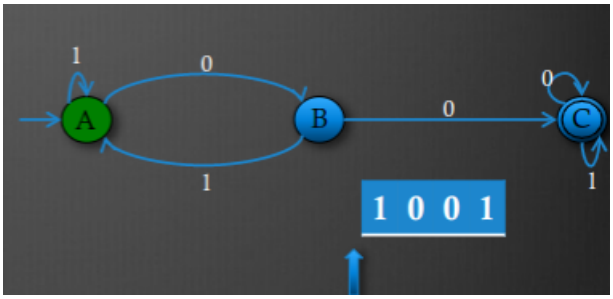
- **How to use an FA**: The purpose of a FA is to define a language over its alphabet Σ. The FA provides the means by which to test a candidate string from Σ and determine whether or not the string is in the language. It does this by "writing" the candidate string on an fictitious input tape and proceeding as follows:

  1. Set the FA to the start state.
  2. If end-of-string then halt.
  3. Read next symbol on tape.
  4. Update the state according to the current state and the last symbol read.
  5. Goto step 2.

  When the process halts check which state the FA is in. If it is in any accepting state, then the string is in the language defined by the FA, otherwise the string is not in the language

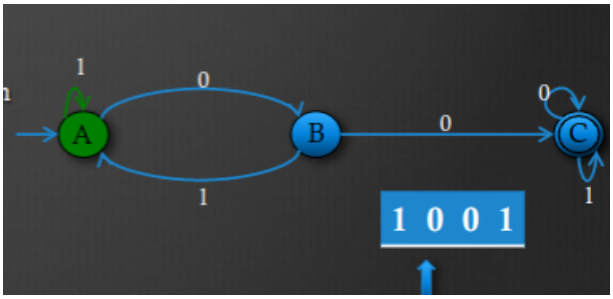
- **Using the previous FA**: Let's now try to use our FA to test whether or not the string 1001 is in the language

  We start by writing the string on an input tape, placing the read head at the beginning of the tape, and placing the FA in its initial state, *A*



  Since the tape head is not at the end of the tape we

  1. Read the next symbol from the tape.
  2. Follow the edge from the state we are currently in that corresponds to the symbol we just read to transition to the next state.
  3. Move the tape head

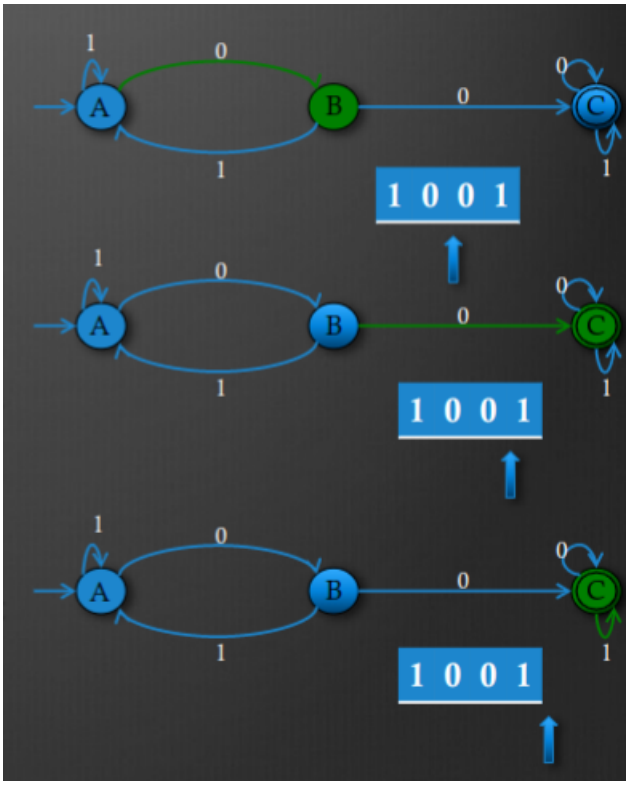


In this case, we started in state A, read symbol 1, and followed the edge labeled 1 from A which brought us back to A

We proceed in this way, read, change state, move tape head until we reach the end of the tape

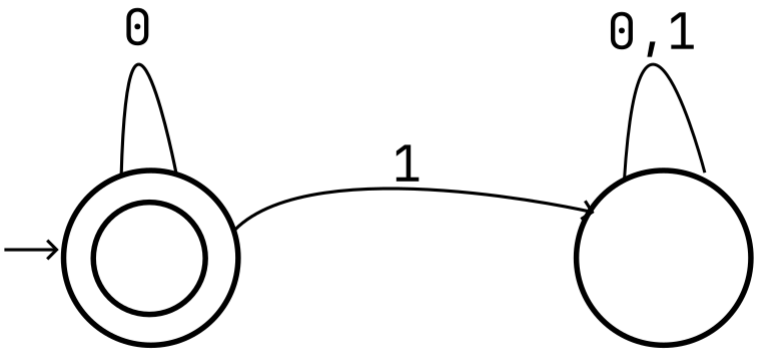
Once the tape head reaches the end of the tape we simply look to see whether or not the FA ended in an accepting state.

In this case it ended in state C, which is an accepting state, which means that string 1001 is in the language.



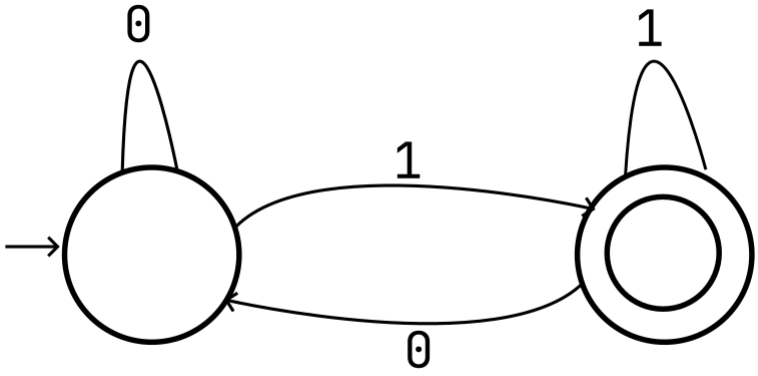
We deduce that the language has only strings with two consecutive zeroes somewhere.

- **FA Example Two: The set of all strings that do not contain a one** ($\Sigma = \{0, 1\}$):

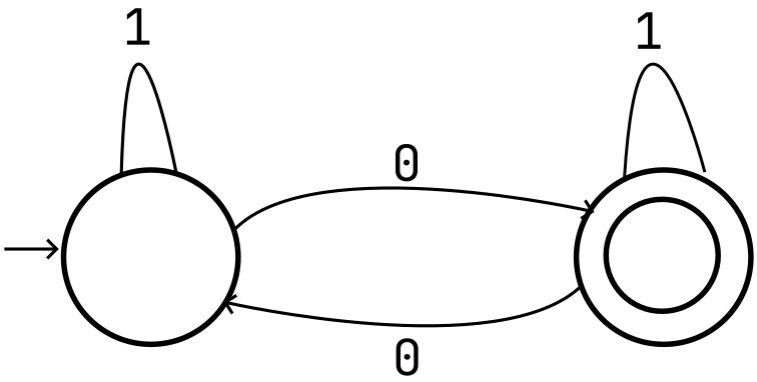


This one is pretty simple. If we have a zero, stay in the accepting state, if we see a one, toss it to the other non-accepting state, its not coming back.

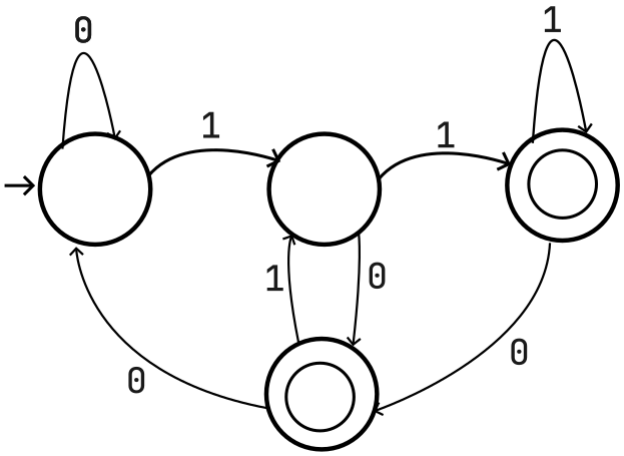
- **FA Example Three: The set of all strings that end in one** ($\Sigma = \{0, 1\}$):



- **FA Example Four: The set of all strings with an odd number of zeros** ($\Sigma = \{0, 1\}$):



- **FA Example Five: The set of all strings where the second to last symbol is one** ($\Sigma = \{0, 1\}$):



- **States are "memory"**: Consider the four FA we just created, in each instance the solution required us to design an FA that remembered at least part of what it had already read from the input tape. The type of memory that an FA has is very different than the RAM we find in contemporary computers, but the FA does have memory. Each time the FA enters a different state it is, in effect, redefining the memory of the entire FA. The FA can only be in a finite number of states, and that number can be arbitrarily large, but (as we will see) that difference in memory has a profound limiting effect in what FAs can compute.

- **Limits of a FA**:

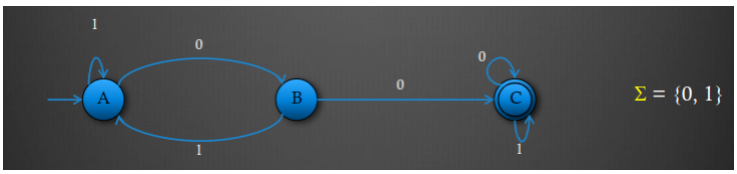
  Limited Memory:

- **Finite State:** A finite automaton has a finite number of states. This means it can only "remember" a limited amount of information about the input it has processed. Once a finite automaton transitions to a new state, it forgets all previous information except for the current state.
- **No Stack or Tape:** Unlike more powerful models such as pushdown automata (which have a stack) or Turing machines (which have an infinite tape), finite automata cannot use any form of auxiliary memory to keep track of an unbounded number of items or to perform operations that require more complex memory management.

Inability to Count Unboundedly:

- **No Arbitrary Counting**: Finite automata cannot count occurrences of symbols beyond the number of states they have. For example, a DFA with n states can only count up to $n-1$ occurrences of a symbol reliably. Thus, they cannot recognize languages that require matching counts of different symbols if those counts are unbounded, such as $\{a^n b^n \mid n \geqslant 1\}$, where the number of 'a's must match the number of 'b's.

- **FA Formal Definition**: We formally denote a *finite automaton* by a 5-tuple $(Q, \Sigma, q_0, T, \delta)$, where
  - Q is a finite set of *states*.
  - Σ is an alphabet of *input symbols*.
  - $q_0 \in Q$, is the *start state*.
  - $T \subseteq Q$, is the set of *accepting states*.
  - δ is the *transition function* that maps a state in Q and a symbol in Σ to some state in Q. In mathematical notation, we say that $\delta : Q \times \Sigma \to Q$. With:
    - $Q \times \Sigma$: The Cartesian product of the set of states Q and the alphabet Σ. This represents all possible pairs of a state and an input symbol.
    - $\to Q$: Indicates that the transition function maps each pair (q, σ) (where $q \in Q$ and $\sigma \in \Sigma$) to a single state in Q.
- **Formally Specifying Our First FA**:



Recall our first FA that accepts any string with two consecutive zeros somewhere.

We drew it as a Finite State diagram, but to formally define this FA we must specify each of the elements from the 5-tuple $(Q, \Sigma, q_0, T, \delta)$.

- Q is a finite set of *states*: $Q = \{A, B, C\}$
- Σ is an alphabet of *input symbols*: $\Sigma = \{0, 1\}$
- $q_0 \in Q$, is the *start state*: $q_0 = A$
- $T \subseteq Q$, is the set of *accepting states*: $T = \{C\}$

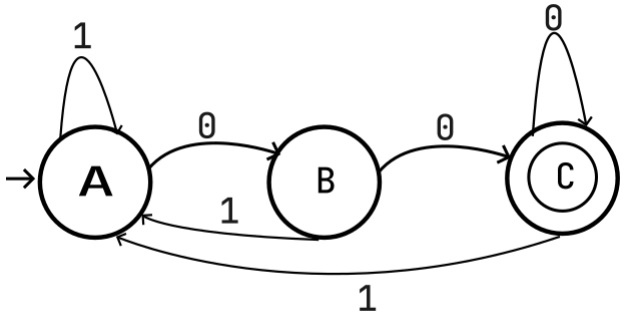
– δ is the *transition function* $\delta : Q \times \Sigma \to Q$

δ	0	1
A	B	C
B	C	A
C	C	C

- **Unary**: consisting of or involving a single component or element.
- **Unary language**: One where the alphabet has only one symbol.
- **Binary**: Relating to, composed of, or involving two things.
- **Ternary**: Composed of three parts.
- **Dead state (trap state)**: This is a state that once entered, can never be left.
- **Deterministic finite automaton (DFA)**: The FA's we have looked at thus far have been DFA's. A DFA is a finite automaton where, for each state and each input symbol, there is exactly one transition to a new state. This means that given a current state and an input symbol, the next state is uniquely determined. In the future we will look at nondeterministic finite automaton (NFA). An NFA is a finite automaton where, for each state and input symbol, there can be multiple possible transitions to different states. Additionally, an NFA can have transitions that do not consume any input symbol (ϵ-transitions).

1.2.2 Finite Automata: More examples

- $\Sigma = \{0, 1\}$**, all strings that start with 00**
- $\Sigma = \{0, 1\}$**, all strings that end with 00**



With:

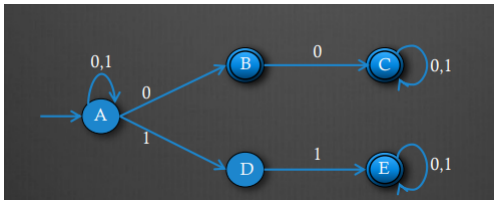
- $Q = \{A, B, C\}$
- $\Sigma = \{0, 1\}$
- $q_0 = A$
- $T = C$
- $\delta :\ Q \times \Sigma \to Q$ defined by

δ	0	1
A	B	A
B	C	A
C	C	A

1.2.3 nondeterministic Finite automata (NFA)

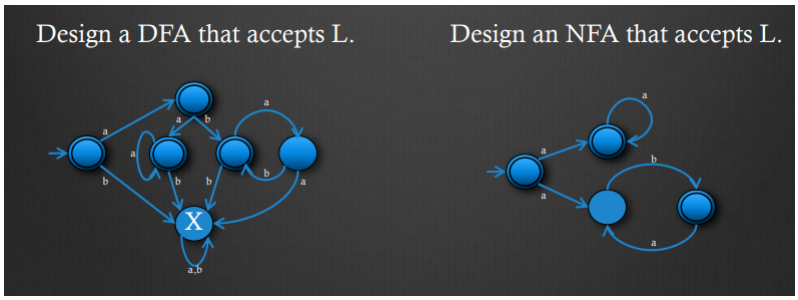
- **NFA definition**:
  - If an automaton gets to a state where there is more than one possible transition corresponding to the symbol read from the tape, the automaton may choose any of those paths. (nondeterminism) We say it **branches**
  - if an automaton gets to a state where there is no transition for the symbol read from the tape, then that path of the automaton halts and rejects the string. We say it **dies**
  - the automaton accepts the input string if and only if there exists a choice of transitions that ends in an accept state.

  Example: Consider this nondeterministic FA (NFA) over $\Sigma = \{0, 1\}$

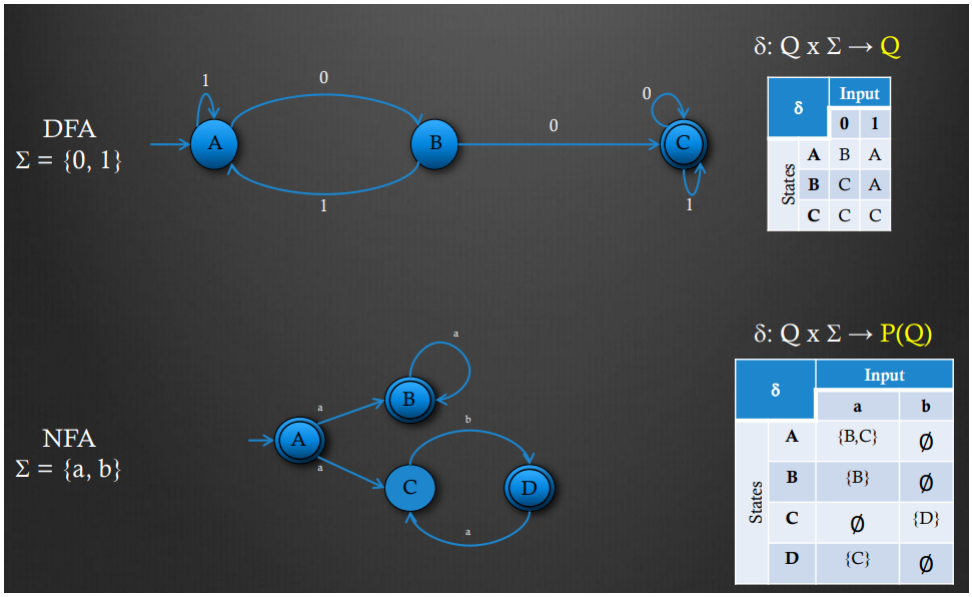


- **DFA or NFA?**: Consider the language L over $\Sigma = \{a, b\}$ which is defined by

$$L = (a^*) + (ab)^*.$$



- **NFA Formal definition**: We define an NFA $M(Q, \Sigma, q_0, T, \delta)$
  - Q is a finite set of states
  - Σ is an alphabet of input symbols
  - $q_0 \in Q$ is the start state
  - $T \subseteq Q$ is the set of accepting states
  - δ is the transition function $\delta : Q \times \Sigma \to P(Q)$
- **Transition function, DFA vs NFA**:

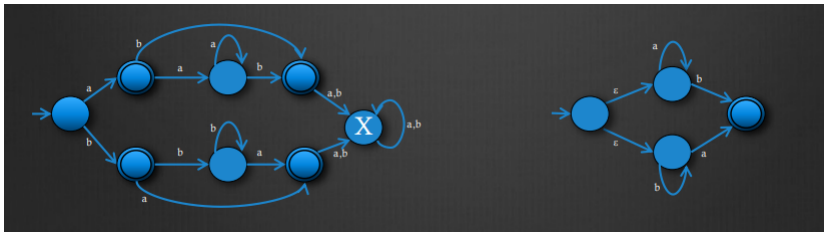


- **NFA with ϵ-transitions**: ϵ-transitions allow the automaton to change state without consuming an input symbol

  Changing states without consuming input symbols can go on arbitrarily long as there are ϵ-transitions to traverse.

- **DFA or NFA with ϵ-moves?**: Consider the language L over $\Sigma = \{a, b\}$ which is

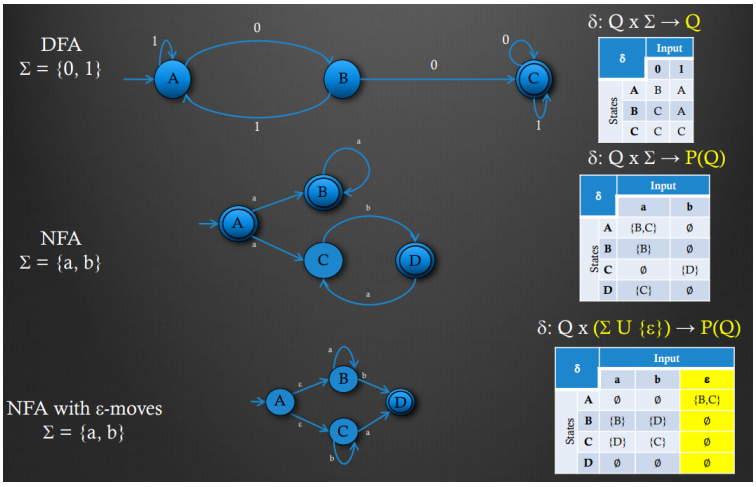
$$L = (b^*a) + (a^*b).$$



- **NFA with ϵ-transitions formal definition**: Everything is the same except for the transition function, we now have

$$\delta : \ Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q).$$

- δ – **DFA, NFA, and NFA with ϵ-moves**:



- **DFA, NFA, or NFA with ϵ moves, who can define the most languages?**: We begin by noting, by definition, every DFA is an NFA. This means that any language you can define with a DFA can also be defined by an NFA. Thus,

$$\text{Languages defined by DFA} \subseteq \text{ Languages defined by NFA.}$$

  Also, by definition, every DFA is an NFA with ϵ-moves, an NFA is an NFA with ϵ moves, even if it doesnt have any. Thus,

$$\text{Languages defined by DFA} \subseteq \text{ Languages defined by NFA with } \epsilon\text{-moves.}$$

  But, by definition, every NFA is an NFA with ϵ-moves. Thus,

$$\text{Languages defined by NFA} \subseteq \text{ Languages defined by NFA with } \epsilon\text{-moves.}$$

  This tells us that

  - NFAs are at least as powerful in defining languages as DFAs
  - NFAs with ϵ-moves are at least as powerful in defining languages as DFAs and NFAs.

  It turns out that these three are **equally** as powerful. We assert

$$\begin{aligned}&\text{Languages defined by DFA's}\\ &= \text{Languages defined by NFA's}\\ &= \text{Languages defined by NFA's with } \epsilon\text{-moves.}\end{aligned}$$

  We prove this by showing an algorithm that converts any NFA with ϵ-moves (or any NFA) to a DFA that accepts the exact same language

  This means that there does not exist a language that can be defined by an NFA with ϵ-moves (or NFA) that cannot also be defined by a DFA.

- **ϵ-closure**: Before we can look at the algorithm we must first define the ϵ-closure of a set of states

Given:

- an NFA with ϵ-moves $M(Q, \Sigma, q_0, T, \delta)$
- Some set of states $S \subseteq Q$

We define the ε-closure(S) as the set of states that are reachable from the set of states S using only zero or more ε-moves in δ.

Note: it is always the case that S $\subseteq \varepsilon$-closure(S)

The formal definition is

$$\epsilon - \text{closure}(q) = \{q\} \cup \{p : \ q \xrightarrow{\epsilon} p\}.$$

- **ϵ-closure alternate notation**.

$$\epsilon\text{-closure}(\{A\}) = \epsilon(\{A\}) = E(\{A\}).$$

- **ϵ-closure of the empty set $\varnothing$**: The epsilon closure of the empty set is $\epsilon(\varnothing) = \varnothing$
- **Algorithm: Converting NFA with ϵ-moves to DFA**: The algorithm constructs a new DFA $M'(Q', \Sigma, q_0', T', \delta')$ From an NFA with ϵ-moves $M(Q, \Sigma, q_0, T, \delta)$. Σ will remain the same

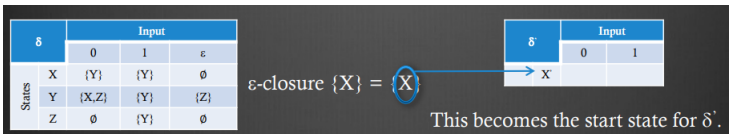
Things to note about the conversion:

- Same alphabet Σ
- Lose column ϵ
- Lose all nondeterminism
- Lose all empty sets
- Cell values change from sets of states to states

Example: **Consider the following NFA with ε-moves M(Q, Σ, q_0, T, δ) over $\Sigma = \{0, 1\}$ and its associated transition table δ: Q $\times$ ($\Sigma \cup \{\varepsilon\}$) $\rightarrow$ P(Q)**

	0	1	ε
X	$\{Y\}$	$\{Y\}$	$\emptyset$
Y	$\{X, Z\}$	$\{Z\}$	$\{Z\}$
Z	$\emptyset$	$\{Y\}$	$\emptyset$

Start by computing the ϵ-closure of the start state in δ.



There is a subtle - but very important - point to be made here ...

we cannot simply take the ϵ-closure (a set) and use it to create a row in δ' (which needs to be a state). What we do is create a label for the new state in δ' that represents the set of states from δ and then add that new state to δ'

In this instance we represented the set of states $\{X\}$ by a single state whose label is X'

We continue by filling the columns of the start state for each symbol $\Sigma = \{0, 1\}$

Processing δ' state X' which represents the set of states $\{X\}$ in M:

- Processing input symbol 0 (process each state in $\{X\}$ using δ):
  - Process X

$$\delta(X, 0) = \{Y\}$$

$$\varepsilon\text{-closure}(\{Y\}) = \{Y, Z\}$$

Since there are no more states in $\{X\}$ to process, we have finished processing the symbol 0 and have produced the set of states $\{Y, Z\}$.

We create a new state with label $Y'Z'$ (or $Z'Y'$, order does not matter) for δ' that represents $\{Y, Z\}$ in M and define:

$$\delta'(X', 0) = Y'Z'$$

We note that $Y'Z'$ is a new state in δ' and so we create a new row for it in δ'.

We continue this until we reach

δ'	Input	
	0	1
X'	Y'Z'	Y'Z'
Y'Z'		

Processing δ' state $Y'Z'$ which represents the set of states $\{Y, Z\}$ in M:

- Processing 0:
  - Process Y

$$\delta(Y, 0) = \{X, Z\}, \quad \varepsilon\text{-closure}(\{X, Z\}) = \{X, Z\}$$

  - Process Z

$$\delta(Z, 0) = \emptyset, \quad \varepsilon\text{-closure}(\emptyset) = \emptyset$$

Here is our first instance of processing a state and symbol where the state in δ' represents multiple states in NFA M. When this happens, the set of states in NFA M is computed by *taking the union of the ε-closures*: $\{X, Z\} \cup \emptyset = \{X, Z\}$.

This produces a new label $X'Z'$ which we use to define:

$$\delta'(X'Y', 0) = X'Z'$$

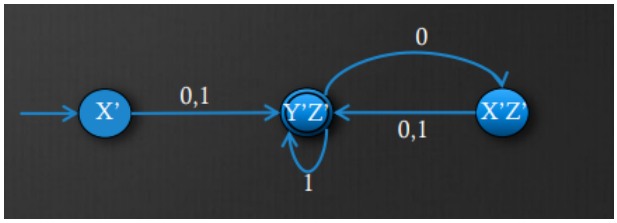
and since $X'Z'$ is a new state, we add it to δ'.

We continue this until we reach

δ'	Input	
	0	1
X'	Y'Z'	Y'Z'
Y'Z'	X'Z'	Y'Z'
X'Z'	Y'Z'	Y'Z'

A state in M' is an accepting state iff at least one of the states that it represents in M is an accepting state … in this case $T' = \{Y'Z'\}$.

We can now draw the new DFA



Note: If the closure or union of closures is the empty set, we do this

δ'	Input	
	0	1
A'	B'C'	*empty*
B'C'		
empty	*empty*	*empty*

This ”emtpy” is a state and represents a garbage state, what goes does not leave.

- **Kleene's theorem revisited**: The following are equivalent for a language L
  1. There is a DFA for L
  2. There is an NFA for L
  3. There is an RE for L

- **Union of two DFA's (cartesian product construction)**: The process of finding the union of two deterministic finite automata (DFAs) involves creating a new DFA that accepts the union of the languages accepted by the original DFAs. This is done using a product construction (also called the Cartesian product construction), where you combine the states of both DFAs in a systematic way to ensure the resulting DFA accepts strings from either of the original DFAs.

  Let's say we have two DFAs:

  $$D_1 = (Q_1, \Sigma, \delta_1, q_1^{\text{start}}, F_1)$$

  that recognizes language L_1.

  $$D_2 = (Q_2, \Sigma, \delta_2, q_2^{\text{start}}, F_2)$$

  that recognizes language L_2.

  Create a New DFA State Set:

  - The states of the new DFA are pairs of states, one from each of the original DFAs. The new state set will be the Cartesian product $Q_1 \times Q_2$, meaning every possible combination of a state from D_1 and a state from D_2.
  - If D_1 has n states and D_2 has m states, the new DFA will have $n \times m$ states.

  Define the New Start State:

  - The new start state is $(q_1^{\text{start}}, q_2^{\text{start}})$, where q_1^{start} is the start state of D_1 and q_2^{start} is the start state of D_2.

  Define the New Transition Function:

  - The transition function δ for the new DFA operates by taking an input symbol and applying the transition functions of both original DFAs in parallel.
  - For each input symbol $a \in \Sigma$, the new DFA transitions from state (q_1, q_2) to state $(\delta_1(q_1, a), \delta_2(q_2, a))$.
  - In other words, if q_1 moves to q_1' on input a in D_1, and q_2 moves to q_2' on input a in D_2, the new DFA will move from (q_1, q_2) to (q_1', q_2').

  Define the New Set of Accepting (Final) States: The new DFA will accept a string if either of the original DFAs would accept it. Therefore, the set of final states F in the new DFA is defined as:

  $$F = \{(q_1, q_2) \mid q_1 \in F_1 \text{ or } q_2 \in F_2\}$$

  This means that if either q_1 is a final state in D_1, or q_2 is a final state in D_2, the pair (q_1, q_2) is a final state in the new DFA.

  Note: It is possible in the new DFA (constructed as the union of two DFAs) to have states that are unreachable—meaning there are states in the DFA that cannot be reached from the start state. This typically happens because, in the product construction, we generate all possible pairs of states from the two original DFAs, but not all of these pairs are necessarily reachable.

The union of two finite automata (FAs) is useful for constructing a new automaton that recognizes any string accepted by either of the two original automata. This has several practical applications in theoretical computer science and programming:

- **Finding the intersection of two DFA's**: The process is basically the same as finding the union, but it differs in how we define the accepting states in the new machine, the accepting states will be

$$T = \{(q_1, q_2) : \ q_1 \in T_1 \text{ and } q_2 \in T_2\}.$$

Note: The intersection of two DFAs is useful in various practical applications where you need to accept only the strings that satisfy the conditions or rules of both automata

- **Concatenation of two DFA's**: The process is simple

For two machines $M_1(Q_1, \Sigma, q_{0_1}, T_1, \delta_1)$, and $M_2(Q_2, \Sigma, q_{0_2}, T_2, \delta_2)$

1. Connect the final states of the first machine to the start state of the second machine (With ϵ-transitions)
2. Clear T_1, There are no more final states in the first machine
3. Convert ϵ-NFA to DFA

Note: The concatenation of two DFAs has practical uses in many scenarios where the language of interest is the concatenation of two sublanguages. Concatenating two DFAs allows you to recognize strings that can be divided into two parts, where the first part is recognized by one DFA and the second part is recognized by the other.

- **Finding the union of two NFA's**: taking the union of two nondeterministic finite automata (NFAs) involves constructing a new NFA that accepts any string that is accepted by either of the original NFAs. This process can be done by creating a new NFA that combines the two original NFAs.

Given $M_1(Q_1, \Sigma, q_{0_1}, T_1, \delta_1)$, and $M_2(Q_2, \Sigma, q_{0_2}, T_2, \delta_2)$

1. **New start state**: Start by defining a new start state q_0', this state will have ϵ transitions to the start states of both machines.
2. **Define Q', the new set of states**: The new set of states will be the set of all states in M_1, and it will include all the states in M_2, along with the new start state. Thus,

$$Q' = Q_1 \cup Q_2 \cup \{q_0'\}.$$

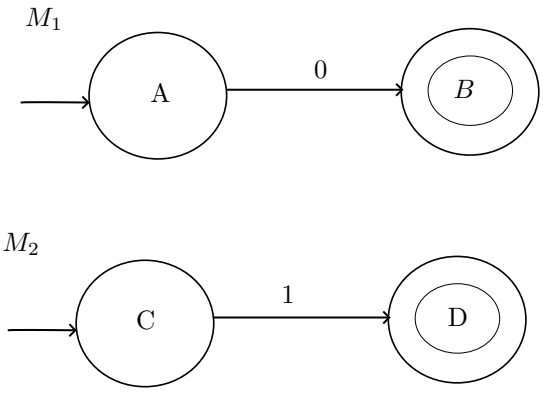
3. **Define the transition function**: The transition function δ' of the new NFA will include:
   - All the transitions of M_1 and M_2
   - Two ϵ transitions from the new start state to the start states of the two original machines q_{0_1} and q_{0_2}. Thus,

$$\delta'(q_0', \epsilon) = \{q_{0_1}, q_{0_2}\}.$$

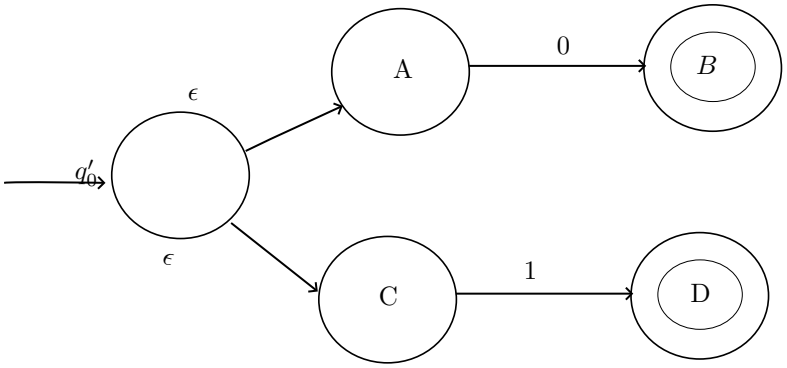
4. **Define the set of accepting states**: The set of accepting states will be

$$T' = T_1 \cup T_2.$$

Example:



$M_1 \cup M_2$ is then



- **Finding the intersection of two NFA's**: For NFAs, intersection is more complex because NFAs are nondeterministic and don't handle intersection naturally. Typically, you convert the NFAs to DFAs and then apply the DFA product construction
- **Concatenation of two NFA's**: The process is the same as with two DFA's (see above), but you don't need to convert to a DFA at the end.

- **Properties of union, intersect, and concatenation for two FA's**: the properties of union, intersection, and concatenation for finite automata (FAs) are directly tied to the properties of regular languages.

  Union of two FA

  - **Closure**: The class of regular languages (those recognized by FA) is closed under union. This means the union of two regular languages is also regular, and there exists an FA that recognizes the union of the languages.
  - **Commutative:** Union is commutative for FA, meaning the order of combining automata does not matter.
  - **Associative:** Union is associative, so it doesn't matter how automata are grouped when performing multiple unions.
  - **Distributive over Intersection** Union distributes over intersection for regular languages, just as with sets.

  Intersection of two FA

  - **Closure:** The class of regular languages is also closed under intersection, meaning there is always an FA (typically constructed as a DFA) that recognizes the intersection of two regular languages.
  - **Commutative:** Intersection is commutative, meaning the order of combining automata doesn't matter.
  - **Associative:** Intersection is associative, so the grouping doesn't matter.
  - **Distributive over Union:** Intersection distributes over union for regular languages, just as with sets.

  Concatenation

  - **Closure:** Regular languages are closed under concatenation.
  - **Associativity:** Concatenation is associative. This means that the way in which you group the automata when performing concatenation doesn't matter.
  - **Identity Element:** The identity element for concatenation is the language that contains only the empty string,

    $$L(A) \cdot \{\epsilon\} = L(A).$$

  - **Distributivity Over Union:** Concatenation distributes over union. This means:

    $$L(A) \cdot (L(B) \cup (L(C)) = L(A) \cdot L(B)) \cup (L(A) \cdot L(C)).$$

  - **Concatenation with the Empty Set:** Concatenating any language with the empty set results in the empty set. This is because there are no strings to concatenate if one of the languages is empty:

  Not commutative

1.2.4 Regular expressions

- **RE**: A RE corresponds to a set of strings; that is, a RE describes a language
- **RE three operations**:
  1. Union (+)
  2. concatenation (xy)
  3. star (zero or more copies)
- **RE special symbols**

$$+ \quad * \quad (\quad).$$

- **Grouping**: The parenthesis are used for grouping,
- **Union**: the plus sign means **union**. Thus, writing

$$0 + 1.$$

  Means zero or one, we refer to + as "or"
- **Concatenation**: We concatenate simply by writing one expression after the other, with no spaces

$$(0 + 1)0.$$

  Is the pair of strings 00 and 10
- **Empty string**: We can also use the empty string ϵ

$$(0 + 1)(0 + \epsilon).$$

  corresponds to 00, 0, 10, and 1
- **Zero or more copies (star)**: Using the start indicates zero or more copies, thus

$$a*.$$

  corresponds to any string of a's: $\{\epsilon, a, aa, aaa, ...\}$
- **More on union**: If you form an RE by the or of two REs, call them R and S, then the resulting language is the union of the languages of R and S.

  Suppose $R = (0 + 1) = \{0, 1\}$, and $S = \{01(0 + 1)\} = \{010, 011\}$, then $R + S = (0 + 1) + (01(0 + 1)) = \{0, 1, 010, 011\}$
- **More on concatenation**: If you form an RE by the or of two REs, call them R and S, then the resulting language consists of all strings that can be formed by taking one string from the language of R and one string from the language of S and concatenating them.

  Suppose $R = (0 + 1) = \{0, 1\}$, and $S = \{01(0 + 1)\} = \{010, 011\}$, then $RS = (0 + 1)01(0 + 1) = \{0010, 0011, 1010, 1011\}$
- **More on star**: If you for man RE by taking the star of an RE R, then the resulting language consists of all strings that can be formed by taking any number of strings from the language of R (they need not be the same and they need not be different), and concatenating them.

Suppose $R = 01(0+1) = \{010, 011\}$, then $R^* = 01(0+1)*\{010, 010010, ..., 011, 011011, ...010011, ...\}$

- **Precedence of the operations**
  1. Star (*)
  2. Concatenation
  3. Union (+)
- **Recursive definition of the kleene star (closure) (L^*)**:
  1. $\epsilon \in L^*$
  2. If $x \in L^*$ and $y \in L$, then $xy \in L^*$

  Base case: The first rule provides a starting point by ensuring that the empty string ϵ is in L^*.

  Recursive step: The second rule allows you to take any string x already in L^* and concatenate it with a string $y \in L$ to produce a new string $xy \in L^*$.

  After using the second rule once to generate a new string $xy \in L^*$, you can apply the rule again by concatenating this new string with another string from L. This recursive process can continue indefinitely, generating all possible strings that can be formed by concatenating zero or more strings from L.
- **Recursive definition of the kleene star (other)**
  1. $L^0 = \{\epsilon\}$ (Start with the empty string, always in the closure)
  2. $L^i = LL^{i-1}$ for $i > 0$ (Start recursively building strings)
  3. $L^* = \bigcup_{i=0}^{\infty} L^i$ (the whole thing)

  Note: We also define the positive closure of L, denoted L^+, as $L^* - \{\epsilon\}$ or

$$L^+ = \bigcup_{i=1}^{\infty} L^i.$$

- **Closure of the empty language**: $\Phi^* = \{\epsilon\}$
- **Regular expression for the empty language**: $\Phi = \varnothing$ is the regular expression for the empty language (empty set)
- **More on language composition operators**: The language composition operators were defined over any language and, in turn, generate new languages. As such, composition operators take any one or two languages from $P(\Sigma^*)$ and can produce any language in $P(\Sigma^*)$.
- **Regular languages (regular sets), regular expression limits**: Although regular expressions are based on language composition operators, their recursive definition (i.e., only regular expressions, therefore only languages defined by regular expressions) limits the languages that they can define.

  Note: Regular expressions cannot produce all languages in $P(\Sigma^*)$.

In fact, the set of languages that regular expressions can define have a special name – they are called regular languages (or sometimes regular sets).

- **Kleene's theorem**: There is an FA for a language if and only if there is an RE for the language
- **Regular expressions order of operations**: From highest to lowest precedence
    1. Parenthesis
    2. Kleene star
    3. Concatenation
4. Union (+ or ||)

- **Properties of regular expressions**:

  Note: Intersection is a operation not defined for regular expressions

  Union

  – **Commutative**:
  $$R_1 \cup R_2 = R_2 \cup R_1$$
  – **Associative**:
  $$(R_1 \cup R_2) \cup R_3 = R_1 \cup (R_2 \cup R_3)$$
  – **Identity Element**:
  $$R_1 \cup \emptyset = R_1$$
  – **Idempotent**:
  $$R_1 \cup R_1 = R_1$$

  2. Concatenation ($\cdot$)

  – **Non-commutative**:
  $$R_1 \cdot R_2 \neq R_2 \cdot R_1$$
  – **Associative**:
  $$(R_1 \cdot R_2) \cdot R_3 = R_1 \cdot (R_2 \cdot R_3)$$
  – **Identity Element**:
  $$R_1 \cdot \epsilon = \epsilon \cdot R_1 = R_1$$
  – **Concatenation with $\emptyset$**:
  $$R_1 \cdot \emptyset = \emptyset \cdot R_1 = \emptyset$$

  Kleene Star ($*$)

  – **Kleene Star of ϵ**:
  $$\epsilon^* = \{\epsilon\}$$
  – **Kleene Star of $\emptyset$**:
  $$\emptyset^* = \{\epsilon\}$$
  – **Idempotent**:
  $$(R^*)^* = R^*$$

Distributive Properties

- **Union over Concatenation**:

$$R_1 \cdot (R_2 \cup R_3) = (R_1 \cdot R_2) \cup (R_1 \cdot R_3)$$

- **Concatenation over Union**:

$$(R_1 \cup R_2) \cdot R_3 = (R_1 \cdot R_3) \cup (R_2 \cdot R_3)$$

- **Language of a RE notation**: $L(RE)$ is the language defined by the regular expression RE, If we have an RE R, then the language $L(R)$ is the language defined by the RE R
- **When a regular expression is the empty set** $\varnothing$: When a regular expression (RE) represents the empty set it means that the RE matches no strings at all, not even the empty string.

  The language is then

$$L(\varnothing) = \Phi.$$

  Where Φ denotes the empty language
- **One or more occurences** RR^*: We denote this by plus instead of star, ie $RR^* = R^+$, but you also must redefine union as $|$ instead of $+$
- **Simplifying regular expressions (Some can also be found above in properties)**:
  - **Concatenation of stars**: $(R^*)^* = R^*$
  - **Concatenation of Repeated Expressions**: $R^*R^* = R^*$
  - **Idempotence of Union**: $R \mid R = R$
  - **Empty Set in Union and Concatenation:** $R \mid \varnothing = R$, $R\varnothing = \varnothing$
  - **Empty string in concatenation**: $\epsilon R = R\epsilon = R$
  - **Union with the kleene star**: $R^* \mid R = R^*$
  - **Distributive Property:** $R_1(R_2 \mid R_3) = R_1R_2 \mid R_1R_3$
  - **Absorption:** $R \mid (RR^*) = RR^* = R^+$
- **The RE operators with the empty language** Φ:
  1. $\varnothing r = r\varnothing = \varnothing\varnothing = \varnothing$ for any regular expression r
  2. $r + \varnothing = \varnothing + r = r$
  3. $\varnothing + \varnothing = \varnothing$
  4. $\varnothing^* = \{\epsilon\}$

  These cases can also be represented with language notation

  1. $\Phi L = L\Phi = \Phi\Phi = \Phi \ \forall L$
  2. $L + \Phi = \Phi + L = L$
  3. $\Phi + \Phi = \Phi$
  4. $\Phi^* = \{\epsilon\}$
- **Convert RE to NFA-**ϵ: The conversion algorithm starts by defining an NFA with ϵ-moves for each of the three base cases from the recursive definition of a regular expressions over an alphabet Σ
  1. $\varnothing$ is a regular expression and denotes the empty set (i.e., the empty language Φ)
  2. ϵ is a regular expression and denotes the set $\{\epsilon\}$
  3. For each symbol $x \in \Sigma$, x is a regular expression and denotes the set $\{x\}$.

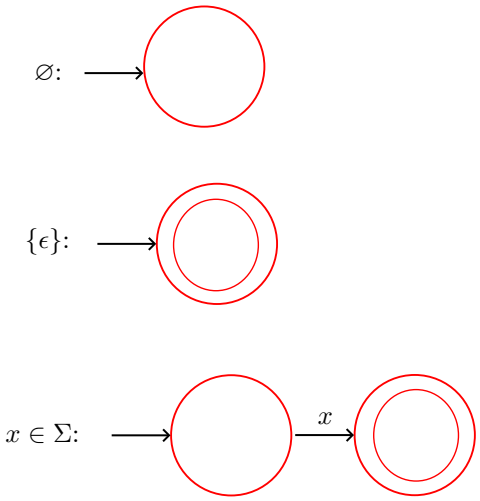
Conditions on the NFAs with ϵ-moves for This Algorithm

1. There must be exactly one accepting state.
2. No transitions (not even ϵ-moves) may leave the one accepting state.

Note: If faced with an NFA with ϵ-moves that has more than one accepting state and/or accepting states with transitions leaving it then simply modify the NFA with ϵ-moves by

1. Adding a new accepting state.
2. Add an ϵ-move from each of the original accepting states to the newly added accepting state.
3. Convert all of the original accepting states to non-accepting states.

The three base cases have the following nfa that satisfy the above criteria



We use those NFAs as the basic building blocks to iteratively build more complex NFA's with ϵ-moves (all the while honoring the accepting state conditions for this algorithm) as we apply the recursive part of the regular expression definition. Recall:

If r and s are regular expressions denoting the sets R and S, respectively, then

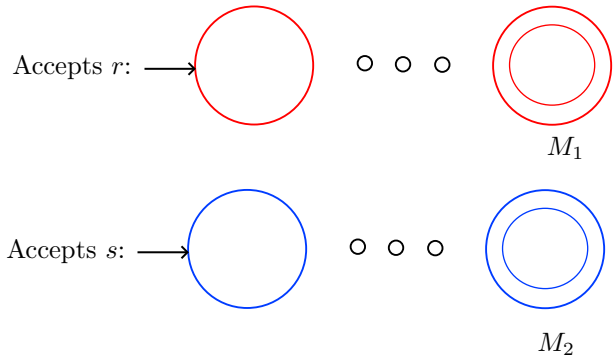
1. $r + s$ is a regular expression denoting the set $R + S$, (i.e,. union of languages),
2. rs is a regular expression denoting the set RS (i.e., concatenating languages), and
3. $r*$ is a regular expression denoting the set R^* (i.e., Kleene closure of a language).

Once we define an NFA with ϵ-moves for each of the base cases (which we have done) then when we address each recursive part of the definition (e.g., union above)

1. We may assume that there already exists NFAs with ϵ-moves for each of the regular expressions r and s (and that each also satisfies the acceptance state conditions of this algorithm) and
2. Then our job is to use those NFAs with ϵ-moves to create a new NFA with ϵ-moves that accepts $r + s$ and that also satisfies the acceptance state conditions of this algorithm.

The algorithm:

- **Handling union:** We start by assuming there already exists NFAs with ϵ-moves M_1 and M_2 that accept regular expressions r and s, respectively, and that both M_1 and M_2 satisfy the acceptance state conditions (i.e., one accepting state, no exit) of this algorithm.



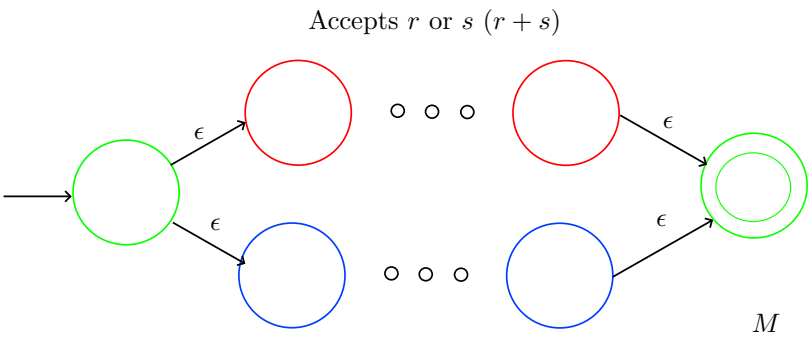
Note: The details of the machine arn't important here, all we know is the machine has a start, does whatever else it needs to (repesented by the elipsis), and then accepts strings represented by r in the top machine and s in the bottom

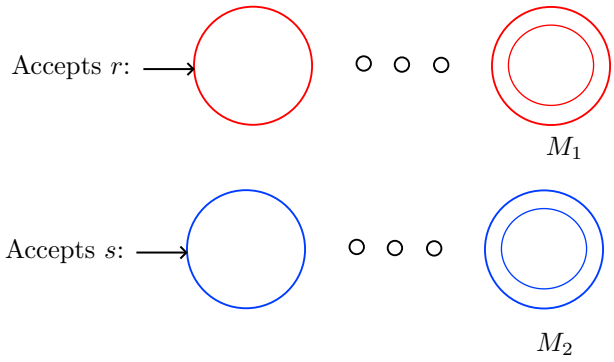
We then use these machines M_1 and M_2 to create a new machine M that accepts $r + s$

So what did we do here

1. Create new start state and add ϵ-moves to the original start states
2. Create new accepting state and add emoves from all the original accepting states.
3. Change the original accepting states to non-accepting states.

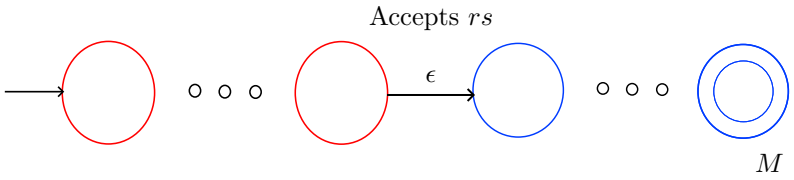
- **Handle Concatenation**: We again start by assuming there already exists NFAs with ϵ-moves M_1 and M_2 that accept regular expressions r and s, respectively, and that both M_1 and M_2 satisfy the acceptance state conditions (i.e., one accepting state, no exit) of this algorithm.



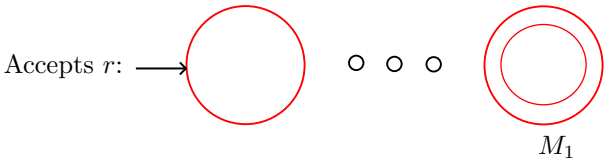


We use M_1 and M_2 to construct new NFA with ϵ-move M that accepts rs.

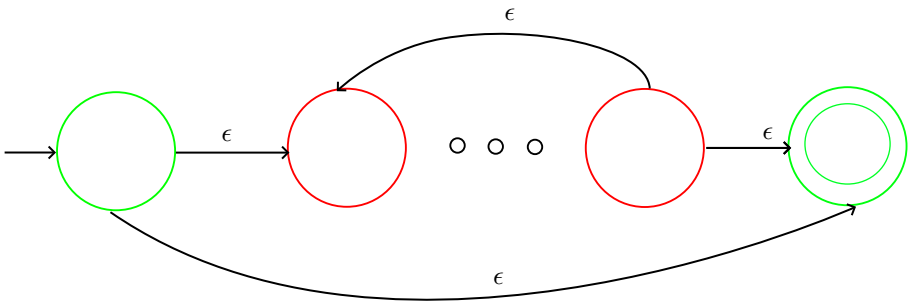
1. Add an ϵ-move from M_1's accepting state to M_2's start state.
2. Change M_1's accepting state to a nonaccepting state.



– **Handle Kleene closure**: We again start by assuming there already exists an NFA with ϵ-moves M_1 that accepts regular expressions r and that satisfies the acceptance state conditions (i.e., one accepting state, no exit) of this algorithm.



We use M_1 to construct new NFA with ϵ-move M that accepts r^*

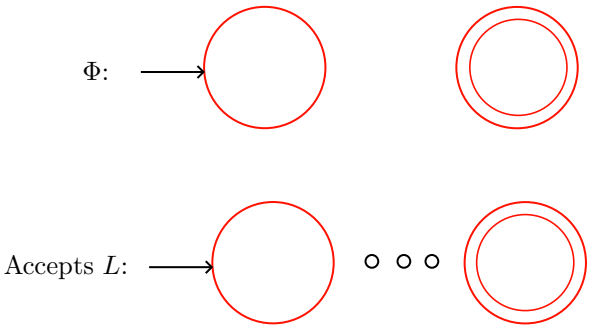


1. Create new start and accepting states.
2. Add ϵ-move from new start to M_1 start, M_1 accepting to new accepting, and new start to new accepting.
3. Add ϵ-move from M_1 accepting to M_1 start.
4. Change M_1's accepting state to a non accepting state.

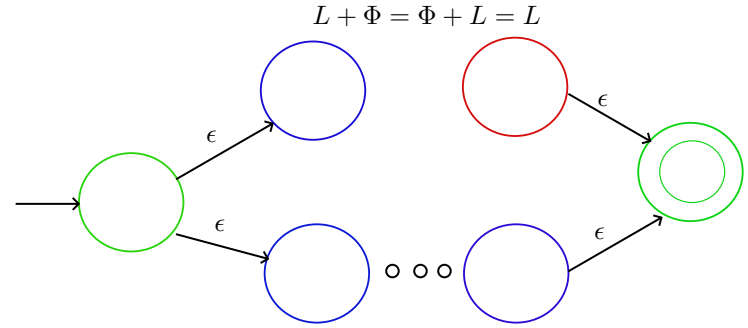
- **RE to NFA conversion: Special cases**: Recall the special case with regular expressions, the empty language Φ – and how it behaved with the three regular expression operators;

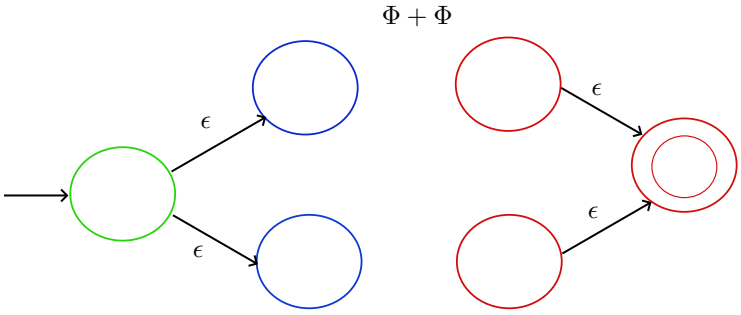
1. $\Phi L = L\Phi = \Phi\Phi = \Phi \ \forall L$
2. $L + \Phi = \Phi + L = L$
3. $\Phi + \Phi = \Phi$
4. $\Phi^* = \{\epsilon\}$

We can now check these operations using the algorithm with Φ. First, we define the base case NFA's

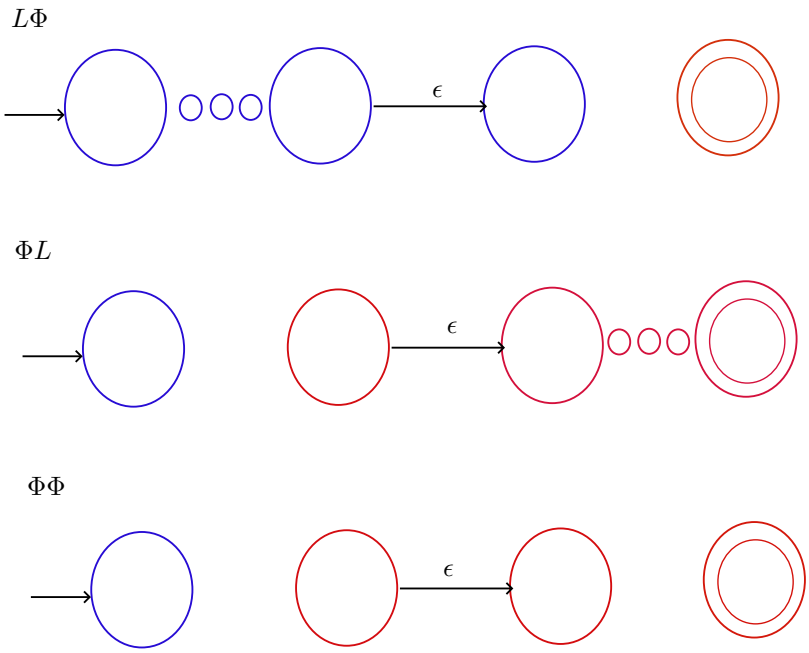


1. Confirming $L + \Phi = \Phi + L = L$ and $\Phi + \Phi = \Phi$:

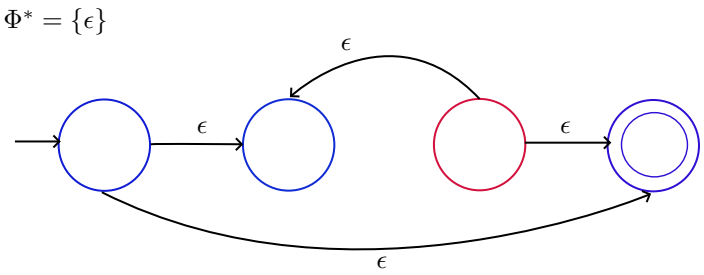




2. Confirming $\Phi L = L\Phi = \Phi\Phi = \Phi$



3. Confirming $\Phi^* = \{\epsilon\}$



- **Convert NFA-ϵ to RE**: If necessary, first modify the NFA with ϵ-moves to satisfy these two conditions (i.e., conditions of this algorithm, not requirements of all NFA's with ϵ-moves);

  a) No transition may enter the start state – not even a loop.

  b) If there exists even one accepting state, then there can be only one accepting state and no transition may leave that accepting state.

Note: If there was no accepting state then do not create one, stop the algorithm, and output the regular expression $\varnothing$ to denote the empty language Φ.

Then we start the algorithm. We need to convert the label on each transition to a regular expression until there is only two states, a start state and an accepting state, and all transitions between these two states are regular expressions. The final regular expression will be the union of all transitions.

1. While there are more "middle" states (i.e., states that are neither the start state or accepting state)
   (i) Select one of the remaining middle states.
   (ii) Bypass the middle state creating new transitions as necessary annotating each new transition with a regular expression.
   (iii) Remove the bypassed middle state.
2. If there are any transitions between the start and accepting state, then the regular expression that accepts the same language as the original FA is the the union (i.e., "+") of the regular expressions of all the transitions.

   If there are no transitions between the start and accepting state, then output the regular expression $\varnothing$ to denote the empty language Φ.

Recall the following from the definition of regular expressions;

(i) [base case]: L is a regular expression and denotes the set $\{L\}$

(ii) [base case]: For each symbol $x \in \Sigma$, x is a regular expression and denotes the set $\{x\}$

(iii) [recursive case]: $r + s$ is a regular expression denoting the set $R + S$, (i.e,. union of languages).

We use those to covert every ϵ to a Λ, every symbol $x \in \Sigma$ to a regular expression of the same symbol, and every case comma-separated transition label to a regular expression with "+".

After ensuring the FA abides by the start and end state conditions, and we convert every transition to the simple regular expressions, we begin eliminating states.

Some notes:

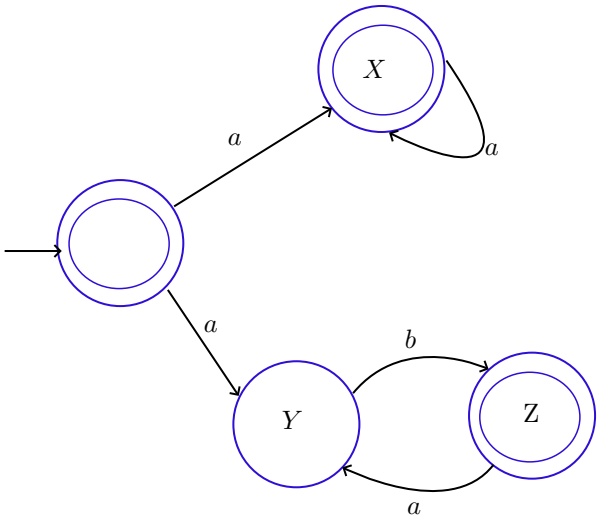
(a) Regarding the middle states (states that are neither the start nor accepting state), it doesn't matter in which order we choose to eliminate them.

(b) For each state we are eliminating, we count the number of incoming and outgoing transitions (loops don't add to the count but we still need to take care of them with the regular expressions), there will be a new regular expression transition for all combinations of outgoing and incoming transitions. Ie pick a state to eliminate, then

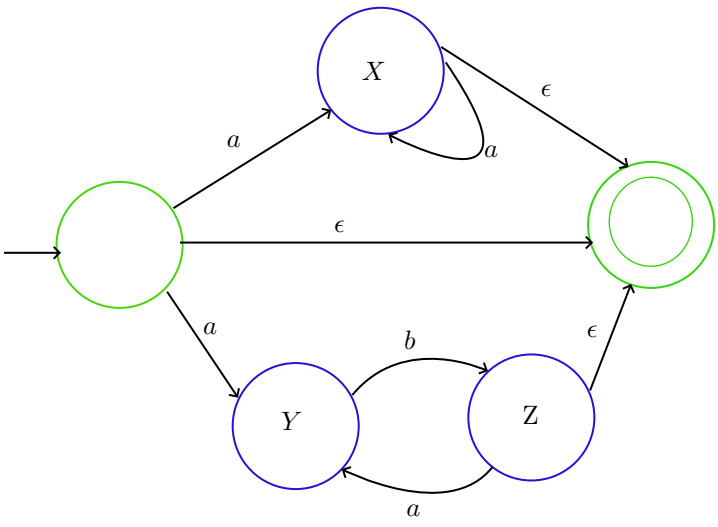
$$\text{New RE transitions} = N(\text{outgoing}) \times N(\text{incoming}).$$

Not including the loops

Example: Consider the NFA-ϵ

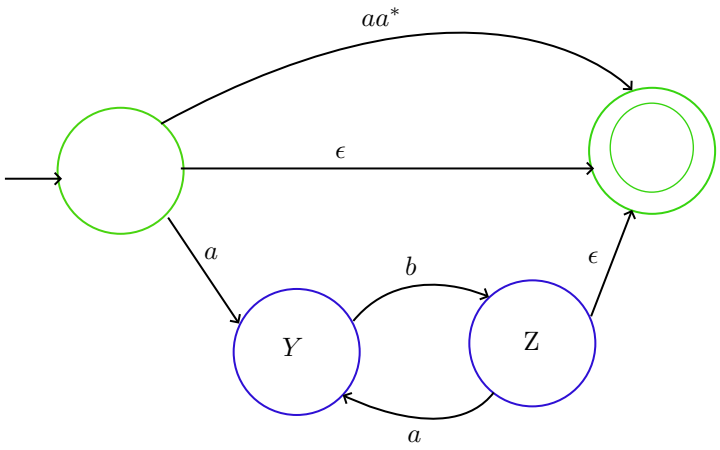


Before we begin eliminating states, we see that this FA does not obey the two constraints described above. So, we create a new accepting state such that there is only one accepting state. Each old accepting state has ϵ transitions to this new accepting state. This FA has no incoming transitions to the start state so nothing to fix there.

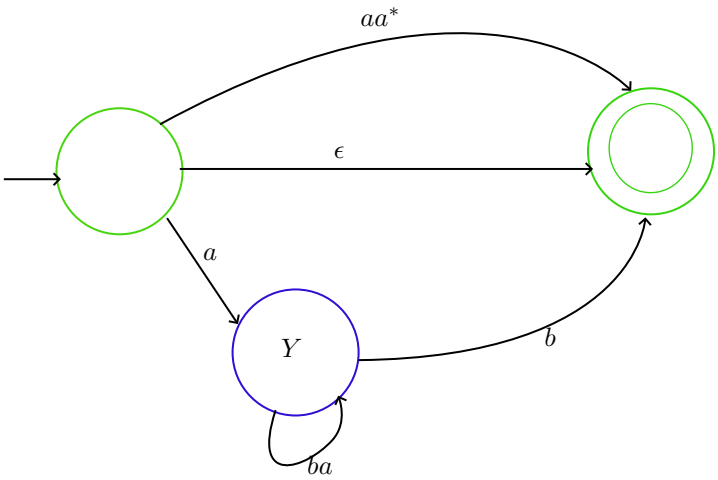


Now, we start eliminating states one at a time. We recall that it does not matter the order in which we eliminate them.

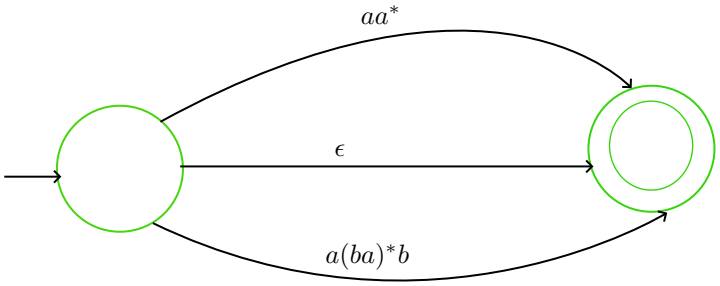
We start by eliminating state X. There is one incoming transition and one outgoing transition. Thus, there is $1 \times 1 = 1$ new transition. To get from the start state, through X, to the accepting state, the regular expression is $aa^*\Lambda = aa^*$. Thus,



Next, we choose to eliminate state Z. We have one incoming and two outgoing transitionss. Thus, we have $1 \times 2 = 2$ new regular expression transitions. To get from Y through Z to the accepting state, the RE is $b\epsilon = b$. To go from Y through Z back to Y, we have ba. Thus



Finally, we eliminate Y. To get from the start state, through Y, to the accepting state, the regular expression is $a(ba)^*b$. Thus,



The final regular expression is then

$$aa^* + \epsilon + a(ba)^*b.$$

■

1.2.5 Properties of regular languages

- **Recall: Regular language**: Recall that we call a language a regular language if, and only if, the language is accepted by some regular expression.
- **Recall: Recursive definition of regular expressions**: Recall also our recursive definition of regular expressions over some alphabet Σ

  Let Σ be an alphabet. The regular expressions over Σ and the sets (i.e., languages) that they denote are defined recursively as follows:

  Base cases:

  1. $\varnothing$ is a regular expression and denotes the empty set (i.e., the empty language Φ).
  2. L is a regular expression and denotes the set $\{L\}$.
  3. For each symbol $x \in \Sigma$, x is a regular expression and denotes the set $\{x\}$.

  Recursion: If r and s are regular expressions denoting the sets R and S, respectively, then

  1. $r + s$ is a regular expression denoting the set $R + S$, (i.e,. union of languages),
  2. rs is a regular expression denoting the set RS (i.e., concatenating languages), and
  3. r^* is a regular expression denoting the set R^* (i.e., Kleene closure of a language).
- **Relationship between regular languages and the set of all possible languages** $\mathcal{P}(\Sigma^*)$: We know that the set of regular languages must be a subset of $\mathcal{P}(\Sigma^*)$ (it may be equal to $\mathcal{P}(\Sigma^*)$), and so, we can start by creating that subset.

  We can then start noting the languages that we know are regular based on the base cases from the definition of regular expressions and for some given alphabet, say $\Sigma = \{a, b\}$.

  The recursion from the definition tells us that we can take any language, or pair of languages, from the already existing set of regular languages, and use it/them to create a new language this is also a regular language. And the recursion may be applied over and over (i.e., without limit), always taking only regular languages that have been previously created (original base case languages or languages subsequently derived), to create new languages. (i.e., the set of regular languages is infinite).

- **Closure of regular languages and their operations**: Expanding on the item above, formally, we say that the set of regular languages is **closed** under the language composition operations union, concatentation, and Kleene star
- **Closure of complement and intersection**: In addition to the three operations that come from the definition of regular expressions (i.e., union, concatenation, and Kleene star), the set of regular languages is also closed under;
  - Complement
  - Intersection
- **Complement of a Language**: Recall that every language is a set of strings – empty, finite, or infinite – that is always a subset of Σ^*

That is,

- For any alphabet Σ we get Σ^*
- We define some language L over that alphabet Σ
- Then L is a subset of Σ^*; $L \subseteq \Sigma^*$

We define a new language, the complement of L, denoted L' as the set of strings that are not in the language L (i.e., $L' = \Sigma^* - L$).

- **Proof: Regular languages are closed under complement**: We assert that if you take the complement of a regular language, the resulting language is then regular

  Proof:

  1. A regular language is one that is accepted by some regular expression.
  2. By Kleene's Theorem we know that any regular expression can be converted to a NFA with ϵ-moves that accepts the same language, and vice versa.
  3. We can convert any NFA with ϵ-moves to a DFA that accepts the same language and every DFA is, by definition, an NFA with ϵ-moves.

  So L is a regular language iff it is accepted by some DFA...

  Consider some DFA $M(Q, \Sigma, q_0, T, \delta)$ that accepts regular language L.

  We construct a new DFA $M'(Q, \Sigma, q_0, T', \delta)$ from M by defining $T' = Q - T$, that is, every accepting state in M becomes a non- accepting state in M', and vice versa.

  Since M' accepts L' and M' is a DFA, then L' is a regular language.

  Since M was chosen arbitrarily, the complement of any regular language is also a regular language

  ■

  Note: Note: The proof must be based on DFA's ... would not have worked for non-deterministic FA's.

- **Intersection of Languages**: Given any two languages, L_1 and L_2, over some alphabet Σ we can create a new language L that is the intersection of the two sets L_1 and L_2.

  That is,

  $$L = L_1 \cap L_2 = \{x : \ x \in L_1 \wedge x \in L_2\}.$$

- **Proof: Regular languages are closed under intersection**: This means that when you take the intersection of any two regular languages, the resulting language is always regular.

  Assume you have two regular languages L_1 and L_2.

We define a new language L, which is the intersection of L_1 and L_2, namely

$$L = \{x \mid x \in L_1 \wedge x \in L_2\},$$

where "$\wedge$" means "logical and."

Demorgan's law:

$$(a \wedge b) = \sim (\sim a \vee \sim b).$$

To prove that $L = L_1 \cap L_2$ is regular, we use the fact that regular languages are closed under complement and union. This leads us to apply De Morgan's Law:

$$L_1 \cap L_2 = \sim (\sim L_1 \cup \sim L_2).$$

Here, $\sim L_1$ and $\sim L_2$ represent the complements of L_1 and L_2, respectively. Since L_1 and L_2 are regular, their complements $\sim L_1$ and $\sim L_2$ are also regular (because regular languages are closed under complement).

Next, since regular languages are closed under union, the language $\sim L_1 \cup \sim L_2$ is also regular.

Finally, the complement of $\sim L_1 \cup \sim L_2$, i.e., $(\sim L_1 \cup \sim L_2)'$, is also regular because regular languages are closed under complement. But $(\sim L_1 \cup \sim L_2)'$ is precisely $L_1 \cap L_2$.

Thus, $L = L_1 \cap L_2$ is regular, as required.

$$L_1 \cap L_2 = \sim (\sim L_1 \cup \sim L_2) = \{x \mid x \in L_1 \wedge x \in L_2\}.$$

■

- Addings to regular expression recursive: Thus, we add complement and intersection to recursive cases when building regular languages defined in $\mathcal{P}(\Sigma^*)$

  Recursion: If r and s are regular expressions denoting the sets R and S, respectively, then

  1. $r + s$ is a regular expression denoting the set $R + S$, (i.e,. union of languages),
  2. rs is a regular expression denoting the set RS (i.e., concatenating languages), and
  3. r^* is a regular expression denoting the set R^* (i.e., Kleene closure of a language).

  or by taking the complement or intersection of one or two previously created regular languages.

1.2.6 Applications of finite automata

- **FA with output**: Thus far we have only considered finite automata as language acceptors (i.e., defining some regular language).

  Finite automata can also serve another purpose. They can be used to process an input string to produce some output.

  When used in this way, the finite automata do not define a language. In fact, they do not have any accepting states.

  Their sole purpose is to process input to generate output

- **Moore machine**: A Moore machine is a deterministic finite automaton and is defined by a 6-tuple $(Q, \Sigma, q_0, \delta, \Gamma, O)$, where

  - Γ is an alphabet of output symbols.
  - O is the output function: $O: \ Q \to \Gamma$

  Each state is annotated with an output symbol. Output "printed" upon entering state

  Note: start state's output symbol always "printed", even on empty string ϵ

- **Mealy Machine**: A Mealy machine is a deterministic finite automaton and is defined by a 5-tuple $(Q, \Sigma, q_0, \delta, \Gamma)$, where

  - Γ is an alphabet of output symbols

  Each transition is annotated with an output symbol Output "printed" when traversing edge

  Note: The number of input and output symbols are always identical.

1.3 Nonregular languages

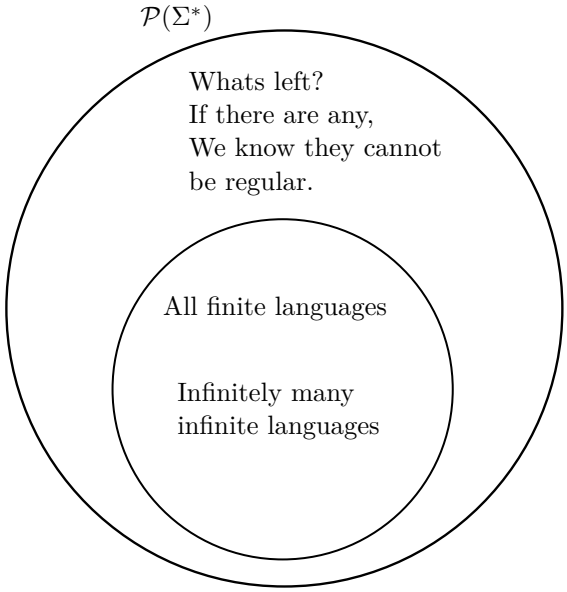
- **Intro to nonregular languages**: Suppose we have some alphabet Σ, we can then find Σ^*, which is the language consisting of all possible strings (including the empty string ϵ) using the symbols from Σ. The we can take the power set of Σ^* to get the set of all possible subsets of Σ^*. The alphabet Σ is always finite, if Σ is nonempty, then Σ^* is always infinite. (If $\Sigma = \varnothing$, $\Sigma^* = \{\epsilon\}$). Since Σ^* is infinite, $\mathcal{P}(\Sigma^*)$ is also infinite. If $\Sigma = \varnothing \implies \Sigma^* = \{\epsilon\} \to \mathcal{P}(\Sigma^*) = \{\{\epsilon\}, \varnothing\}$. Note that for any set S, $\varnothing \subset S$.

  What types of languages are contained in $\mathcal{P}(\Sigma^*)$?

  1. If a language is finite, it is always regular. This is a consequence of Kleene's theorem, which asserts that if a regular expression expresses a language, the language is regular. If a language is finite, we can **always** make a regular expression for it. Simply take all strings in the language, and take the union. For example, if $L = \{a, ab, abc\}$, then a regular expression for the language is simply $a + ab + abc$ and the language is therefore regular. ■
  2. We also know that there are infinitely many regular expressions than can express infinitely many languages from $\mathcal{P}(\Sigma^*)$. This is a consequence of the recursive definition of regular expressions. Therefore there are infinitely many regular expressions to describe infinite languages.

  So, $\mathcal{P}(\Sigma^*)$ is the infinite set of all possible languages, finite or infinite, that can be generated from a given language Σ. All Finite languages from this set are regular, and there are also infinitely many infinite languages from this set that are regular. But, are there any languages that are *not* regular? Infinite languages that cannot be expressed as a regular expressions?

  It may seem unlikely that nonregular languages exist at all. To claim that a language is nonregular one must prove that no regular expression or FA that accepts the language exists



- **The Pumping Lemma**: Before we continue our conversation about nonregular languages we first look at a lemma about regular languages

  Lemma. Let L be an infinite regular language, then

  $$\exists\ x, y, z \in \Sigma^* \mid y \neq \epsilon \wedge xy^n z \in L\ \forall\ n > 0.$$

  In other words, all regular languages have the property that, there exists some strings x, y, z, where $y \neq \epsilon$ such that all the strings of the form $xy^n z \in L$ for all $n > 0$

  This means that for some x, y, z, we can infintely pump in more copies of y, and the string remains in the language. For example,

  $$xyz \quad xyyz \quad xyyyz \quad ....$$

- **Pumping Lemma Proof**: Assume you have some regular language L with infinitely many strings. Because L is regular, there must exists some DFA $M(Q, \Sigma, q_0, T, \delta)$ that accepts L. Since FA's are required to have finite states, let's say it has n states.

  Because L is infinite, we can always find strings that have length greater than n. Ie $|w| \geq n$, where $w \in L$. Because w has at least as many characters as there are states in M, as we process w with M we know that one or more of the states in M must be revisited. We know that as we process w, it must traverse what we call the *circuit*. which is a sequence of one or more edges that contains at least one state that is visited more than once.

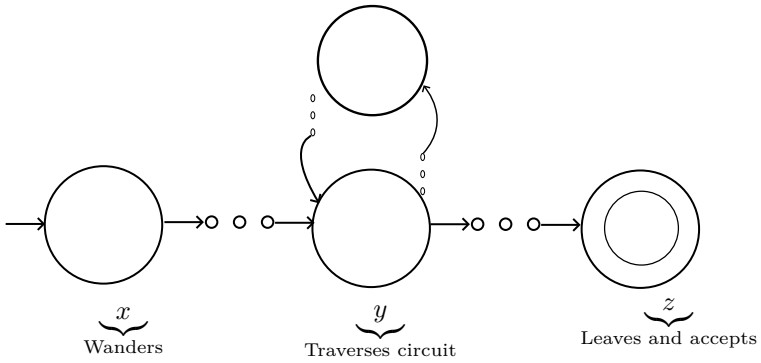
  Given that circuit and because we also selected $w \in L$, we note that we can modify w to create a new word w' pumping into w as many symbols as are needed and in just the right location in w that would cause M to traverse the circuit one more time than it did when processing w. We note that the new word $w' \in L$.

  In fact, given DFA M with n states and string $w \in L$, $|w| \geqslant n$, we can generate an infinite supply of new words by simply pumping into w, and t the right location, more and more copies of the string that causes M to traverse the circuit. We note that the new words created in this way are all in L.

  This gives us the existence of the x, y, z strings the Pumping Lemma needs as follows:

  - x is the prefix of w that is consumed by M as the DFA wanders up to the circuit (x may be Λ and this sequence of states may be empty).
  - y is the substring of w that is consumed by M as the DFA traverses the circuit (since the circuit must visit at least one state more than once, it must consume some symbols, and so y cannot be Λ).
  - z is the suffix of w that is consumed by M as the DFA leaves the circuit and goes to an accepting state (z may be Λ and this sequence of states may be empty).

  Therefore, L must contain all the strings of the form $xy^n z$ for all $n > 0$.



- **The value of the pumping lemma**: The Pumping Lemma tells us something that is true of all regular infinite languages.

  The real value of The Pumping Lemma is to prove that some infinite language is nonregular, that is a language **cannot** be regular if it does not satisfy the claim in the pumping lemma. Thus, we can prove that a language is non-regular by contradiction.

  1. Assume the language is regular
  2. You show that it is not possible to find strings x, y, z that satisfy the claim in the pumping lemma.
  3. We conclude that our assumption that the original language is a regular language must be false, and therefore, it must be nonregular.

- **Note on the pumping lemma**: The pumping lemma gaurantees that for a large enough $w \in L$ ($|w| \geqslant n$), where n is the number of states in the assumed machine) we can find an x, y, z such that $y \neq \epsilon$ and $xy^n z \in L \ \forall \ n > 0$. If the word you select is greater than or equal to n, and no such y holds, then the language must be nonregular

  The condition is $\geqslant n$, because, for a machine with n states, there are $n - 1$ edges that must be traversed to reach the end. If a machine has four states, then there are $4 - 1 = 3$ edges to reach the end.

- **The pumping language with length: Background**: In the case that you find an x, y, z such that $y \neq \epsilon$ and $xy^n z \in L \ \forall \ n > 0$. This does **not** mean the language must be regular. We know that all regular languages do have this property, but that does not mean that simply possessing this property makes the language regular. In logic theory

$$a \to b \not\iff b \to a.$$

  In words, if a implies b, b does not imply a

Thus, the pumping lemma described above is sometimes not enough, and we may need something more powerful.

- **The pumping lemma with length**: Let L be an infinite language accepted by a FA with n states. Then, for all $w \in L$, where $|w| \geqslant n$, there exists some three strings x, y, z such that $w = xyz$, $y \neq \epsilon$, $|xy| \leqslant n$, and all the strings of the form $xy^i z \in L$ for all $i > 0$

  The Pumping Lemma With Length adds that you must always be able to find x, y, z in all sufficiently long words $w \in L$ This means:

  - For each word in L that has length greater than n, where n is the number of states in the assumed machine, there must be a composition xyz, where $x, y, z \in \Sigma^*$, $y \neq \epsilon$, and $|xy| \leqslant n$. The x, y, z need not be the same for every word in L with length greater than n, but a pair needs to exist for each word.
  - Furthermore, for each word w, where $|w| \geqslant n$, that has x, y, z such that $w = xyz$. That same x, y, z needs to have the property $xy^i z \in L \ \forall \ i > 0$.
  - Thus, to show that a language is not regular, we only need to show that such an x, y, z does not exist for a single word. If we choose a word and x, y, z exists, we need to keep looking.

  If words keep leading to valid x, y, z at some point we need to stop looking and start trying to prove that the language is actually regular. Whether by creating an RE or an FA. Keep in mind there will be infinite words to check.

1.3.1 Pumping lemma examples

- **Pumping lemma example**: Use the pumping lemma to show that the language $L = a^k b^k \ \forall \ k > 0$ over $\Sigma = \{a, b\}$ is nonregular

  Suppose that L is regular, then we should be able to find some x, y, z, where $y \neq \epsilon$ such that $xy^n z \in L \ \forall \ n > 0$. For simplicity, let's first examine the possible choices for y

  1. $y = a^\ell$ or $y = b^\ell$ for some $\ell > 0$
  2. $y = a^\ell b^\lambda$ for some $\ell, \lambda > 0$

  In case one, pumping would lead to an imbalance in the number of a's or b's. In case two, pumping would lead to more than one ab boundary.

  Thus, no such x, y, z exists and the language is nonregular ■

- **Pumping lemma example:** Show that the language $L = \{a^t : \ t \in \mathbb{P}\}$ over $\Sigma = \{a\}$ is nonregular.

  Suppose L is regular. Then we will find strings x, y, z such that $y \neq \epsilon$ and $xy^n z \in L \ \forall \ n > 0$.

  The only choice of y in this case is $y = a^r, \ r > 0$.

  First, define $w = a^t = xyz, \ t \in \mathbb{P}$. That is, since w is a member of L, we can partition it into the form $a^t = xyz$. Furthermore, $xy^n z \in L \ \forall \ n > 0$. Since this needs to hold **for all** $n > 0$, showing that it doesn't work for a single $n > 0$ breaks the argument. Let $n = t + 1$. This leads us to some algebreic manipulations

  $$\begin{aligned} xy^n z &= xy^{t+1} z = xyy^t z \\ &= xyzy^t \quad \text{(Everything is } a\text{, we can commute)} \\ &= a^t y^t \\ &= a^t (a^r)^t \\ &= a^{t+rt} \\ &= a^{t(1+r)}. \end{aligned}$$

  This next assertion is one of number theory. We assert that if $t \in \mathbb{P}$, $t(r+1) \notin \mathbb{P}$ (Since $r > 0$, $r + 1 > 0$).

  Since we only have one choice of y in this case, and we showed that it does not hold when $n = t + 1$, the language must be irregular. ■

- **Using the Pumping Lemma With Length: Palindrome example**: Prove that the language *Palindrome* over $\Sigma = \{a, b\}$ is a nonregular language.

  Assume there exists an FA with n states that accepts palindrome. Consider $w = a^{n+1} b a^{n+1} \in$ Palindrome

Since we assume L is regular, then for $w = a^{n+1}ba^{n+1}$, which is clearly a palindrome with length $\geqslant n$, must have the property $w = xyz$, for some $x, y, z \in \Sigma^*$, where $y \neq \epsilon$, and $|xy| \leqslant n$. Thus, y must be contained within a^{n+1}, which implies xy must be contained within a^{n+1}. This leads to the conclusion that pumping more copies of y will lead to an a imbalance to the left of the b, which is **not** a palindrome. Thus, the language is nonregular. ■.

- **Pumping lemma with length example**: Show with the pumping lemma that $L = a^n b^m c^{n+m},\ \forall\ m, n > 0$ is a nonregular language

  The pumping lemma states that for an infinite regular language that has an FA with k states, then

  $$\forall w \in L,\ |w| \geq n,\ \exists\ x, y, z \in \Sigma^* \mid w = xyz,\ |xy| \leq n \wedge xy^i z \in L\ \forall\ i > 0.$$

  Let $m, n = k$, then $w = a^k b^k c^{2k}$. Since $|xy| \leq n$, y must be $a^r,\ 0 < r \leq k$. This implies $w = a^r a^{k-r} b^k c^{2k}$. Furthermore, $w' = a^{ir} a^{k-r} b^k c^{2k} \in L\ \forall\ i > 0$. If $i = 2$, $w' = a^{2r} a^{k-r} b^k c^{2k} = a^{k+r} b^k c^{2k}$. For w' to be in L, $2k$ must equal $k + r + k$. Since $2k \neq 2k + r$, we have a contradiction. Thus, pumping more copies of y not satisfy $a^n b^m c^{n+m}$, as the number of c's would not equal the number of a's plus the number of b's ■

- **Pumping lemma with length example**: Over the alphabet $\Sigma = \{a, b, c\}$, show that the language that houses all strings that are not palindromes is nonregular.

  Assume the language has an FA with k states, then $\forall\ w \in L, |w| \leqslant k,\ \exists\ x, y, z \in \Sigma^* \mid w = xyz, |xy| \leq k\ \wedge\ xy^i z \in L\ \forall\ i > 0$.

  An easy way to prove this is by showing that $L =$ Palindrome is nonregular, which is much simpler. Since regular languages are closed under complement, the complement of a regular language must be regular. Thus, the complement of a nonregular language must be nonregular.

  Assume $L =$ palindrome is infinite and regular defined by an FA with k states.

  Let $w = a^k b^k a^k$, then $y = a^r,\ 0 < r \leqslant k$. Then $w = a^r a^{k-r} b^k a^k$ and $w' = a^{ir} a^{k-r} b^k a^k \in L\ \forall\ i > 0$. If $i = 2$, $w' = a^{2r} a^{k-r} b^k a^k = a^{k-r+2r} b^k a^k = a^{k+r} b^k a^k$. Since $r > 0,\ k + r > k \implies a^{k+r} > a^k$ and w' is not a palindrome. Thus, we have a contradiction and L must be nonregular. Since L is nonregular, L' must be nonregular. Therefore, the language of all strings that are not palindromes is nonregular. ■

- **Pumping lemma with length example**: Show that the language $a^t,\ t \in \mathbb{P}$ over the alphabet $\Sigma = \{a\}$ is nonregular.

  Assume the language is regular defined by a FA with n states. Choose $w = a^\ell,\ \ell \in \mathbb{P},\ \ell \geqslant n$. Then, y must be $a^r,\ r > 0$. From this, $w = xyz = a^r a^{\ell - r}$, and $w' = xy^i z \in L\ \forall\ i > 0$. Thus, $w' = a^{ir} a^{\ell - r} \in L\ \forall\ i > 0$. If we can show that some selection of i yields a $w' \notin L$, then we have a contradiction and the language must be nonregular.

  Choose $i = \ell + 1$, which implies $w' = a^{(\ell+1)r} a^{\ell - r} = a^{\ell - r + \ell r + r} = a^{\ell + \ell r} = a^{\ell(1+r)}$. Since $\ell \in \mathbb{P}$, and $r > 0$, $\ell(1 + r)$ cannot be prime. Thus, we have a contradiction and the assumption does not hold for $L = a^t,\ t \in \mathbb{P}$.

1.4 Context free grammers

DSA

2.1 C++ Stuff

2.1.1 Type declarations

- **Discern any type**: Some rules,

  1. Start with the variable name, we read from inside to out
  2. const, %, *, and basic types go on the left
  3. const refers to what is immediately on the left (except for `const int*`), but the standard form of this is actually `int const*`. Thus, the exception to this is const is at the very left, then it refers to what is immediately right.
  4. arrays and functions go on the right, function args are type declaration subproblems

  The Algorithm:

  - Start with the variable name, or the implied name position
  - Read right until end or)
  - Read left until end or (
  - If something still left to read, move out one level of parenthesis and go to 2, else done.

  Thus, using parenthesis allows us to change direction, this will come in handy.

  Examples:

  - a is an int $\implies$ `int a`
  - a is a pointer to an int $\implies$ `int * a`
  - a is a pointer to a constant int $\implies$ `int const * a` (also `const int * a`)
  - a is a constant pointer to an int $\implies$ `int * const a`
  - a is a constant pointer to a constant int $\implies$ `int const * const a` (also `const int * const a`)
  - a is an array of 5 ints $\implies$ `int a[5]`
  - a is an array of 5 pointers to constant ints $\implies$ `int const * a[5]`
  - a is a pointer to an array of 5 constant ints $\implies$ `int const (* a)[5]`

- **Multi dimensional arrays (matrices)**: Think of multi-dimensional arrays as arrays of arrays. More indicative of what's happening internally. `float dat [3][4];` can be read as: "dat is an array of 3 arrays of 4 floats" (Using the algorithm from above).

  Examples:

  - `arg1` is a reference to an array of 25 constant pointers to arrays of 8 strings. $\implies$ `string (* const (& arg1)[25])[8]`

Note: Notice how we use parenthesis to change direction

- **Function Pointers**: Pointers point to bytes, which can be interpreted different ways. Pointers can point to bytes that can be interpreted as code, i.e. a function pointer.

  Examples:

  – f is a pointer to a function which takes an int and returns void. $\implies$ `void (* f) (int)`

2.1.2 G++

- **Compliation and linking**: Compilers turn source code into executable code.
  - **Source code → object code (Compilation)**: Object code is almost executable. It contains pieces that it provides to other objects, and holes to be filled in. It is a slow process
  - **Object code → executable (Linking)**: Connects pieces of object files together. This is a fast process

  Note: Many "compilers" do both compiling and linking. Most programs are built in two stages:

  1. Compile all the source code files
  2. Link the object code file into an executable

  This is the most efficient way to compile large projects. Changing a single source code file requires a small number of compilations (slow), followed by linking (fast).

- **Standard unix c compiler**: The standard is GNU gcc
- **Standard unix cpp compiler**: The standard is GNU g++
- g++ Options: With no options, g++ will go from source to an executable named a.out
  - **-o**: The -o option gives the name of the output file
  - **-c**: The -c option makes the compiler stop after the compilation stage. No linking is done. The name of the object code file is the same as the source with the extension replaced with .o
  - **-W**[***warning***]: Tell the complier to look for a specific warning
  - **-Wall (Warning all)**: There are many -W*warning* options, which warn of various conditions. -Wall warns about all of them. The compiler keeps going through warnings

    Note: A compiler warning is usually a bug waiting to happen. Do all you can to get rid of all warnings.
  - **-Werror**: The -Werror option turns all warnings into errors. The compiler aborts on an error.
  - **-g**: The -g option turns on debugging, and leaves much extra information in an object file. Executable is much larger, possibly slower.
  - **-0**: The -O option turns on optimization. There are several different levels of optimization, e.g. -O0, -O1, -O2, -O3.

    Note: Optimization may break your code, and -O and -g don't always work well together
  - **-I**[***directory***]: The -I option specifies an additional directory to search for include files. No space between -I and directory

    Thus,

    ```
    1 #include "./dir/headerfile" // Without -I
    2 #include "headerfile" // With -I : g++ -I./dir ...
    ```

  - **-L**[***directory***]: The -L option specifies an additional directory to search for libraries. No space between -L and directory.

Note: This option is meant for linking only. It has no effect in compilation.

- **-l[*librarynname*]**: The -l option specifies a library for linking. No space between -l and library name. The library name is related to the libray file name, but it is not identical. Library names start with "lib" and end with ".so.*" or ".a". These are removed. For example
  - The math library /lib/x86_64-linux/gnu/libm.so.6 is linked as -lm
  - The X11 graphics library /usr/lib/x86_64-linux-gnu/libX11.so is linked as -lX11

  Note: This option is for linking only. It has no effect in compilation. Libraries are the last things listed in a linking command.

  If you're linking against a library that is located in a non-standard directory (a directory that is not automatically searched by the linker, such as ./libs), then you need to tell the linker where to find that library using the -L option. Thus, -L tells the compiler where to look, -l specifies which one to grab.

2.2 Elementary complexity theory

- **Idea**: The same problem can frequently be solved with algorithms that differ in efficiency. The differences between the algorithms may be immaterial for processing a small number of data items, but these differences grow with the amount of data. To compare the efficiency of algorithms, a measure of the degree of difficulty of an algorithm called computational complexity was developed by Juris Hartmanis and Richard E. Stearns

  Computational complexity indicates how much effort is needed to apply an algorithm or how costly it is. This cost can be measured in a variety of ways, and the particular context determines its meaning. This book concerns itself with the two efficiency criteria: time and space. The factor of time is usually more important than that of space, so efficiency considerations usually focus on the amount of time elapsed when processing data. However, the most inefficient algorithm run on a Cray computer can execute much faster than the most efficient algorithm run on a PC, so run time is always system-dependent. For example, to compare 100 algorithms, all of them would have to be run on the same machine. Furthermore, the results of run-time tests depend on the language in which a given algorithm is written, even if the tests are performed on the same machine. If programs are compiled, they execute much faster than when they are interpreted. A program written in C or Ada may be 20 times faster than the same program encoded in BASIC or LISP.

- **Units**: To evaluate an algorithm's efficiency, real-time units such as microseconds and nanoseconds should not be used. Rather, logical units that express a relationship between the size n of a file or an array and the amount of time t required to process the data should be used

  If there is a linear relationship between the size n and time t, that is, $t_1 = cn_1$, then an increase of data by a factor of 5 results in the increase of the execution time by the same factor. If $n_2 = 5n_1$, then $t_2 = 5t_1$

  Similarly, if $t_1 = \log_2 n$, then doubling n increases t by only one unit of time. Therefore, if $t_2 = \log_2(2n)$, then $t_2 = t_1 + 1$.

- **Eliminating insignificant terms**: A function expressing the relationship between n and t is usually much more complex, and calculating such a function is important only in regard to large bodies of data; any terms that do not substantially change the function's magnitude should be eliminated from the function. The resulting function gives only an approximate measure of efficiency of the original function. However, this approximation is sufficiently close to the original, especially for a function that processes large quantities of data.

- **Asymptotic complexity**: This measure of efficiency is called asymptotic complexity and is used when disregarding certain terms of a function to express the efficiency of an algorithm or when calculating a function is difficult or impossible and only approximations can be found

- **Big-O Notation**: The most commonly used notation for specifying asymptotic complexity—that is, for estimating the rate of function growth—is the big-O notation introduced in 1894 by Paul Bachmann.

  Given two positive-valued functions f and g, consider the following definition:

$f(n)$ is $O(g(n))$ if there exist positive numbers c and N such that $f(n) \leqslant c \cdot g(n)$ for all $n \geqslant N$.

$$f(n) \text{ is } O(g(n)) \iff \exists\, c, N \in \mathbb{Z}^+ \mid f(n) \leq cg(n)\ \forall\ n \geq N.$$

Big-O notation says that for large enough n, the function $f(n)$ does not grow faster than a constant multiple of $g(n)$. So, $g(n)$ provides an upper bound on how fast $f(n)$ can grow as n increases.

In other words, f is big-O of g if there is a positive number c such that f is not larger than $c \cdot g$ for sufficiently large ns; that is, for all ns larger than some number N. The relationship between f and g can be expressed by stating either that $g(n)$ is an upper bound on the value of $f(n)$ or that, in the long run, f grows at most as fast as g.

The problem with this definition is that, first, it states only that there must exist certain c and N, but it does not give any hint of how to calculate these constants. Second, it does not put any restrictions on these values and gives little guidance in situations when there are many candidates. In fact, there are usually infinitely many pairs of c's and N's that can be given for the same pair of functions f and g.

For example, suppose

$$f(n) = 2n^2 + 3n + 1 = O(n^2).$$

Where $g(n) = n^2$. Candidate values for c and N are

c	$\geqslant 6$	$\geqslant 3^{\frac{3}{4}}$	$\geqslant 3^{\frac{1}{9}}$	$\geqslant 2^{\frac{13}{16}}$	$\geqslant 2^{\frac{16}{25}}$	$\cdots$	$\to$	2
N	1	2	3	4	5	$\cdots$	$\to$	∞

We obtain these values by solving the inequality:

$$2n^2 + 3n + 1 \leqslant cn^2.$$

Or equivalently

$$2 + \frac{3}{n} + \frac{1}{n^2} \leqslant c.$$

For different n's

For large n, the terms $\frac{3}{n}$ and $\frac{1}{n^2}$ get smaller. Let's find N such that for all $n \geqslant N$, the right-hand side stays bounded.

As n gets larger, $\frac{3}{n}$ and $\frac{1}{n^2}$ approach zero. To simplify the analysis, choose $N = 1$ initially and check how small $\frac{3}{n}$ and $\frac{1}{n^2}$ are:

$$2 + \frac{3}{1} + \frac{1}{1^2} = 2 + 3 + 1 = 6.$$

From the inequality, at $N = 1$, we have $6 \leqslant c$. Therefore, we can choose $c = 6$. This ensures that for all $n \geqslant 1$, the inequality holds:

$$2 + \frac{3}{n} + \frac{1}{n^2} \leqslant 6.$$

Thus, you can choose $c = 6$ and $N = 1$.

different pairs of constants c and N for the same function $g(= n^2)$ can be determined.

- **Choosing the best c, N**: To choose the best c and N, it should be determined for which N a certain term in f becomes the largest and stays the largest.

  In the example above, The only candidates for the largest term are $2n^2$ and $3n$; these terms can be compared using the inequality $2n^2 > 3n$ that holds for $n > 1.5$. Thus, $N = 2$ and $c \geqslant \frac{15}{4} = 3.75$.

- **Significance**: What is the practical significance of the pairs of constants just listed? All of them are related to the same function $g(n) = n^2$ and to the same $f(n)$. For a fixed g, an infinite number of pairs of c's and N's can be identified. The point is that f and g grow at the same rate. The definition states, however, that g is almost always greater than or equal to f if it is multiplied by a constant c. "Almost always" means for all n's not less than a constant N. The crux of the matter is that the value of c depends on which N is chosen, and vice versa.

- **Inherent imprecision: Choosing best $g(n)$**: The inherent imprecision of the big-O notation goes even further, because there can be infinitely many functions g for a given function f. For example, the f from Equation 2.2 is big-O not only of n^2, but also of n^3, n^4, ..., n^k, ...for any $k \geqslant 2$. To avoid this embarrassment of riches, the smallest function g is chosen, n^2 in this case.

- **Big-o as approximating terms**: The approximation of function f can be refined using big-O notation only for the part of the equation suppressing irrelevant information. For example, in the equation below, the contribution of the third and last terms to the value of the function can be omitted

$$\begin{aligned} & f(n) = n^2 + 100n + \log(n) + 1000 \\ \implies & f(n) = n^2 + 100n + O(\log(n)). \end{aligned}$$

  Similarly,

$$\begin{aligned} & f(n) = 2n^2 + 3n + 1 \\ \implies & f(n) = 2n^2 + O(n). \end{aligned}$$

  This equation says that for large values of n, the expression $2n^2 + 3n + 1$ behaves like $2n^2$ plus some terms that grow linearly or slower (captured by $O(n)$). The exact contributions of $3n$ and 1 are not important for asymptotic analysis; what matters is that their growth is slower compared to $2n^2$.

- **Algorithm analysis: Most common time complexities**: Ranked slowest to fastest growth
  - $O(1)$**:** Constant time
  - $O(\log(\log(n)))$: Logarithmic time
  - $O(\log(n))$: Logarthmic time
  - $O(n)$: Linear time
  - $O(n \log(n))$: Log-linear time
  - $O(n^k),\ k > 1$: Polynomial time
  - $O(a^n),\ a > 1$: Exponential time
  - $O(n!)$: Factorial time

- **Ranking complexities from slowest to fastest: Process**: Given

  (a) $O(25)$
  (b) $O(n^{\frac{1}{2}} + \log^2(n))$
  (c) $O(\log^{200}(n))$
  (d) $O(n^3 \log^4(n))$
  (e) $O(n^{200} + 3^n)$
  (f) $O(n \log^{40}(n))$
  (g) $O(4^n \log(n))$
  (h) $O(n^3 \log(\log(n)))$

  How can we go about sorting these slowest to fastest. Well, to start, in the expressions with plus or minus, we can throw out the slower terms. Thus,

  (a) $O(n^{\frac{1}{2}})$
  (b) $O(25)$
  (c) $O(\log^{200}(n))$
  (d) $O(n^3 \log^4(n))$
  (e) $O(3^n)$
  (f) $O(n \log^{40}(n))$
  (g) $O(4^n \log(n))$
  (h) $O(n^3 \log(\log(n)))$

  In product terms, we disregard the slower term unless there are complexites with the same dominant term. For example, $O(n^3 \log(\log(n)))$ grows slower than $O(n^3 \log^4(n))$ because although they have the same dominant term n^3, $\log(\log(n))$ grows slower than $\log^4(n)$. Thus, the correct sequence is

  (b) $O(25)$
  (c) $O(\log^{200}(n))$
  (a) $O(n^{\frac{1}{2}} + \log^2(n))$
  (f) $O(n \log^{40}(n))$
  (h) $O(n^3 \log(\log(n)))$
  (d) $O(n^3 \log^4(n))$
  (e) $O(n^{200} + 3^n)$
  (g) $O(4^n \log(n))$

Databases

3.1 Introduction to databases (db concepts)

3.1.1 Definitions and theorems

- **What is a database?**: A database is a collection of stored operational data used by the application systems of some particular enterprise, better yet a collection of related data.
- **What is an enterprise?**: a generic term for any reasonably large-scale commercial, scientific, technical, or other application. Such as
  - Manufacturing
  - Financial
  - Medical
  - University
  - Government
- **Operational data**: Data maintained about the operation of an enterprise, such as
  - Products
  - Accounts
  - Patients
  - Students
  - Plans

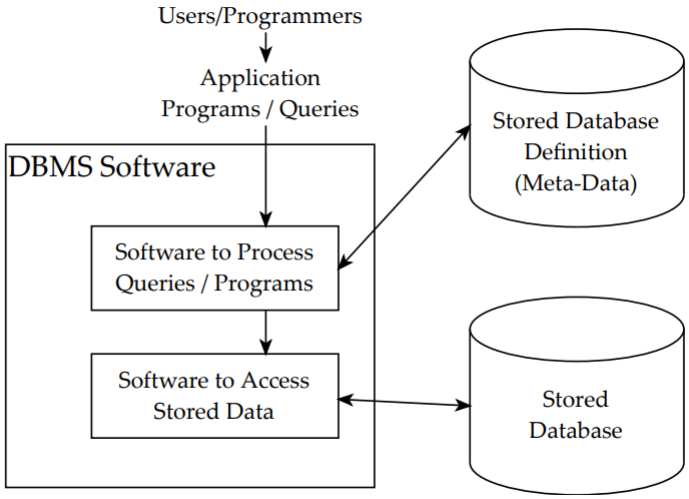
  Note: Notice that this DOES NOT include input/output data
- **Database Management System (DBMS)**: A Database Management System (DBMS) is a collection of programs that enables users to create and maintain a database. Ie a general-purpose software system that facilitates
  - Definition of databases
  - Construction of databases
  - Manipulation of data within a database
  - Sharing of data between users/applications
- **Defining a database**: For the data being stored in the database, defining the database specifies
  - The data types
  - The structures
  - The constraints
- **Constructing a Database**: Constructing a database is the process of storing the data itself on some storage device

Note: The storage device is controlled by the DBMS

- **Manipulating a Database**
  - retrieve specific information in a query
  - update the database to include changes
  - generate reports from the data

  Most likely already defined by whatever dbms you choose

- **Sharing a Database**: Sharing a database Allows multiple users and programs to access the database at the same time, any conflicts between applications are handled by the DBMS
- **Other Important Functions of a Database**: Other important functions provided by a DBMS include
  - Protection, system protection, security protection
  - Maintenence, allows updates to be performed easily
- **Simplified Database System Environment**:



- **Main characteristics of a database system are:**
  - Self-describing nature of a database system
  - Insulation between programs and data, and data abstraction
  - Support for multiple views of the data
  - Sharing of data and multi-user transaction processing
- **Other Capabilities of DBMS Systems**: Support for at least one data model through which the user can view the data, There is at least one abstract model of data that allows the user to see the "information" in the database, Relational, hierarchical, network, inverted list, or object-oriented

Support for at least one data model through which the user can view the data

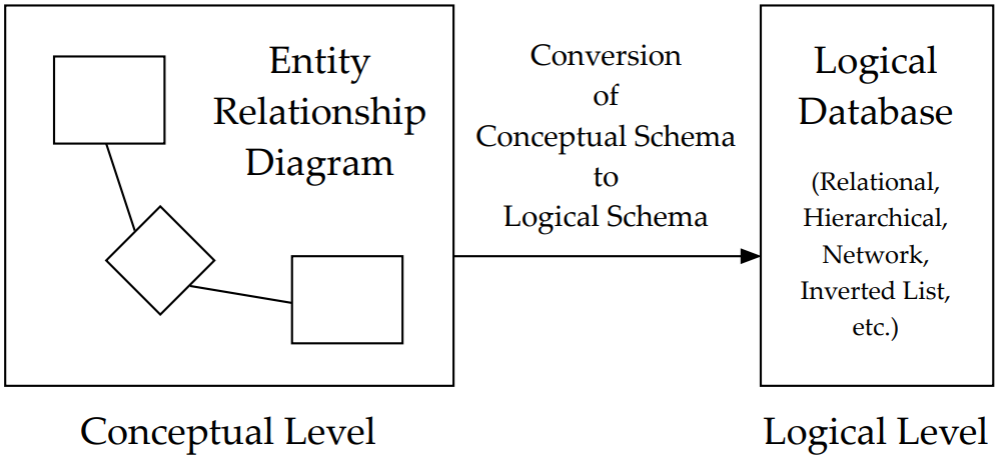
- efficient file access which allows us to "find the boss of Susie Jones"
- allows us to "navigate" within the data
- allows us to combine values in 2 or more databases to obtain "information"

Support for high-level languages that allow the user to define the structure of the data, access that data, and manipulate it

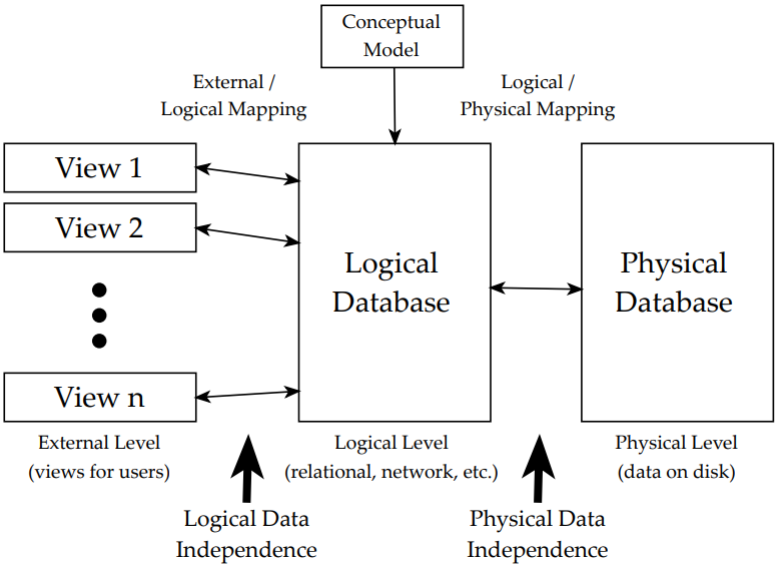
- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Data Control Language (DCL)
- query language access data
- operations such as add, delete, and replace

- **Transaction Management**: Transaction management is a feature that provides correct, concurrent access to the database, possibly by many users at the same time, ability to simultaneously manage large numbers of *transactions*
- **Access Control**: Access control is the ability to limit access to data by unauthorized users along with the capability to check the validity of the data. This is to protect against loss when database crashes and prevent unauthorized access to portions of the data
- **Resiliency**: Resiliency is the ability to recover from system failures without losing data, Ideally, should be able to recover from any type of failure, such as
  - sabotage
  - acts of God
  - hardware failure
  - software failure
  - etc.

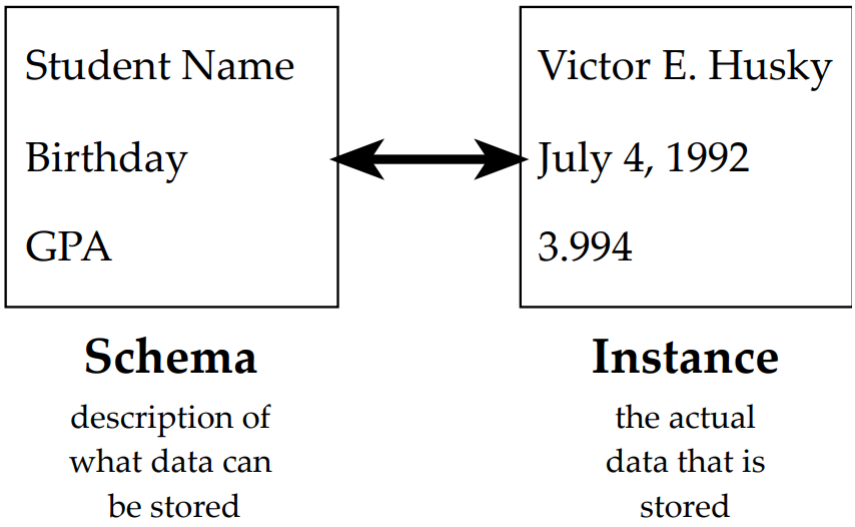
  Note: Obviously, some of these would require more than just software - offsite back-ups, etc
- **Use of Conceptual Modeling**:



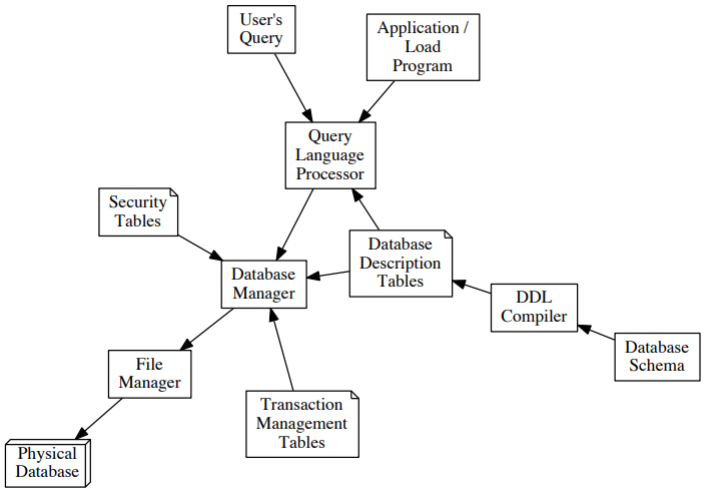
- **Leveled Architecture of a DBMS**:



- **External level**: a view or sub-schema, a portion of the logical database, may be in a higher level language
- **Logical Level**: abstraction of the real world as it pertains to the users of the database. DBMS provides a data definition language (DDL) to describe the logical schema in terms of a specific data model such as relational, hierarchical, network, inverted list, etc.
- **Physical Level**: The collection of files and indices, the collection of files and indices, this is the actual data
- **Instance**: An instance of the database is the actual contents of the data, it could be
  - the extension of the database
  - current state of the database
  - a snapshot of the data at a given point in time
- **Schema**: The schema of a database is the data about what the data represents. Such as,
  - plan of the database
  - logical plan
  - physical plan
  - the intention of the database
- **Schema vs Instance**:



- **Data Independence**: Data Independence is a property of an appropriately designed database system, it has to do with the mapping of logical level to physical level, and logical to external
  - **Physical data independence**: Physical schema can be changed without modifying logical schema
  - **Logical data independence**: logical schema can be changed without having to modify any of the external views
- **DCL (Control), DDL (Definition), DML (Manipulation)**: may be completely separate (example is IMS), may be intermixed (DB2), or may be a host language, for example an application program in which DML commands are embedded such as COBOL or PL/I
- **DBMS Components**:



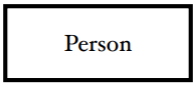
- **Overall DBMS Usage Scenario**: Database Administrator (DBA) define the conceptual, logical, and physical levels using DDL. DBMS software stores instances of these in schemas. User defines views (External Schema) in DDL. User accesses database using DML
- **Advantages of a Database**:
  - Controlled redundancy
  - Reduced inconsistency in the data
  - Shared access to data
  - Standards enforced
  - Security restrictions maintained
  - Integrity maintained more easily
  - Provides capability for backup and recovery
  - Permitting inferences and actions using rules
- **Disadvantages of a Database**:
  - Increased complexity needed to implement concurrency control
  - Increased complexity needed for centralized access control
  - Security needed to allow the sharing of data
  - Necessary redundancies can cause complexity when updating
- **Data vs Information**:
  - **Data**: Data refers to raw, unprocessed facts, figures, and details. It represents basic elements that have not been interpreted or given any meaning.
  - **Information**: Information is processed, organized, or structured data that is meaningful and useful. It is data that has been interpreted or analyzed to provide context, relevance, and purpose.

3.2 Conceptual Modeling and ER Diagrams

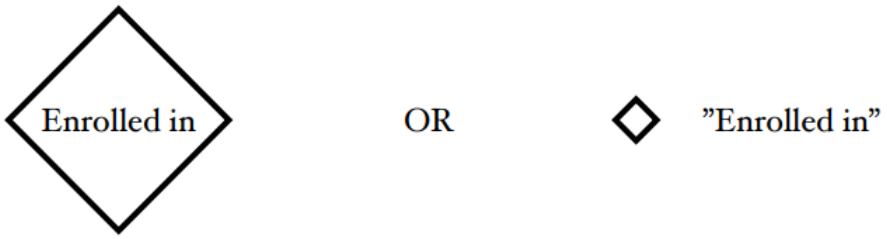
3.2.1 Definitions and theorems

- **Data Models**: A means of describing the structure of data, we typically have A set of operations that manipulate the data (for data models that are implemented)
- **Types of data models**:
  - Conceptual data model
  - Logical data models - relational, network, hierarchical, inverted list, or object-oriented
- **Conceptual Data Model**:
  - Shows the structure of the data including how things are related
  - Communication tool
  - Independent of commercial DBMSes
  - Relatively easy to learn and use
  - Helps show the semantics or meaning of the data
  - Graphical representation
  - Entity-Relationship Model is very common
- **Logical Data Models - Relational**: Data is stored in relations (tables). These tables have one value per cell. Based upon a mathematical model.
- **Logical Data Models - Network**: Data is stored in records (vertices) and associations between them (edges), Based upon a model called CODASYL
- **Logical Data Models - Hierarchical**: Data is stored in a tree structure with parent/child relationships
- **Logical Data Models - Inverted List**: Tabular representation of the data using indices to access the tables, Almost relational, but it allows for non-atomic data values [1], which are not allowed in relations
- **Logical Data Models - Object Oriented**: Data stored as objects which contain
  - Identifier
  - Name
  - Lifetime
  - Structure
- **Entity-Relationship Model**: Meant to be simple and easy to read. Should be able to convey the design both to database designers and unsophisticated users
- **Entities**: Principle objects about which information is kept - These are the *things* we store data about. If you look at the ER Diagram like a spoken language, the entities are nouns - Person, place, thing, event. When drawn on the ER diagram, entities are shown as rectangles with the name of the entity inside.

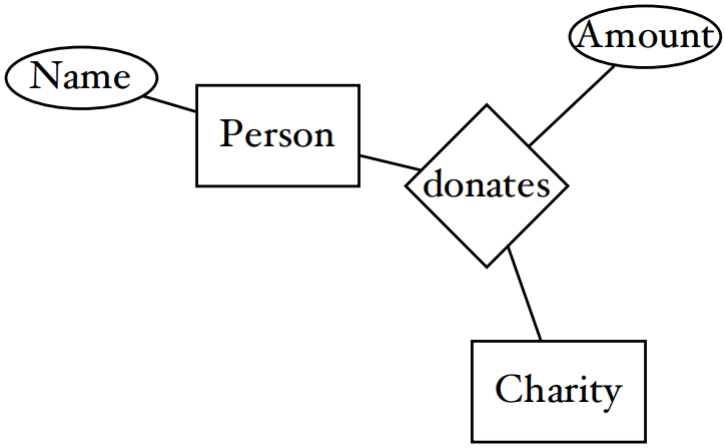
[1] "Non-atomic data values" refer to data structures or values that are composed of multiple components, as opposed to atomic data values, which are indivisible and represent a single value.



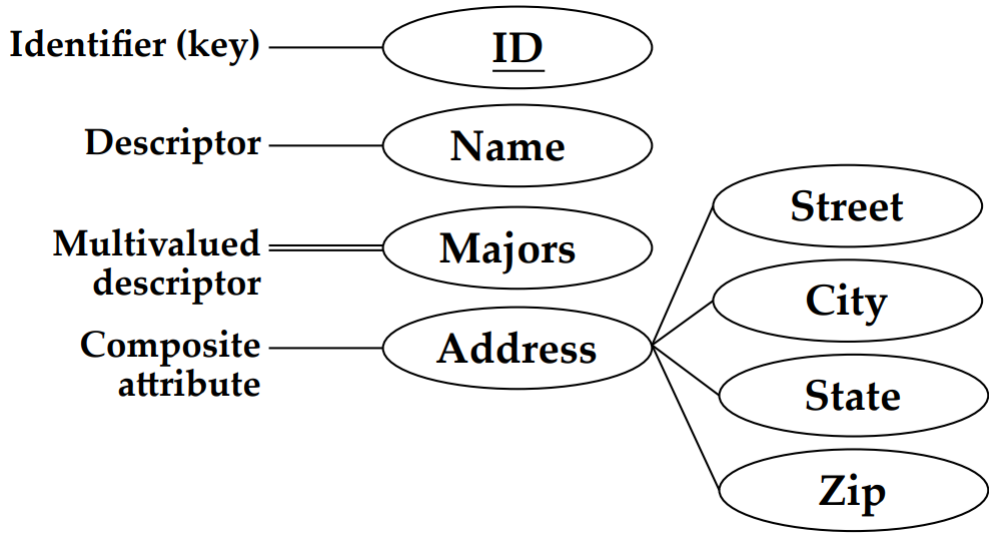
- **Relationships**: Relationships connect one or more entities together to show an association. A relationship *cannot* exist without at least one associated entity. Graphically represented as a diamond with the name of the relationship inside, or just beside it



- **Attributes**: Characteristics of entities **OR** of relationships, Represent some small piece of associated data, Represented by either a rounded rectangle or an oval.

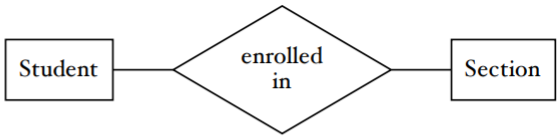


- **Attributes on Entities**: When an attribute is attached to an entity, it is expected to have a value for every instance of that entity, unless it is allowed to be null. For instance, in the diagram above, Name was an attribute of Person. Every person that we store data about will have a value for Name.
- **Attributes on Relationships**: When an attribute is attached to a relationship, it is only expected to have a value when the entities involved in the relationship come together in the appropriate way. In the diagram from before, the Amount attribute is attached to the donates relationship, which connects the Person and Charity entities. Amount will have one value for each time a Person donates to a Charity, denoting how much that person donated to the charity. It will not necessarily have a value for a given person, or a given charity. This can be referred to as the **intersection data**.
- **Types of attributes**:

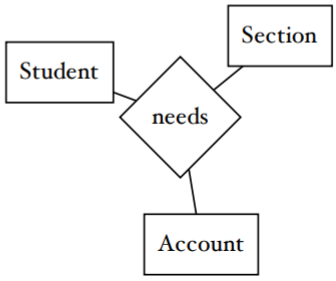


- **Degree of a Relationship**: The degree of a relationship is defined as how many entities it associates. If one entity is associated more than once (such as with a recursive relationship), then the degree counts each time it is referenced.

  - binary



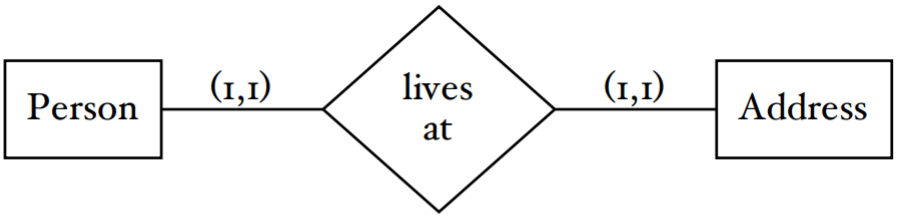
  - ternary



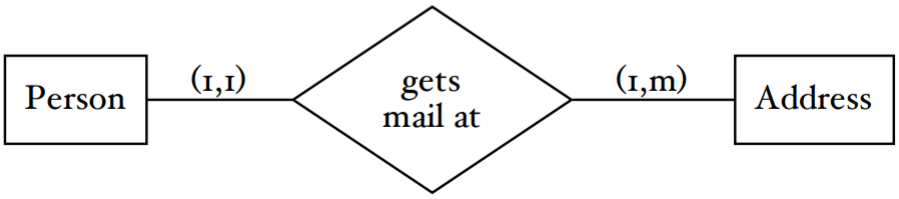
  Note: There is no limit to how many entities there can be in a relationship. After binary, and ternary, we start to call the relationships n-ary, where n is the degree

- **Connectivity of a Relationship**:
  - A constraint of the mapping of associated entities
  - Written as (minimum, maximum).
  - Minimum is usually zero or one.
  - Maximum is a number (commonly one) or can be a letter denoting many.
  - The actual number is called the cardinality.

- one-to-one

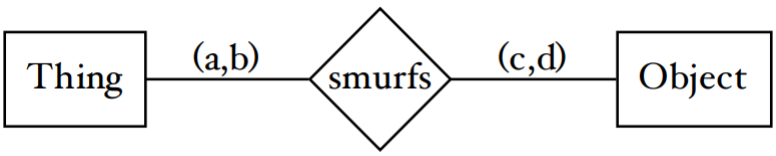


- one-to-many



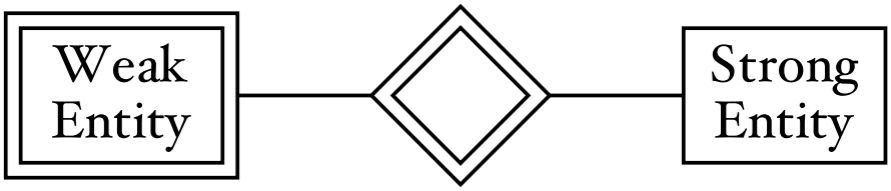
Together (from the image) both sides make up the connectivity, to refer to a single side, we use the term "cardinality", ie the cardinality of a person is (1,1). If we hold Address constant (We know a specific address and are therefore refering to that), how many persons may live at that address, in this case (1,1)

- **Attributes on Relationships (revisited)**: Must be on a many-to-many relationship. (1-many and 1-to-1 relationships should have the attribute on one of the entities involved. Someone needs to know all of the associated entities to access the attribute.
- **Reading Cardinalities**: For binary relationships:
  - For each Thing that smurfs, there are a minimum of c, and a maximum of d Objects.
  - For each Object that smurfs/is smurfed, there is a minimum of a and a maximum of b Things

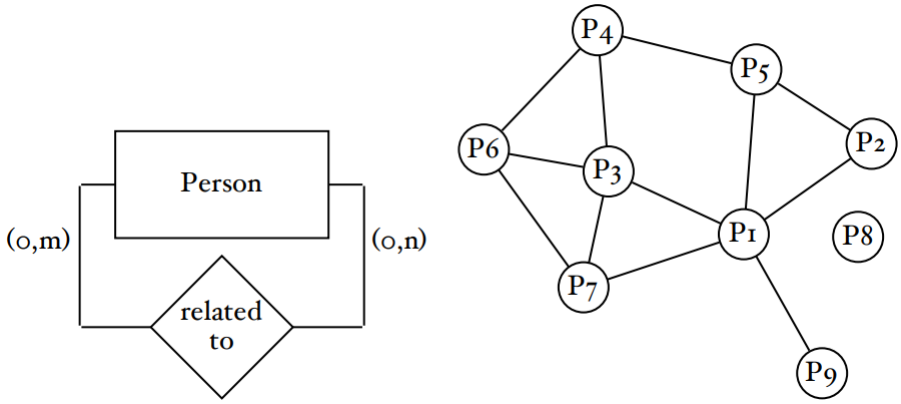


- **Weak Entities**: Sometimes you may run into an entity that depends upon another entity for its existence. The weak entity is a tool you can use to represent this.:w

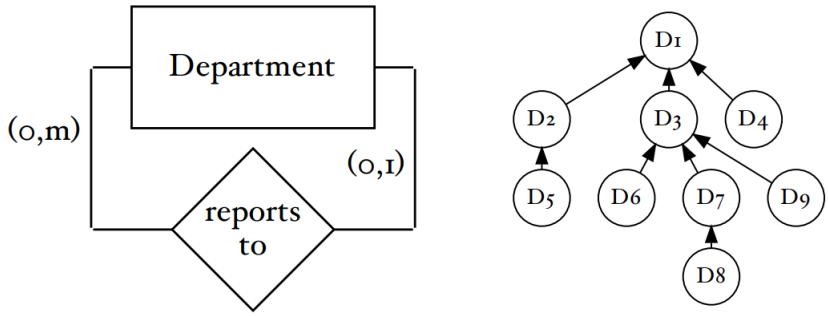
Weak entities are written like normal entities, except that they have a double rectangle outline. The relationship that connects the weak entity to the strong entity it depends upon will be written with a double diamond. This does not mean that the relationship is weak. It is just to indicate upon which entity the weak entity depends.



- **Recursive Relationships**: It is possible for an entity to have a relationship with itself. This is called a recursive relationship. It makes more sense if you think of entities as collections of objects of their appropriate type
- **Recursive Relationships - Many-To-Many**: A many-to-many recursive relationship means that the objects are arranged in a network structure, Notice that the minimum is 0 on both sides. This is important.



- **Recursive Relationships - One-To-Many**: A one-to-many recursive relationship means that the objects are arranged in a tree structure, Notice that the minimum is still 0 on both sides. This is important.

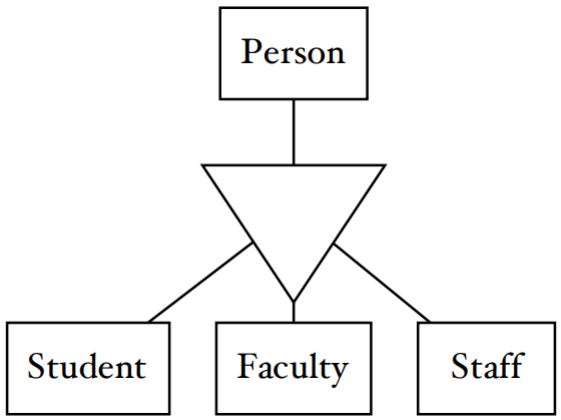


- **Entity or Attribute?**: Sometimes it isn't clear whether something should be an entity or an attribute of some other entity. Usually the decision will come down to how complicated it is to store the data, and how important it is. If it ends up being used in multiple places, it might be a clue that you should use an entity

- **Inheritance**: Two types of inheritance available
  - ”is a” inheritance. This shows that the subtype IS a member of the supertype.
  - ”is part of ” inheritance. This shows that the supertype contains, or is made up of members of the subtypes.

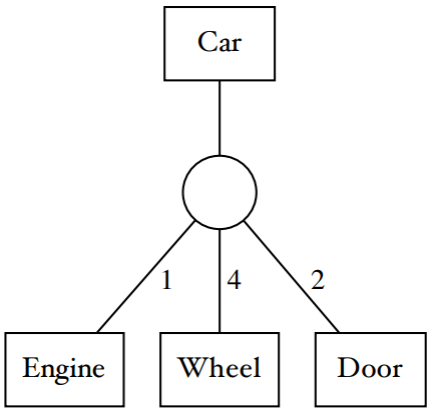
  All attributes of the supertype entity are inherited by the subtype entities. The identifier of the subtypes will be the same as the supertype

- **IS A Inheritance**: This type of inheritance happens when you have a supertype and one or more subtypes that are members of the supertype. Denoted by an upside-down triangle, with the supertype on top, and the subtypes coming out the bottom.



- **Defining IS-A inheritance**: There are two things you need to choose when using IS-A inheritance:
  - **Generalization (no) vs. specialization (yes)**: can the supertype occur without being a member of the specified subtypes?
  - **Overlapped (yes) vs. disjoint subtypes (no)**: is it possible for a single occurrence of the supertype to be a member of more than one subtype?

  They are mutually exclusive so you need to pick one of each, ie. GO, GD, SO, SD
- **IS-A inheritance - Generalization**: Supertype is the union of all of the subtypes, This means that an instance of the supertype CANNOT EXIST without belonging to at least one subtype.
- **IS-A inheritance - Specialization**: The subtype entities specialize the supertype, This means that an instance of the supertype CAN exist without being related to any of the subtypes
- **IS-A inheritance - Overlapping Subtypes**: It is possible for an instance of the supertype to be related to more than one of the subtypes
- **IS-A inheritance - Disjoint Subtypes**: the subtype entities are mutually exclusive, it is not possible for an instance of the supertype to be related to more than one subtype.
- **IS-PART-OF Inheritance**: ”Is part of ” inheritance indicates that the supertype is constructed from instances of the subtypes. It is shown on an ER diagram as a circle, with the supertype on the top, and subtypes on the bottom.

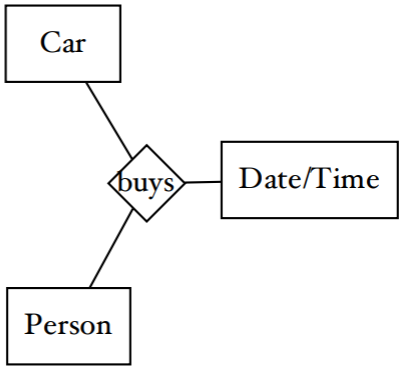


- **Warning about IS-PART-OF**: The IS PART OF inheritance operator does have its uses, but it is not very commonly used, If you see something involving a certain number of things being present, there are several possibilities
  - Sometimes a number is specified that isn't actually important for what we are modeling. This won't even be represented on an ER Diagram. This is the case when changing the number wouldn't have any effect on the necessary structure of a database.
  - If you need a certain number of items for a relationship to hold, you should explore using the connectivity of the relationship to express that.
  - Finally, this IS PART OF inheritance might be useful. It is almost never necessary, however.
- **Are you actually representing what you want to?**: Let's say you're running a business selling used cars. A simple ER diagram for the sales might look like the following:

The resulting database would have one entry for each time a specific person buys a specific car. If the same person buys the same car more than once (obviously selling it to someone else at some point), this model would no longer be appropriate.

The resulting database would have one entry for each time a specific person buys a specific car. If the same person buys the same car more than once (obviously selling it to someone else at some point), this model would no longer be appropriate.

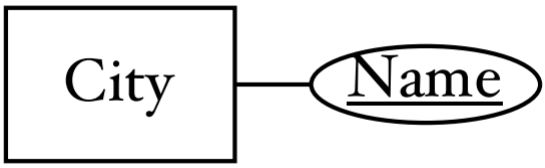
Adding a new entity to the relationship for the date/time of the purchase can fix this problem.



Notice that the connectivities can change when you add new entities to the relationship.

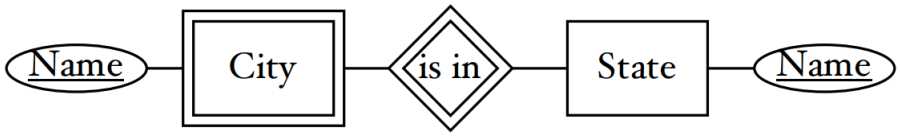
- **Weak Entities - Introduction**: So far, all of the entities we have used have been things that stand on their own. There are some situations where we are modeling an object for which we certainly need to store data, but the items exist only in the context of some other entity. Many of these examples can occur

  One example of a time that an entity depends on another would be the idea of a city. Within a state, we can generally be assured that cities will have unique names. If we were working only at that level, the City could be an entity as we saw above. A good identifier for it would be the name of the city, so we would see the following:



In some situations, this would be valid. The Name attribute can serve, in those circumstances, as an appropriate identifier.

To indicate this sort of dependency, we can make the dependent entity a "weak" entity. This is drawn with a double-edged rectangle, shown below.



Notice that the City entity is now drawn as a weak entity, with a double border. The relationship between the weak entity and the strong entity is also drawn with a double border. The relationship is not weak, per se, but it is used to indicate which strong entity the weak entity depends upon.

- **Discriminant (partial key)**: The discriminant, also known as the partial key, is an attribute (or a set of attributes) within the weak entity that can uniquely identify the weak entity, but only in combination with the primary key of the strong entity it is associated with. In other words, the discriminant helps to distinguish instances of the weak entity when they are tied to a particular instance of the strong entity.
- **Schema**: In databases, a schema is the structural definition of how data is organized in a database. It outlines the way data is stored

3.3 The Relational Model

- **Basic Structure**:
  - **Relations**: In the relational data model, our database is made up of one or more **relations** (tables). Each relation should have a unique name.
  - **Schema**: The schema of a relation is written as **Relation_Name**$(A_1, A_2, ..., A_n)$, Where $A_1, A_2, ..., A_n$ are placeholders for the attribute names
  - **Column headers (attributes)**: The attributes becomes the column headers of the relation.
  - **Instance data, tuples**: When there is instance data, it will come in the form of **tuples** (rows), which have a value for each attribute, as shown below

  Note: No field may contain than one value.

Relation_Name				
A_1	A_2	A_3	...	A_n
x_1	x_2	x_3	...	x_n
y_1	y_2	y_3	...	y_n
...	...	...		...

- **The domain of an attribute**: Each attribute becomes a column heading

  Each attribute (column) also has an associated **domain**. The domain of an attribute is the set of all valid values for it. The domain may be looked at as a data type, but may have additional constraints.

- **The domain of a set of attributes**: The domain of a set of attributes is the set of all possible combinations of values for the attributes in the set.
- **Tuples (Rows)**: A tuple is a special type of (mathematical) set containing values for each attribute within the relation. Tuples are shown as rows in the table, with the value for each attribute under the appropriate column
- **Atomic tuples**: The values are required to be atomic; there can be only one value per tuple per attribute
- **Relation vs relationship**: Though they have similar names, A relation (table) and a relationship (from an ER diagram) **ARE NOT** the same thing.
  - **Degree of relation**: The degree of a relation is the number of attributes present.
  - **Cardinality of a Relation**: The cardinality of a relation is the number of tuples present.
- **Keys**: Speaking generally, the purpose of a key is to uniquely identify a tuple in some relation.
  - **Super keys**: A super key within a relation is an attribute or set of attributes whose values can uniquely identify any tuple within that relation
  - **The trivial key**: Every relation has at least one - the set of all attributes in the relation
  - **Candidate Keys**: A candidate key is a minimal super key – the minimum set of attributes necessary to uniquely identify a tuple within the relation

  - **Primary Key**: The primary key for a relation is chosen by the database designer from among the relation's candidate keys. It becomes the "official" key that is used to reference tuples within the relation. There can be only one
  - **Prime, non-prime attributes**: Once a primary key is chosen, each of the attributes in the relation will be either **prime** or **non-prime** with respect to the relation. A prime attribute is one of the attributes that can be found in any of the candidate keys. A non-prime attribute is one of the attributes not found in any of the candidate keys

    Once a primary key is chosen for it, the schema of a relation is written with the primary key's attributes underlined
  - **Foreign Keys**: A foreign key is a tool used to link relations within a database. Since every relation has a primary key that uniquely identifies each tuple, the values of those key attributes can be used from another relation to reference individual tuples.

    The relation whose primary key is being used is the **home relation**

- **Order Independence**: In relations, the order things appear doesn't matter. There are ways to force them to sort later when we're working with SQL, but the relation itself has no order for either rows or attributes...
- **Order Independence - Attributes**: It doesn't matter what order the attributes appear in, if two relational schemas have the same name, the same attributes, and the same primary key, then they are equivalent.
- **Order Independence - Tuples**: Tuples are stored unordered. If you need to have them appear in some order later, you will be able to sort based on the values inside of them using SQL.
- **Constraints**: Constraints are limits imposed on the domains of various attributes. These can come from the system your database is modeling
- **Entity Integrity Constraint**: The entity integrity constraint applies to all relations. It states that no tuple may exist within a relation that has null value for any of attributes that make up the primary key. This is a consequence of the primary key being a candidate key, which is minimal and cannot do its job with any less data.
- **Referential Integrity Constraint**: It constrains the values of foreign keys in relations to values that actually exist as primary keys for tuples within the home relation. If the foreign key is otherwise allowed to be NULL, then that is also an acceptable value.
- **Summary: Terms**:
  - **Relations**: Tables
  - **Colums**: Attributes
  - **Tuples**: The rows in the relation that holds the instance datae
  - **Domain of an attribute**: Set of all possible values for the attribute
  - **Domain of a set of attributes**: Set of all possible combinations of values for the attributes in the set
  - **Degree of relation:** The degree of a relation is the number of attributes present.
  - **Cardinality of a Relation:** The cardinality of a relation is the number of tuples present.

3.4 Relational Model Normalization

- **Designing Relational Databases**: There are a large number of possible ways to represent each problem with using relations. Some choices will perform better than others for various reasons. The option chosen should be the best one, but how do we know which one that is?

  We should study:

  - Problems that can come up
  - How to avoid them
  - Desirable properties
  - How to guarantee them

- **Basic Example**: If our database is a single relation with schema **SP**(SuppName, SuppAddr, Item, Price) with the instance data:

SuppName	**SuppAddr**	**Item**	**Price**
John	10 Main	Apple	$2.00
John	10 Main	Orange	$2.50
Jane	20 State	Grape	$1.25
Jane	20 State	Apple	$2.25
Frank	30 Elm	Mango	$6.00

  There are some common things that we might want to do that would cause issues

  - **Insertion Anomaly**: Let's say we want to add a new vendor, "Sally", and store her address, "40 Pine", but she is not selling anything yet. Can this be inserted into the relation SP?

    NO. The primary key is (SuppName, Item), but we only have SuppName. The entity integrity constraint is violated if we try to insert the data as a tuple in this relation. It cannot fit. We call this an insertion anomaly.

  - **Deletion Anomaly**: This time, let's say that Frank no longer sells Mango. We want to take that out of the database so nobody can order a mango that is not available. Can this tuple remain in the relation with the Mango information removed?

    NO. The primary key is (SuppName, Item), and the Item is going away. The entity integrity constraint is violated if we remove the data from the tuple in this relation. We can either keep the whole tuple, advertising fake mango, or delete the whole tuple and lose the information on Frank, which doesn't exist in any other tuples. We call this a deletion anomaly.

  - **Update Anomaly**: Next, let's say that John is moving to a different address. We would have to change it once for every item John is selling. This isn't a big deal with only two items, but as John's list of supplied items grows, so does the amount of database work that needs to be done every time he moves. If any of the SuppAddr values for John don't agree, then it may not be clear which is the right address for John. This is an update anomaly.

  - **Redundancy**: Redundancy is when values are repeated.

It can be

  * **Good:** If you have an off-site backup of your entire database, the redundancy is useful, and can be used to restore in case of a failure.
  * **Bad:** Redundancy on the same physical device is unnecessary. It wastes space and comes with the potential for update anomalies.

- **Note:** The good redundancy is something the DBA/IT department should handle. When we talk about redundancy in the design of our database, we will be talking about the bad kind.

- **Anomolies summarized**:

  Insertion anomalies:

  - When a piece of data cannot be inserted because it violates some constraint of the relation.
  - Usually this is the entity integrity constraint being violated, but not always. See the Sally example

  Deletion anomalies:

  - When deleting some piece of data, a deletion anomaly is when more data is lost than intended
  - Usually this is caused when the data removed is part of the primary key, which would cause a violation of the entity integrity constraint. See the Frank example

  Update anomalies:

  - When updating a single value requires changes to multiple tuples, this is an update anomaly. See the John example.
  - This is caused by unnecessary redundancies in the data.
  - These cause inefficiency, and potential inconsistencies.

- **Decomposition**: There is no rule that says that a relational database must be made up of a single relation. The way we will solve these anomalies is to add new relations to our database and change the old ones. This is called decomposition.

  Using the example from above, we can remove the anomalies by decomposing the database into two relations.

  SP(<u>SuppName</u>, <u>Item</u>, Price)

SuppName	**Item**	**Price**
John	Apple	$2.00
John	Orange	$2.50
Jane	Grape	$1.25
Jane	Apple	$2.25

  S(SuppName, SuppAddr)

SuppName	SuppAddr
John	10 Main
Jane	20 State
Frank	30 Elm
Sally	40 Pine

- **When to decompose**: One way of designing a database could be to list all of the possible anomalies and then decompose to fix each of them. The problem with this is that any anomalies you don't see coming will not be fixed.

  We will look at a systematic method of identifying the potential for anomalies. This method is called normalization

- **Normalization**: Normalization involves making sure that each of your relations follows certain rules. Depending on which rules are followed, each of the relations in your database will be in one or more normal forms. These rules are based on functional dependencies

- **Functional Dependencies**: A functional dependency is a statement about which attributes can be inferred from other attributes. If we take X and Y as sets of attributes, we can write:

  $$X \rightarrow Y.$$

  Which means, if, whenever unique values for **all** of the attributes in X are known, unique values for **each** of the attributes of Y are guaranteed to be possible to look up or to infer using those values.

  This is read either as:

  - X functionally determines Y
  - Y is functionally dependent upon X

- **Functional Dependencies: Real-life Examples**:

  - **ZID** $\rightarrow$ **StudentFirstName, StudentLastName, Birthday**: If I identify a student using their ZID, that student has one first name, last name, and birthday
  - **StudentFirstName** $\not\rightarrow$**ZID**: The first name is not enough to determine a single ZID, as there are multiple students with the same first name
  - **ZID, CourseID, Semester** $\rightarrow$**Grade**: If I know which student, which course, and which semester, I can find a single grade

- **Functional Dependencies: Keep In Mind**: FDs are constraints present within the operational data your database models. They don't necessarily describe how things work in the real world, but they do have to accurately describe any data you will store in your database

  FDs **must** hold for all possible data values. Attempts to add data that does not obey the FDs will result in anomalies.

FDs can be enforced during insertion if the database is set up properly

- **Armstrong's Axioms**: Armstrong's Axioms are a set of rules for operations that are permissible when manipulating functional dependencies
  - **Reflexivity**: If $Y \subseteq X$, then $X \to Y$
  - **Augmentation**: If $X \to Y$, the $XZ \to YZ$ for any Z
  - **Transitivity**: If $X \to Y$ and $Y \to Z$, then $X \to Z$
  - **Decomposition**: If $X \to YZ$, then $X \to Y$ and $X \to Z$
  - **Composition**: If $X \to Y$ and $A \to B$, then $XA \to YB$
  - **Union (Notation)**: If $X \to Y$ and $Y \to Z$, then $X \to YZ$
  - **Pseudo-transitivity**: If $X \to Y$ and $YZ \to W$, then $XZ \to W$
  - **Self-determination**: $I \to I$ for any I
- **Functional Dependencies: Keys Revisited**: Now that we know about functional dependencies (FDs), we can assert:

  The attributes of a superkey must functionally determine all of the attributes of the relation.

  Candidate keys and primary keys are superkeys, so this is true of them as well, and they also satisfy additional requirements.

  Example: As an example, say we have the relation **R**(a,b,c,d,e,f). We can say

$$\begin{aligned} & a \to a, b, c, d, e, f \\ \implies & a \to b, c, d, e, f. \end{aligned}$$

- **First Normal Form (1NF)**: You should recall from the introduction to relations that all of the values in a tuple with a relation must be atomic. This means that there is a maximum of one value per attribute per tuple

  The requirement for a relation to be in First Normal Form (1NF) is this same requirement that all of the values must be atomic

  What this usually looks like is a table with mutltiple values in a single cell. A non-1NF relation would not even technically count as a relation.

  Given the table:

X	Y	Z
x1	y1	z1 z2 z3
x2	y2	z4
x3	y2	z5

  It looks like X would have been the primary key, but it's not doing its job of uniquely determining Z, which is showing as a repeating group so X can't be a key

What usually causes this is not having the correct primary key

The table above has the following function dependencies:

$$X \to Y$$
$$X, Z \to Z.$$

To move this pseudo-relation into an actual relation that doesn't violate 1NF, we need to choose a real primary key that meets the requirements. We do that using the FDs. In this case, (X , Z) works.

Changing the primary key yields: - $R(X, Y , Z))$

X	Y	Z
x1	y1	z1
x1	y1	z2
x1	y1	z3
x2	y2	z4
x3	y2	z5

- **Pseudo-relation**: The notation for a "pseudo-relation" like the one above would be to use inner parenthesis on the repeating group, ie. $\mathbf{R}(X, Y , (Z))$
- **Second Normal Form (2NF)**: Second Normal Form (2NF) has to do with the concept of full dependence.

  Given two sets of attributes, X and Y , we can say that Y is fully dependent on X, if (and only if)

  $$X \to Y.$$

  And no subset of X determines Y

  A relation is in 2NF if:

  – It already meets the requirements of 1NF, and
  – All non-prime attributes of the relation are fully dependent upon the entire primary key

  What breaks 2NF is when attributes are dependent upon only part of the primary key. To fix 2NF violations once we're in 1NF, decomposition is the solution.

  Example: Going back to our earlier example: **EmpProj**(EmpID, Project, Supv, Dept, Case)

EmpID	Project	Supv	Dept	Case
e1	p1	s1	d1	c1
e2	p2	s2	d2	c2
e1	p3	s1	d1	c3
e3	p3	s1	d1	c3

Functional Dependencies:

EmpID, Project → Supv, Dept, Case
EmpID → Supv, Dept
Supv → Dept

A quick glance confirms all of the values are atomic, so 1NF is confirmed.

There is a 2NF violation caused by (EmpID → Supv, Dept) because the primary key is (EmpID, Project), but only EmpID is on the LHS.

Observing the instance data, you should easily see that the attributes of the RHS cause update anomalies in this table. We also can't insert a new employee with no project (insertion anomaly), and removing e2 from p2 would remove e2 from the database entirely (deletion anomaly). These are symptoms of the 2NF violation.

Decomposition Pattern: There is a pattern to follow for the decomposition. Start with the original relation, and the FD that causes the violation.

EmpProj(EmpID, Project, Supv, Dept, Case)
EmpID → Supv, Dept.

The attributes on the RHS of the FD are removed from the original relation and placed into a newly created relation that has the FD's LHS as its primary key. A foreign key links the attribute from the LHS in the original table (the LHS is not removed) to the corresponding tuple in the new table, where it is the primary key.

EmpProj(EmpID, Project, Case)
Employee(EmpID, Supv, Dept).

Instance of 2NF Version:

EmpProj(EmpID, Project, Case)

EmpID	Project	Case
e1	p1	c1
e2	p2	c2
e1	p3	c3
e3	p3	c3

Employee (EmpID, Supv, Dept

EmpID	Supv	Dept
e1	s1	d1
e2	s2	d2
e3	s1	d1

- **Third Normal Form (3NF)**: To be in Third Normal Form (3NF), a relation must
  1. already qualify to be in 2NF
  2. none of the non-prime attributes may be transitively dependent upon the primary key

  By definition, all non-prime attribute are functionally dependent upon the primary key. What makes a transitive dependency is that there is also some non-prime attribute (which also depends on the key) that also functionally determines the attribute.

  To quickly identify the transitive dependencies from the list of FDs, look on the LHS for attributes that are non-prime in the context of the current relation.

  Example:

  EmpProj(EmpID, Project, Case)
  Employee (EmpID, Supv, Dept
  EmpID, Project → Supv, Dept, Case
  EmpID → Supv, Dept
  Supv → Dept.

  In this case, the FD that causes our relations to violate 3NF is (Supv → Dept), and the violation happens in the Employee relation. If you refer back to the instance data of that in the 2NF solution, you can see that the violation can cause anomalies, so we want to fix it.

  Just like 2NF, we fix 3NF by decomposing using the FD that causes the violation to occur. **AT NO POINT DO WE CHANGE THE FDs**

  Decomposition Pattern: We follow the same pattern for decomposition in 3NF as we did in 2NF. Start with the relation that has the violation, and the FD that causes the violation to occur.

  Employee (EmpID, Supv, Dept)
  Supv → Dept.

  The attributes on the RHS of the FD are removed from the violating relation and placed into a newly created relation that has the FD's LHS as its primary key. A foreign key links the attribute from the LHS in the original table (the LHS is not removed) to the corresponding tuple in the new table, where it is the primary key.

  Employee(EmpID, Supv)
  SupvDept(Supv, Dept).

  The RHS (Dept) that was a violation when it was in Employee because the LHS (Supv) was non-prime is no longer there to cause the problem. It is in the new relation where the LHS (Supv) is the primary key, and therefore we don't have a transitive dependency. These two relations no longer have the 3NF violation.

- **Summary of the normalization forms**:

First Normal Form (1NF):

- No repeating groups. All values are atomic.
- A primary key must have been chosen, and this primary key must be a proper superkey – it needs to be able to functionally determine every attribute in the relation.

1NF violations are fixed by choosing an appropriate primary key

Second Normal Form (2NF) - To be in Second Normal Form, a relation must conform to 1NF and:

- All of the non-prime attributes must be fully dependent upon the entire primary key.
- No non-prime attribute may be functionally determined by any subset of the primary key.
- No partial key dependencies

2NF violations are fixed by decomposition.

Third Normal Form (3NF) - To be in Third Normal Form, a relation must conform to 2NF and:

- There may be no transitive dependencies.
- No non-prime attribute may functionally determine another non-prime attribute.

3NF violations are fixed by decomposition.

3.5 ERD to Relations (Conceptual to logical)

- **The basic outline (steps)**

  1. Handle all of the entities
  2. Handle all of the relationships

- **Entity handling**: We will start with entities, because they can stand on their own, unlike relationships or attributes. In general, each entity will get its own relation. The attributes of the entity will become attributes in the schema of the relation created. There are some special cases to take into account, which will be handled from most independent to least, so:

  a. Strong (non-weak) entities that are not subtypes
  b. Strong.(non-weak) entities that are subtypes
  c. Weak entities

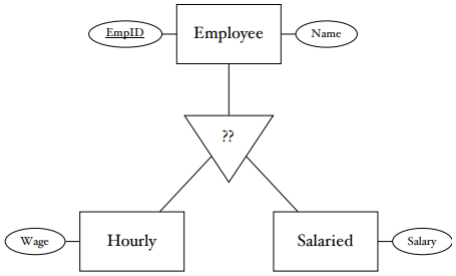
- **Entities like date**: there is no reason to make a relation for a "Date" entity or similar. The single value for the date is enough to determine it, and any other data associated with it is generally happening through a relationship anyway. Think about what data would go into such a table and how little use there would be for storing it separately.

- **Handling strong, non subtype entities**: Make a new relation, whose name will be the same as the name of the entity The primary key of the relation will be all of the identifier attributes, taken together All attributes of the entity become attributes of the relation Every instance of the entity gets the relevant values put into a new tuple in the relation

  Example: Suppose we had an entity A with attributes $\underline{ID}$, and other:

  Then, we would make a relation $A(\underline{ID}, other)$

- **Handling strong, subtype entities**: Suppose



  Employee is a supertype (not subtype) so it gets handled in the previous step

  Employee(<u>EmpId</u>, name)

  Hourly and **Salaried** are each strong, but they are subtypes (each is a type of Employee), so they are handled here

  This type of inheritance means that the subtypes are types of the supertype, so they are identified by **Employee's** EmpID

There are two methods of handling these.

1. **Big table**: The first method involves putting the attributes of the subtypes into the relation made for the supertype. So, the original relation:

   Employee(EmpID, Name)

   Would become something like:

   Employee(EmpID, Name,Wage, Salary)

   but it would need to be modified to indicate which subtypes a given employee belongs to. Let's examine that on the next page.

   The big table method needs a way to know which of the subtypes the current instance of the supertype belongs to, which is handled differently depending on the IS-A's configuration.

   For **disjoint subtypes**, where an instance of the supertype can only be one of the subtypes at a timei, we can add an attribute, EmpType that has a value indicating which type this employee is.:

   Employee(EmpID, Name, EmpType,Wage, Salary)

   For generalization, EmpType would not allow NULL. For specialization, it would be allowed.

   For **overlapping subtypes**, it is possible to be more than one at a time, so we need an individual true/false answer for each type:

   Employee(EmpID, Name, IsHourly,Wage, IsSalaried, Salary)

   In this case, nothing about the schema would indicate generalization vs. specialization

2. **New relation**: Method 2 involves creating a new relation for the subtype entity The name of new relation would be the same as the name of the entity.

   The primary key of the new relation would be the same as the primary key for the supertype's relation.

   The primary key is also a foreign key to the existing table.

   An instance of the supertype entity will only have a tuple in the subtype relation if it is a member of that subtype, so we will not need any extra attributes like we did in method 1.

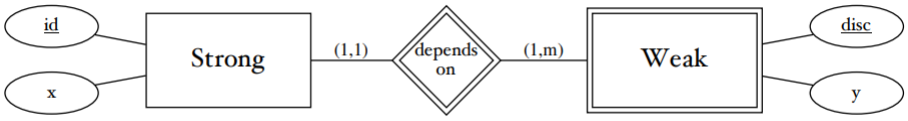
   The foreign key can be used to look up any of the attributes that are being inherited from the supertype

   Thus, we would have

   Employee(EmpID, Name
   Hourly(EmpID†,Wage)
   Salaried(EmpID†, Salary).

Note: The (†) (dagger symbol) will be used in these slides to indicate that the attribute is part of a foreign key (and, in this example, the whole thing).

- **Handling weak entities**: Suppose



Example with weak entity

The strong entity would already have a relation.

Strong(id, x)

The weak entity gets its own relation. The primary key will be the concatenation of the weak entity's discriminator with the strong entity's identifier. The other attributes of the entity are brought in as non-prime attributes.

Weak(id†, disc, y)

The id portion is a foreign key to the Strong relation

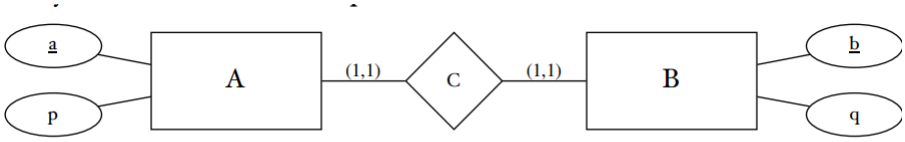
- **Entities: Functional Dependencies**: The only functional dependencies introduced by the entities of an ER diagram are the ones introduced when the identifiers become primary keys. Remember that a primary key has to functionally determine all of the other attributes in a relation
- **Handle relationships**: The relationships will be handled in order from lowest degree to highest degree, and within that, from simplest cardinality (one-to-one) to more complicated cardinalities (many-to-many, etc.).

  The purpose of a relationship is to form connections between entities. We know that we are using relations to represent our entities, so we will need to use a tool that can link those relations to each other.

  The tool best suited to linking tuples from relations together is the foreign key.

  Every relationship we model in the relational model will have one or more foreign key involved. Where we put these foreign keys will depend on the cardinality, and the decisions are motivated by the normal forms we discussed.

  1. **Binary one-to-one Relationships**: In a binary relationship, we will already have made a relation for each of the entities involved.



Here, C is a binary, one-to-one relationship between A and B.

A(a,p) *and* **B**(b, q)

Since each instance of B will have one of A, and each instance of A will have one of B through C, we can represent this one-to-one relationship by putting a new foreign key into the entity for either side. Choose either:

$$\mathbf{A}(\underline{\text{a}},\text{p},\text{b}\dagger) \qquad \textit{or} \qquad \mathbf{B}(\underline{\text{b}},\ \text{q},\ \text{a}\dagger)$$

The relationship implies the functional dependencies:

$$a \to b$$
$$b \to a.$$

2. **Binary one-to-many Relationships**: In a binary relationship, we will already have made a relation for each of the entities involved.

$$\mathbf{A}(\underline{\text{a}},\text{p}) \qquad \textit{and} \qquad \mathbf{B}(\underline{\text{b}},\ \text{q})$$

For this one-to-many relationship, there can be many instances of B for each of A, so we can't have the foreign key in the A table (wouldn't be atomic, so 1NF would be violated). We still do have the option of putting a foreign key in the B table pointing to the corresponding A, so our only option is:

$$\mathbf{B}(\underline{\text{b}},\text{q},\text{a}\dagger)$$

The only FD is

$$b \to a.$$

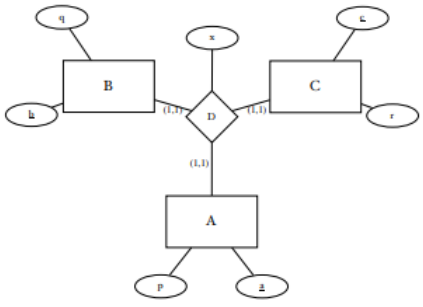
3. **Binary many-to-many Relationships**: In a binary relationship, we will already have made a relation for each of the entities involved.

$$\mathbf{A}(\underline{\text{a}},\text{p}) \qquad \textit{and} \qquad \mathbf{B}(\underline{\text{b}},\ \text{q})$$

There are no new functional dependencies introduced by the relationship, and putting a foreign key into either relation would not be atomic (1NF violation). The many-to-many relationship requires a new relation. Its foreign key will be the concatenation of the primary keys of each of the entity relations, which will be used as foreign keys to the corresponding tables. Any intersection data is put into this new relation as a non-prime attribute.

$$\mathbf{C}(\underline{\text{a}}\dagger,\ \underline{\text{b}}\dagger,\ \text{x})$$

4. **Relationships Greater than Binary: one-to-one-to-one**:



So we have

$$\mathbf{A}(\underline{a},\ p) \text{ and } \mathbf{B}(\underline{b},\ q) \text{ and } \mathbf{C}(\underline{c},r$$

Each of the "one legs" represents a functional dependency, and each of them gives us a potential relation to choose from for our relation.

Note: If we have say only two ones, like a one to one to many relationship, we would just have less functional dependencies and therefore less options to choose from (see table above)

Functional Dependency	**Potential Relation for D**
$a, b \to c$	$\mathbf{D}\left(a^{\dagger}, b^{\dagger}, c^{\dagger}, x\right)$
$b, c \to a$	$\mathbf{D}\left(a^{\dagger}, b, c^{\dagger}, x\right)$
$a, c \to b$	$\mathbf{D}\left(a^{\dagger}, b, c^{\dagger}, x\right)$

5. **Greater than Binary without any "ones"**: No functional dependencies are implied by this relationship. To stay in 3NF, the relation we must use is:

   $$\mathbf{D}(\underline{a}\dagger,\ \underline{b}\dagger,\ \underline{c}\dagger,\ x)$$

6. **Date entities (and similar)**: For relationships that have a "Date" entity (or the equivalent), recall that we did not make a relation for that entity. The only change necessary for your relationship involving that entity is that the date value is used instead of a foreign key, and that attribute will not be a foreign key, because the home relation would not exist

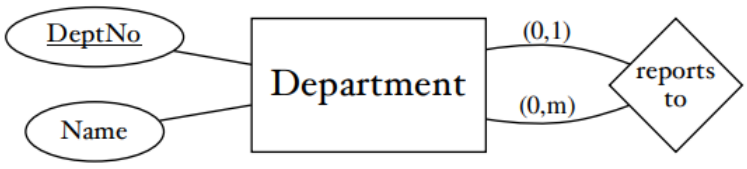
   As an example, if the C entity in the ternary relationship with no "ones" ER diagram were a Date entity, we would not create the C relation for it, and the relation to represent the relationship would be modified. Notice that the attribute is still part of the primary key, but no longer a foreign key.

   From: $\mathbf{D}(\underline{a}\dagger,\ \underline{b}\dagger,\ \underline{c}\dagger,\ x)$
   To: $\mathbf{D}(\underline{a}\dagger,\ \underline{b}\dagger,\ \underline{c},\ x)$

   .

7. **Recursive Relationships: one-to-many**: Recursive relationships will be handled as if they were normal relationships of the same degree and cardinality. The practical difference is that the entity that is linked multiple times will still only have one relation, so multiple foreign keys might go to the same table.

   Suppose:



   There should obviously only be one relation for the entity Department, because it is only a single entity.

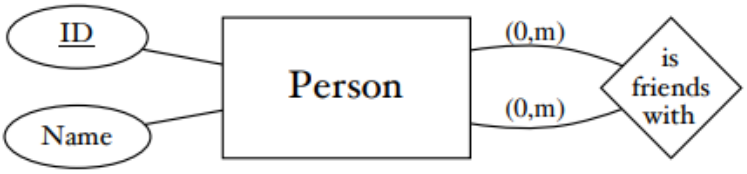
   Department(DeptNo, Name)

With a non-recursive one-to-many binary relationship, we would have put a foreign key to the relation for the one side into the relation on the one side. In this version, we only have one table, so the decision is easy. We will need to come up with another name for the foreign key, as we cannot have two attributes with the same name inside the same relation. Thus, we grow Department into the following:

Department(DeptNo, Name, ReportsToDept†)

Where the home relation for the new attribute, ReportsToDept, is that same relation, Department. The tuple of the department that the current department reports to will be have a DeptNo that equals the ReportsToDept in the current tuple. Alternatively, ReportsToDept can be NULL if the department does not report to another.

8. **Recursive Relationships, many-to-many**: Suppose



Like non-recursive many-to-many relationships, we will need to create a new relation. Unlike the non-recursive version, we only have one home relation for our two foreign keys. As in the one-to-many version, we will need to choose a new name for at least one copy of the foreign key, since they can't share the same name. The relation for our Person entity would be **Person**(ID, Name)

The new relation created to represent the relationship would be in the following form:

Friends(activeFriend†, passiveFriend†)

ActiveFriend and PassiveFriend are foreign keys to the tuple in Person with data for the person that is taking part in the relationship. This can be done in a directed or undirected way, and you probably want to put a comment somewhere about which way you intend to use it.

directed: (Person1, Person2) would not imply (Person2, Person1)

undirected: (Person1, Person2) does imply (Person2, Person1)

ID	Name
1	Regina George
2	Karen Smith
3	Cady Heron
4	Gretchen Wieners
5	Janis Ian

ActiveFriend	PassiveFriend
2	1
3	5
3	2
3	4
4	2
4	3
5	3

- **Summary**:
  1. Strong, non-subtype entities
     - New relation, PK is entities identifiers
  2. Sub-type entities
     - New relations, PK is supertype identifiers, which are foreign keys to supertype relation
  3. Weak entities
     - New relation, PK is concatenation of strong identifier and discriminator. Strong Id from the concat is FK to strong relation.
  4. **Relationships: Binary 1-1**
     - Put foreign key in either side
  5. **Binary 1-m**
     - Put foreign key to one side in the many side
  6. **Binary m-m**
     - New relation, PK is concatenation of both entities keys, which also serves as foreign keys to entities.
  7. **n-ary 1-1-...-1 (all ones)**
     - New relation, choose n-1 entities for PK, put remaining entity ID as non-prime, but foreign.
  8. **n-ary 1-1-...-m (Two ones)**
     - New relation, choose all many legs and one of the one legs for PK, put remaining one leg as non-prime but foreign
  9. **n-ary 1-m-...–m (Single one)**
     - New relation, choose all many legs for PK, put remaining one leg as non-prime but foreign
  10. **n-ary m-m-...-m (No ones)**
      - New relation, all legs are PK

11. **Handle date entities, and things of that nature**
    - Since we do not create relations for these types of entities, we cannot make them foreign keys, because the home relation will not exist. They can still be part of the PK.
12. **Recursive relationships**
    - We handle these the same, but the foregin key will link to the same relation. Make sure to put a comment somewhere to specify directed or undirected.

3.6 MariaDB, SQL

- **DDL**:
  - CREATE TABLE
  - ALTER TABLE
  - DROP TABLE
- **DML**:
  - INSERT
  - UPDATE
  - DELETE
- **MariaDB navigation**:
  - **USE <x>**: select the database <x>
  - **SHOW TABLES** list all of the tables in the current database
  - **DESCRIBE <x>** show the properties of each column of table <x>
  - **SHOW CREATE TABLE** <x> show a CREATE TABLE statement that can be used to reconstruct table <x>

3.6.1 DDL

- **Creating a new table with CREATE TABLE**: The basic format of a CREATE TABLE statement. []'s and <>'s are not to be typed. [] indicates that the contents are optional, and the <>'s indicate placeholders:

```
CREATE TABLE <table_name> (
    <attribute> <type> [NOT NULL] [UNIQUE] [PRIMARY KEY], [
    ... ]
    [PRIMARY KEY(<pkattrs>),]
    [FOREIGN KEY(<attr_here>) REFERENCES
    <home_table>(<attr_home>)]
);
```

  - <table_name> name of the table
  - <attribute> name of the current attribute
  - <type> data type of the current attribute
  - <pkattrs> comma-separated list of the attributes makeing up the table's primary key
  - <attr_here> comma-separated list of attributes in the current table forming a foreign key
  - <home_table> name of the home table
  - <attr_home> comma-separated list of attributes in the home table, matching the attributes in <attr_here>

- **Table / Column names**: When choosing a name for a table or a column, we can use the following characters:

  - any of the normal upper or lower case letters (regexp: [A-Za-z]
  - an underscore – _
  - a dollar sign – $
  - digits, but only after the first character

  The following limits are in place:

  - Table names must be unique within the database. They share the same namespace with views.
  - Attribute/column names must be unique with each table.
  - Unless quoted properly with backticks, reserved keywords cannot be used as identifier

  Note: These identifiers may or may not be case sensitive, depending on the locale setting of the server.

  Generally, the maximum length of an identifier is 64 characters.

- **Data types**

  - **INT/INTEGER**: integer values
  - **FLOAT**: single precision floating point numbers

  - **DOUBLE/REAL**: double precision floating point numbers
  - **DECIMAL(i,j)**: decimal numbers, i digits total, j after the decimal point .
  - **CHAR(n)**: character string exactly n characters long
  - **VARCHAR(n)**: variable-length character string up to n characters long
  - **DATE**: date in 'YYYY-MM-DD' format
  - **TIME**: time in 'HH:MM:SS' format
  - **DATETIME**: date/time in 'YYYY-MM-DD HH:MM:SS' format, no timezone conversion
  - **TIMESTAMP**: date/time in 'YYYY-MM-DD HH:MM:SS' format, timezone conversion

- **Column Options**: Here are some common options that can be applied to a column/attribute. They are written right after the type when defining a new column in a CREATE TABLE statement
  - **NULL**: allows NULL to be stored as the value for this attribute (default)
  - **NOT NULL**: prevents NULL from being stored as the value for this attribute
  - **UNIQUE**: ensures that no two tuples have the same value for this attribute
  - **PRIMARY KEY**: declares this attribute to be the entire primary key
  - **AUTO_INCREMENT**: next-available value auto-assigned for this attribute when not provided
  - **DEFAULT <x>**: sets the default value of the attribute to <x> when not supplied

- **Setting the Primary Key**: There are two ways to set the primary key:
  1. For single-attribute primary keys, you can use the PRIMARY KEY column option. The option may only be used once, and proclaims that the single attribute is the entirety of the primary key.
  2. If you have multiple attributes in the primary key, the only way is to add the separate constraint:

     ```
     PRIMARY KEY(<x>,<y>,<z>,<etc>)
     ```

     This can also be used for single attribute primary keys.

     Note: It should be obvious that only one primary key can be set.

- **Comments**: MariaDB supports the following comment syntax
  1. **Pound ()**:
  2. **Double hypen (--)**: This is the standard style
  3. **C-style multiline comments (* ... *\)**

- **Quotes**: There are two types of quotes that you may encounter in SQL.
  1. **Quotes for values – single quotes 'value'**: not necessary for numeric values, but can be used without breaking them, always required for string values. If it is ambiguous whether something is a value or an identifier, use these quotes
  2. **Quotes for identifiers – backticks 'identifier**: not necessary for identifiers that follow the rules from above, but can be used anyway, can allow identifier names to contain characters not otherwise allowed. Can allow identifiers to use names that would normally be reserved keywords

Note: Notice that identifier in the SQL context is a different thing than an identifier in an ER diagram. Here, identifier will mean the name of some table, column, variable, etc.

- **An example of CREATE TABLE**: Let's go ahead and make the SQL CREATE TABLE statement to create a table for the relation:

Person(<u>SSN</u>, FNAME, LNAME, PHONE)

```
CREATE TABLE Person(
    SSN CHAR(9) PRIMARY KEY, # SSN BAD IDEA, PK on same line
    ↪ (1)
    FNAME CHAR(20) NOT NULL, # First name
    LNAME CHAR(20) NOT NULL, # Last name
    PHONE CHAR(10) # Phone number
);
```

The relational schema we started with does not have information on data types or column options other than PRIMARY KEY, so we choose them while creating the table.

- **Setting up a foreign key**: A foreign key links the current table to another table, which we call the home relation.

  1. The foreign key must contain all of the attributes of the primary key of the home relation.
  2. They may have different names in each of the tables, but there needs to be a match for each.
  3. Each of these attributes must have the exact same data type as its counterpart in the home table.

  If a table is to contain a foreign key, we include a constraint in our CREATE TABLE statement like the following:

```
FOREIGN KEY (<localnames>) REFERENCES
    ↪ <home_table>(<homenames>)
```

  This can be done for multiple foreign keys, filling in the placeholders <localnames>, <home_table>, and <homenames> appropriately for each.

- **Table with foreign key example**: Let's make a table for a subtype of Person, Student:

Student(<u>SSN</u>†, CLSYEAR, GPA, TOTALHRS)

```
CREATE TABLE Student (
    SSN CHAR(9) NOT NULL, -- SSN is BAD IDEA
    CLSYEAR CHAR(9), -- fresh/soph/junior/senior
    GPA DECIMAL(4.3), -- 4.000, we hope
    TOTALHRS INT,

    PRIMARY KEY (SSN), -- set up the primary key separately (2)
    FOREIGN KEY (SSN) REFERENCES Person(SSN) -- a Student is a Person
);
```

Note: We need to use SHOW CREATE TABLE to show the get information of the foreign keys of a table.

- **Change existing table schema: ALTER TABLE**: An ALTER TABLE statement will allow you to have the DBMS make changes to the schema of a table that has already been created. It works with various subcommands. The three we will cover are:

  1. ALTER TABLE ADD
  2. ALTER TABLE MODIFY
  3. ALTER TABLE DROP

- **ALTER TABLE ADD**: The ALTER TABLE ADD command can be used to add a new column or new columns to the schema of an existing table.

  To add a single column/attribute

```
ALTER TABLE <table_name> ADD <attribute> <type>;
```

  To add multiple columns/attributes:

```
ALTER TABLE <table_name> ADD (<attribute> <type>, ...);
```

- **ALTER TABLE MODIFY**: The ALTER TABLE MODIFY command can be used to change properties of a column/attribute (including type, length, and other column options) in a table that already exists.

```
ALTER TABLE <table_name> MODIFY <col_name> <new_options>;
```

- **ALTER TABLE DROP**: The ALTER TABLE DROP command can be used to remove a column/attribute from the schema of a table.

```
ALTER TABLE <table_name> DROP <col_name>;
```

- **SHOW TABLES**: In MariaDB/MySQL, if you want to see a list of the tables present in the current database, you can use the command:

```
SHOW TABLES;
```

- **DROP TABLE**: To remove a table from the database, we can use the DROP TABLE command.

```
DROP TABLE <table_name>;
```

- **Termination of commands (;)**: Notice in all sql code examples we have a semi colon after the command / line, this is needed to execute the command.

3.6.2 DML except SELECT

- **DML Introduction**: The Data Manipulation Language (DML) is the language used to work with the instance data. In SQL, this means doing things with the rows contained by tables, rather than to the tables themselves. We have
  - **INSERT**: Add a new row to a table
  - **UPDATE**: Change values in an existing row
  - **DELETE**: Remove rows from the table
  - **SELECT**: Display the data stored in rows (In the next subsection)
- **INSERT**:

```
INSERT INTO <table_name>
    VALUES (<value_list>);

INSERT INTO <table_name>
    (<attr_list>)
    VALUES (<value_list>);

INSERT INTO <table_name>
    <another_query>;
```

  Where
  - **<table_name>**: The name of the table where the row should be added.
  - **<value_list>**: A list of values for the new row. If no <attr_list> is given, then the values are for each of the columns of the table, in order.
  - **<attr_list>**: A list of names of attributes that match up with the values in <value_list>. This allows us to omit optional columns or change the order.
  - **<another_query>**: A query that returns rows, like a SELECT statement. The rows returned are inserted into the table.

  Notes: Without the attribute list, there must be a value in the VALUES() for every column, and they have to be in the same order as they had in the table.

  Columns not in the attribute list are set to their default value if possible. This is why PHONE is NULL. This version of the INSERT statement is better if you're making SQL that needs to be in a script that is to be run later, as it tolerates more changes to the table schema than the other version.
- **The WHERE clause**:

```
... WHERE <expression> ...
```

  When working with DML statements, it will be desirable to be able to work only with specific rows. This can be accomplished using a WHERE clause.

  The WHERE clause is the keyword WHERE followed by an expression that evaluates to either true or false. It is included in an SQL query to control which rows are affected by the query

The expression after WHERE is evaluated one time per row. Rows where the expression evaluates as true are included in the operation. Rows where the expression evaluates to false are excluded from the operation

WHERE clauses are generally used in UPDATE, DELETE, and SELECT statements.

- **UPDATE**:

```
UPDATE <table_name>
    SET <attr> = <value> [, <attr> = <value> ...]
    [ WHERE <expression> ];
```

Where

  - **<attr>**: name of a column to change
  - **<value>**: value to assign to <attr>
  - **<expression>**: expression evaluated for each row to determine if the row is affected

- **DELETE**: To delete the rows without getting rid of the table, use a DELETE statement.

```
DELETE FROM <table_name>
    [ WHERE <expression> ];
```

It is important to realize that all rows are affected by default, so if a WHERE clause is not supplied, all of the rows will be deleted.

- **Views in SQL**: A view in SQL is a virtual table. It does not store its own data, but rather derives it from the other tables (or views) via a query that is a part of its definition.

Views do not contain their own data. They dynamically grab their data from the base tables on demand. Thus, changes to the data in the base tables will be reflected in the views that derive from them automatically

- **CREATE VIEW**:

```
CREATE VIEW <view_name>
    [( <view_col_name> [, <view_col_name>]...)] # can rename
    ↪ columns here
    AS SELECT <attr_name> [, <attr_name>] ...
        FROM <source_table_or_view> [, ...]
        WHERE <condition>;
```

The portion after the AS keyword is a SELECT statement, part of the DML that is used to ask the DBMS to show portions of instance data (rows from tables).

Once the view is created, it supports DML queries in most of the same ways a non-virtual table can be. Writing to a view is sometimes possible, but depends on how the SELECT statement that constructed it was formulated. It is generally a better idea to write directly to the base tables.

Example:

```
CREATE VIEW dekalb_people
    (SSN, first_name, last_name) # control the names of the
    ↪ columns as seen in the view
    AS SELECT SSN, FNAME, LNAME # control which columns are
    ↪ returned by SELECT
        FROM Person # get rows from the Person table
        WHERE ZIP = '60115'; # control which rows make it
    ↪ into the view
```

- **DROP VIEW**: Although tables and views share the same namespace (so it is not possible to have a view and a table with the same name) and work the same in a lot of queries, DROP TABLE is one of the exceptions and will not work to delete a view. It will give you an error message

  Instead, use DROP VIEW, which has generally the same syntax:

```
DROP VIEW <viewname>;
```

- **Advantages of Views**:
  - Base tables should always be designed in Third Normal Form or better. Views allow us to access them in possibly more convenient ways while still having the benefits of 3NF.
  - Views can free users from complicated DML operations, such as joins.
  - Users can be denied direct access to base tables, but given access to portions of them through the views. This enhances security

3.6.3 DML SELECT

- **SELECT Statement Format**: Two versions of the basic format of a SELECT statement follow.

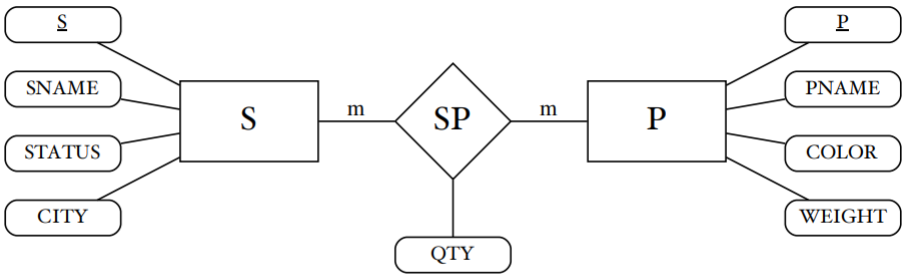
```
SELECT [DISTINCT|ALL] <column_list> # most common, show row data
    FROM <table_list>
    [ WHERE <where_exp> ]
    [ GROUP BY <group_key> ]
    [ HAVING <having_exp> ]
    [ ORDER BY <sortcols> ] ;

SELECT <anyexpression> ; # show results of the supplied expression
```

  Where

  - **<column_list>**: comma separated list of the columns to show in the results, * for all columns
  - **<where_exp>**: boolean expression evaluated once per row to determine whether the row is included
  - **<group_key>**: comma-separated list of the columns to use when grouping the rows
  - **<having_exp>**: boolean expression evaluated once per group to determine whether the group is included
  - **<sortcols>**: comma-separated list of the columns to sort by (most important comes first)
  - **<anyexpression>**: the expression whose results should be displayed

- **Example data**: Here we have a simple database to use for the examples that follow. It tracks suppliers and the parts they supply.



ER diagram representing the example database.

Supplier Info S(S, SNAME, STATUS, CITY)
Part Info P(P, PNAME, COLOR,WEIGHT)
Supplied Parts SP($S^{\dagger}$, $P^{\dagger}$, QTY).

The S table contains the information on the suppliers themselves.

S	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

The P table contains information on parts.

P	PNAME	COLOR	WEIGHT
P1	Nut	Red	12
P2	Bolt	Green	17
P3	Screw	Blue	17
P4	Screw	Red	14
P5	Cam	Blue	12
P6	Cog	Red	19

The SP table contains information on which suppliers supply which parts, and how many.

S	P	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

- **Example query**:

  Get supplier numbers and status for suppliers in Paris.

```
SELECT S,STATUS
    FROM S
    WHERE CITY = 'paris';
```

  Get part numbers for all parts supplied

```
SELECT P FROM SP;
```

Adding the DISTINCT keyword can prevent duplicate output rows from being shown.

```
SELECT DISTINC P
    FROM SP;
```

List the full details of all suppliers.

```
SELECT * FROM S;
```

List supplier numbers for all suppliers in Paris with a STATUS greater than 20.

```
SELECT * FROM S
    WHERE CITY = 'Paris' AND
        STATUS > 20;
```

- **Relational Operators in SQL**:
  - = is equal to
  - < less than
  - <= less than or equal to
  - > greater than
  - >= greater than or equal to
  - <> or ! = not equal to
- **Compound Logical Operators**:
  - **AND**
  - **OR**
  - **NOT**
- **The ORDER BY clause**: Adding the ORDER BY clause allows us to enforce a sorting order upon our results.

```
ORDER BY <attrs>
```

Where <attrs> is a comma-separated list of the attributes to base our sorting upon.

After each attribute, you have the option to add either DESC (for descending) or ASC (for ascending) to affect the sort direction for each attribute. The default sort direction is ascending, if not specified.

The first attribute listed is the most important, and any subsequent attributes is only sorted upon if there are multiple rows in which the values for the previous attributes before them were all the same.

- **ORDER BY example**: List the supplier numbers and status for suppliers in Paris in descending order of status.

```
SELECT S,STATUS FROM S
    WHERE CITY = 'Paris'
    ORDER BY STATUS DESC;
```

- **Cartesian Product in SQL**: For two sets $A = \{a, b, c\}$, and $B = \{d, e, f\}$

$$A \times B = \{(a, d), (a, e), (a, f), (b, d), (b, e), (b, f), (c, d), (c, e), (c, f)\}.$$

  This is relevant because the Cartesian Product is used in SQL when we SELECT from multiple tables. When this happens, the sets (like A and B) to be combined are the tables, and the items inside of them are the tuples/rows they contain.

  When the Cartesian Product is done on two tables,

  - The width of the result is the sum of the widths (in columns) of both of the tables.
  - The length (in rows) of the result will be the product of the lengths of both of the tables.

  Note: The Cartesian Product is an associative operation

- **Aliases and the dot operator**: When we select certain relations, we can give aliases to them and then reference their attributes with a dot (similar to how you access C++ member functions)

  This is important because if we take the cartesian product of the same relations, we need to give them aliases (they would otherwise have the same name)

- **Aliases with AS**: We can also use AS to assign aliases

```
SELECT col AS c1
    FROM relation1 AS r1
    ...
```

- **Using the dot in general**: In general, even without aliases, we can use the dot to refer to relations attributes, this is crucial if we select on two relations with matching attribute names.

- **Cartesian Product Example**: S has 5 rows of 4 columns. The Cartesian product, SELECT * FROM S T1, S T2; returns 25 rows, each with 2 sets of the columns in S, for a total of 8 columns. They don't all fit on the page here.

T1.S	T1.SNAME	T1.STATUS	T1.CITY	T2.S	T2.SNAME	T2.STATUS	T2.CITY
S1	Smith	20	London	S1	Smith	20	London
S2	Jones	10	Paris	S1	Smith	20	London
S3	Blake	30	Paris	S1	Smith	20	London
S4	Clark	20	London	S1	Smith	20	London
S5	Adams	30	Athens	S1	Smith	20	London
S1	Smith	20	London	S2	Jones	10	Paris
S2	Jones	10	Paris	S2	Jones	10	Paris
S3	Blake	30	Paris	S2	Jones	10	Paris
S4	Clark	20	London	S2	Jones	10	Paris
S5	Adams	30	Athens	S2	Jones	10	Paris
S1	Smith	20	London	S3	Blake	30	Paris
S2	Jones	10	Paris	S3	Blake	30	Paris
S3	Blake	30	Paris	S3	Blake	30	
Paris							
S4	Clark	20	London	S3	Blake	30	Paris
S5	Adams	30	Athens	S3	Blake	30	Paris

- **Cartesian product example**: For each part supplied, get the part number and names of all the cities supplying the part. (This is a join which pulls together data from multiple tables.)

```
SELECT DISTINC P, CITY
    FROM SP,S
    WHERE SP.S = S.S
```

List the supplier numbers for all pairs of suppliers such that two suppliers are located in the same city.

```
SELECT T1.S, T2.S /* one S from each side */
    FROM S T1, S T2 /* cartesian product of S with S, giving
    ↪ name to each side */
    WHERE T1.CITY = T2.CITY /* same city for both suppliers
    ↪ */
        AND T1.S < T2.S; /* avoid duplicate pairs; lower S
    ↪ on left */
```

- **Multiple-row subqueries**: Multiple-row subqueries are nested queries that have the potential to return more than one row of results to the parent query. Most commonly used in WHERE and HAVING clauses

  Note: Must be used with multiple-row operators.

- **SQL Sets**: In an SQL statement, we can denote a set with a list of values inside parentheses.

- **Multiple Row Subqueries: IN Set Operator**: IN is a set operator used to test membership.

  The IN operator will have value on its left, and a set on its right. It will evaluate to true if the value from the left hand side is present in the set provided on the right.

Example	Evaluates to
'S1' IN ('S2','S3','S1')	true
'S1' IN ('S2','S3','S4')	false
4 IN (2,1,6,4,5)	true
3 IN (1,5,6,10)	false

When a multiple-row subquery is evaluated, its results are inserted into its parent query as a set. We can use set operations like IN to fit those results into our query

- **Multiple-row subqueries: Example**: List the supplier names for suppliers who supply part P2. (This time using a subquery.)

```
SELECT SNAME
    FROM S
    WHERE S IN              # IN operator used to check
    ↪ current S against the list
        ( SELECT S          # this is the subquery
        FROM SP             # which returns a list (set)
        WHERE P = 'P2' );   # containing all the suppliers
    ↪ that supply part P2.
```

Note: The innermost subqueries are the first to run. It returns

S
S1
S2
S3
S4

Which is inserted into the parent query as ('S1', 'S2', 'S3', 'S4'), in the position where the subquery that returned the results was found.

Thus, after the subquery is run, the outer query effectively becomes:

```
SELECT SNAME
    FROM S
    WHERE S IN                  # IN operator used to check
    ↪ current S against the list
        ('S1', 'S2', 'S3', 'S4');   # <-- results of
    ↪ subquery inserted in place
```

Which would have the following results:

SNAME
Smith
Jones
Blake
Clark

- **Set Operators: ALL and ANY**: The ALL and ANY operators modify the normal relational (in the comparison sense) operators to work on sets.

If we want to compare a value with every item in the set and reduce the answers to a single true/false using the AND operation, we can use ALL

```
<value> <relop> ALL (set)
```

If we want to compare a value with every item in the set and reduce the answers to a single true/false using the OR operation, we can use ANY.

```
<value> <relop> ANY (set)
```

- **Set Operator: EXISTS**: The EXISTS operator is a unary operator working on sets that is used to determine whether the set supplied is non-empty. Once again <set> is either an explicitly written set or a multi-row subquery

```
EXISTS (set)
```

  - Evaluates to true if the set is non-empty (contains at least one element)
  - Evaluates to false if the set is empty (no elements inside)

- **Set operator: NOT EXISTS**: EXISTS, when used in conjunction with the logical inversion operator, NOT, enables two types of queries that were difficult before
  - Queries involving the set difference operation $\{a, b, c, d, e\} - \{b, c\} = \{a, d, e\}$
  - Queries that involve the concept of every

  only include rows where the subquery is EMPTY

- **Union**: The UNION operator causes two sets to be merged, the set union.

```
SELECT P
    FROM P
    WHERE WEIGHT > 18 # first SELECT returns only P6
UNION
    SELECT P
    FROM SP
    WHERE S = 'S2'; # second query returns P1, P2
```

- **Union caveat**: You should be careful in situations where the domain of a column matters, as UNION will put rows together whether the columns match in type/purpose or not.

- **Group Functions**: Group functions are sometimes referred to as aggregate or multiple-row functions. They take a list of columns as an argument, with an optional DISTINCT or ALL inside before those columns are listed.
  - **SUM(<x>)**: add up the value of column <x> in all of the rows of each group
  - **AVG(<x>)**: find the average value of column for each group

- **COUNT(<x>)**: count how many rows there are (usually <x> is a * here.)
- **MAX(<x>)**: returns the maximum value of column <x> for each group
- **MIN(<x>)**: returns the minimum value of column for each group
- **STDDEV(<x>)**: returns the standard deviation of column <x> for each group
- **VARIANCE(<x>)**: returns the variance of column for each group

All of these functions will return a single value for each group present.

If no GROUP BY clause is included, then there is only a single group, which contains all the rows of the query. The GROUP BY clause will allow that to be divided into subgroups.

- **Group function example: COUNT**: Find out the number of suppliers

```
SELECT COUNT(*) FROM S;
```

- **DISTINCT with group functions**: Get the total number of suppliers currently supplying parts

  If you want to count only distinct values, we can do that with DISTINCT

```
SELECT COUNT(DISTINCT S) FROM SP;
```

- **WHERE clause with group functions**: The WHERE clause is evaluated BEFORE any groups are formed.

```
SELECT COUNT(*)
    FROM SP
    WHERE P = 'P2'; # the value of P is known before
    ↪ grouping, so WHERE works
```

- **The GROUP BY clause**: The GROUP BY clause in a SELECT statement takes the following form:

```
GROUP BY <attrs>
```

  It will cause the SELECT statement to examine the rows in its result set, and gather the ones that match on their values for the columns in <attrs> into subgroups.

```
SELECT SUM(QTY) FROM SP
    GROUP BY P; # make a subgroups for each part

SELECT P, SUM(QTY) FROM SP # added P to be shown
    GROUP BY P; # make a subgroup for each part
```

- **Group by caveat**: However, if we try to display columns that aren't part of the <attrs> of the GROUP BY and aren't calculated by a group function, we begin to have problems.

```
SELECT P, S, QTY, SUM(QTY) FROM SP GROUP BY P; # P is good,
↪  but look at S and QTY
```

P	S	QTY	SUM(QTY)
P1	S1	300	600
P2	S1	200	1000
P3	S1	400	400
P4	S1	200	500
P5	S1	100	500
P6	S1	100	100

What do the values of S and QTY mean in this grouped context? Nothing! They are not relevant or correct. Is S1 the only supplier for all of the groups? The SP table indicates no. Is the QTY there valid for P1? No, the correct answer for total P1 supplied is in the SUM(QTY).

There is a distinct value of S and a value of QTY for every row in each subgroup. That is many values, and only one place to show them in – it's not atomic. Unfortunately the DBMS is just choosing one to show anyway, but it has no meaning, and such situations should be avoided.

- **HAVING clause**: Just as the WHERE clause could be used to filter individual rows based on whether they evaluated true for its expression, the HAVING clause allows us to filter out groups based on values that pertain to the group.

```
HAVING <expr>
```

For each group in the results, the HAVING expression, <expr> is evaluated, and only groups where <expr> is true will be included in the final output.

The reason HAVING is necessary is that the WHERE clause is evaluated BEFORE the groups are formed, and is not able to work with values that don't exist until after it has already finished.

- **Example with HAVING**: List the part numbers for all parts supplied by more than one supplier.

```
SELECT P
    FROM SP
    GROUP BY P
    HAVING COUNT(*) > 1;
```

- **Single-Row Subqueries**: Single-row subqueries are subqueries that return a single value (ONE ROW with ONE COLUMN).

Like the multiple-row subqueries, they are evaluated and then their results are used in the parent query that contained them.

They don't need to use the multiple-row operators to work.

- **Single-Row Subquery as a column**: SELECT Title, Retail, (SELECT AVG(Retail) FROM Books) third column will have result 'Overall Average' with a changed title

Title	Retail	Overall Average
The Princess Bride	39.99	42.00
The Life of Pi	3.14	42.00
The Hitchhiker's Guide	29.50	42.00
...	...	...

Having the call to the group function AVG would normally reduce the results to a single row per group, but it happened inside a subquery, so it did not change the outer query. This can be useful when you really want to know an aggregate value but don't want to condense your rows.

- **Single-Row Subquery in a WHERE clause**: Let's use a bookstore as an example. If you knew the ISBN of a book and wanted to run a query to find all of the books that are more expensive than it, you could use a subquery to find out the cost of the book with that ISBN and then compare that value with its result.

```
SELECT Title, Cost
    FROM Books
    WHERE Cost > # compare the cost of current row with result of subquery
        (SELECT Cost # only the Cost returned -- single column
        FROM Books
        WHERE ISBN = '1328948854'); # ISBN is PK -- single row
```

- **Single-Row Subquery in a HAVING clause**: Since the result of the subquery is inserted in place, it will work anywhere a single value makes sense. This includes use as part of a HAVING clause. Using the same book database from the previous slide:

```
SELECT Category,
    AVG(Retail - Cost) 'Average Profit' # calculate average profit of all books, change the label
    FROM Books
    GROUP BY Category
    HAVING AVG(Retail - Cost) > # compare cost of each group with result of the subquery
        ( SELECT AVG(Retail - Cost) # finds the average profit for books in LIT
        FROM Books
        WHERE Category = 'LIT' );
```

- **Single-Row Subquery example 1**: List the supplier numbers for suppliers who are located in the same city as supplier S1

```
SELECT S
    FROM S
    WHERE CITY = # compare each row with result of subquery
        ( SELECT CITY # find out which city S1 is in
        FROM S
        WHERE S = 'S1' );
```

- **The LIKE operator**: So far, all of the string comparisons we've done have been with the = operator, which tests for strict equality. (Locale settings determine whether it's a case sensitive or case insensitive match.)

  Using just =, we'd have to have a lot of OR's strung together to have any kind of flexibility.

  If we have a pattern to be matched, we generally won't use =, but rather the LIKE operator.

```
<val> LIKE <pattern>
```

  The LIKE operator will return true when <val> matches the pattern specified in <pattern>.

- **Patterns with LIKE**: The patterns that LIKE uses to check your values against are defined using these special characters.
  - % any zero or more characters can fit here without breaking the match
  - _ any single character can fit here without breaking the match
  - \ escape the next character
  - % escaped %, so only match the actual % character here
  - _ escaped _, so only match the actual _ character here
  - \\ escaped \, so only match the actual \character here

  Any characters not in this list will only match themselves.

- **LIKE: Character classes and union (or)**: You can specify a list or range of characters with square brackets.

```
SELECT ...
    WHERE ... LIKE "_[abc]%"
    WHERE ... LIKE "_[a-z]%"
```

- **Negating character classes:** To invert a character class, we can use !

```
SELECT ...
    WHERE ... LIKE "_[!abc]%"
```

- **List suppliers whose name starts with the letter 'S'**:

```
SELECT * FROM S
    WHERE SNAME LIKE `S%`;
```

- **Single-valued (non-group) functions**: Unlike the aggregate functions, these functions won't make your results collapse based on groups. They are evaluated, and their value is inserted in place.
  - **LOWER(<str>)**: Returns copy of <str> but all lowercase
  - **UPPER(<str>)**: Returns a copy of <str> but all uppercase
  - **SUBSTR(<str>, <pos>, <len>)**: Returns a copy of the substring of <str> starting at its <pos>th position, <len> characters long
  - **LENGTH(<str>)**: Returns the length in characters of the string, <str>
  - **LPAD(<str>, <len>, <sp>)**: Returns <str> fit into <len> characters, padding with <sp> on the left if necessary
  - **RPAD(<str>, <len>, <sp>)**: Returns <str> fit into <len> characters, padding with <sp> on the right if necessary
  - **ROUND(<num>,<pos>)**: Returns the number <num>, rounded to <pos> digits after the decimal point
  - **CONCAT(<str>, [...])** Returns a the concatenation of the strings <str>, in order.
  - **SOUNDEX(<str>)**: Returns a string containing a code that can be used to compare how <str> sounds like other strings.

  These functions can be nested however you'd like. Just like C++, they're evaluated from the inside out.

- **JOINS**: We've seen joins in some of our examples already.

  A join is an operation that takes information from separate tables and combines it into one set of results.

  There are two basic types of join:

  1. **Inner join**
  2. **Outer join**

  For either of these types of join, it is possible to join a table with itself, in which case we call it a self join.

- **Change to S**: To make things interesting in these joins, let's add a new supplier, S7, to our S table. They won't supply anything yet so leave P and SP unchanged.

  We have already done a few inner joins in the earlier examples,

- **Inner join**: With an inner join, only lines that match up with each other in both tables will be a part of the result. Earlier, we accomplished this by putting together two tables with the Cartesian Product, and then using a WHERE clause to make sure only things that matched on the foreign key were retained.

```
SELECT S.S,P,SNAME,QTY
    FROM S,SP # Cartesian product of S with SP
    WHERE S.S = SP.S; # only keep rows matching S=S
```

Can be written as

```
SELECT S.S,P,SNAME,QTY
    FROM S JOIN SP # replace the comma with the keyword JOIN
    ON S.S = SP.S; # WHERE becomes ON for the foreign key
```

- **Outer Join**: An outer join can be more flexible than an inner join. It will contain everything that the inner join contained, but one or both of the two tables involved will be special, and will have at least one row in the results whether it matched the other side or not. The values for the missing side will be filled with NULL since there is no relevant value.

  Here we have the same query from before, but as an outer join instead of an inner join.

```
SELECT S.S,P,SNAME,QTY
    FROM S LEFT JOIN SP # LEFT means table on LHS of JOIN is
    ↪ the strong one
    ON S.S=SP.S;
```

  LEFT means the table on the left-hand side of the JOIN keyword is the strong one. RIGHT would mean the RHS is strong. In some dialects of SQL, you can use FULL to make both strong, but this does not work in MariaDB. You can accomplish something similar with a UNION if needed.

- **The LIMIT Clause**: the LIMIT clause is used to restrict the number of rows returned by a query. When you specify LIMIT 50, it tells the database to return only the first 50 rows of the result set.

```
SELECT * from sometable LIMIT 50; // Queries the first 50
↪ rows
```

  Note: Redundant if the number of rows in the relation is less than the limit restriction

- **IS NULL and IS NOT NULL**: A field with a NULL value is a field with no value.

  If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

  It is not possible to test for NULL values with comparison operators, We will have to use the IS NULL and IS NOT NULL operators instead.

```
SELECT ...
    WHERE ... IS NULL;

SELECT ...
    WHERE ... IS NOT NULL;
```

- **BETWEEN operator**: the BETWEEN operator is used to filter the result set within a specified range. It works for numbers, dates, and text values.

```
SELECT ...
    WHERE ... BETWEEN val1 AND val2;
```

We can of course negate this with NOT

```
SELECT ...
    WHERE ... NOT BETWEEN val1 AND val2;
```