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:**

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

  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

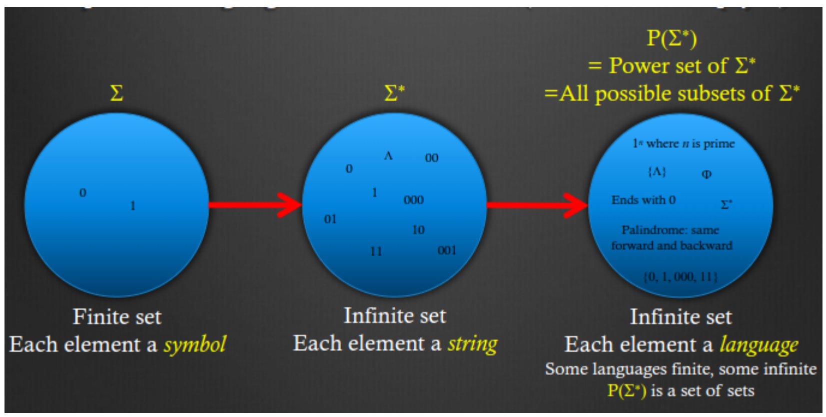
- **Recall: Power set**: The power set of any set S, written $\mathcal{P}(S)$ is the set of all subsets of S, including the empty set and the set S itself.

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.

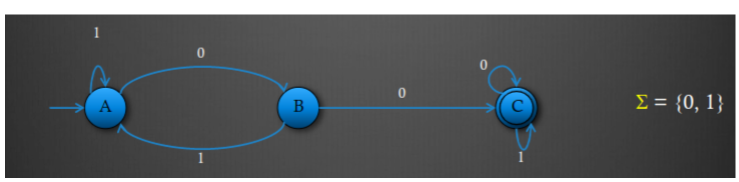
1.2 Regular languages

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.

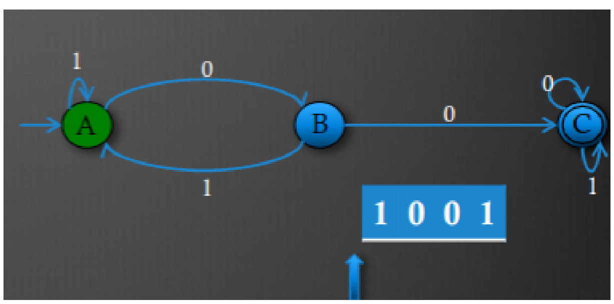
- **How to use an FA**: The purpose of a FA is to define a language over its alphabet Σ. The FA provides the means by which to test a candidate string from Σ and determine whether or not the string is in the language. It does this by "writing" the candidate string on an fictitious input tape and proceeding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

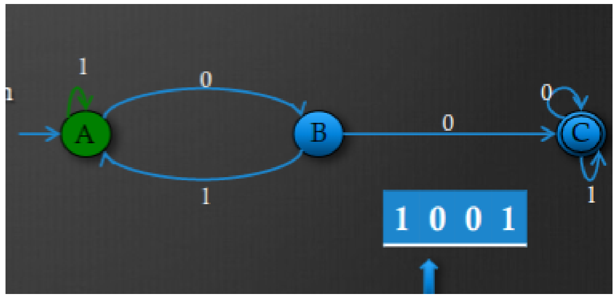
- **Using the previous FA**: Let's now try to use our FA to test whether or not the string 1001 is in the language

  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

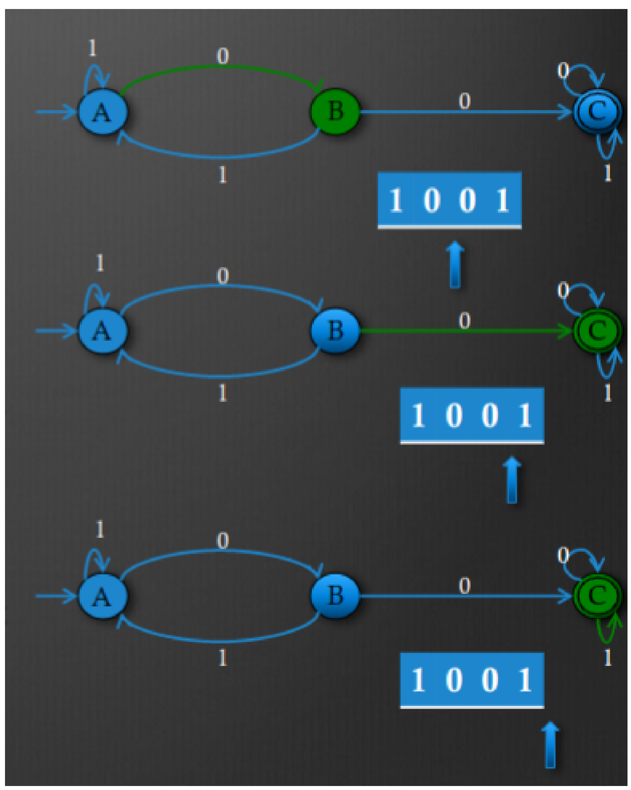


In this case, we started in state A, read symbol 1, and followed the edge labeled 1 from A which brought us back to A

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

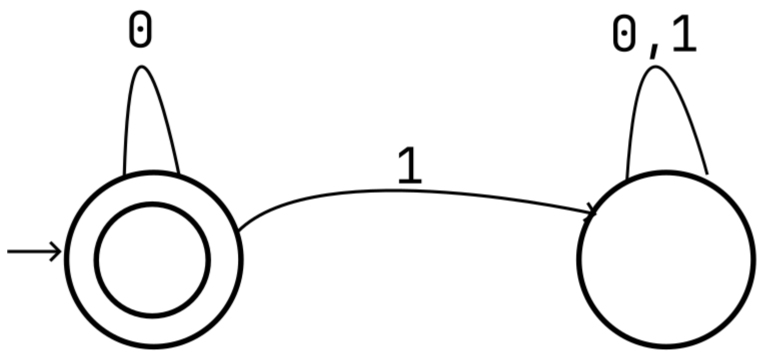
Once the tape head reaches the end of the tape we simply look to see whether or not the FA ended in an accepting state.

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



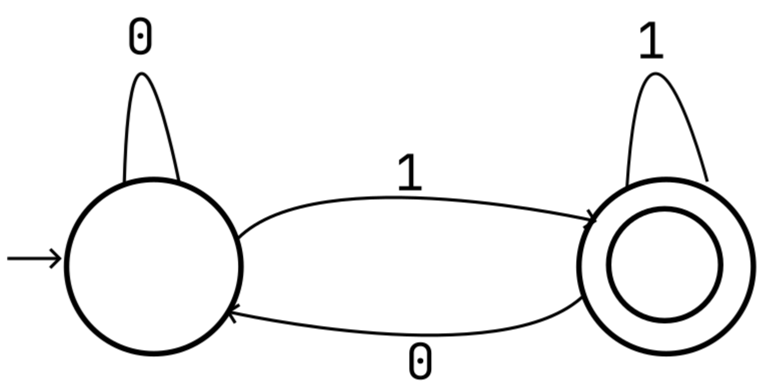
We deduce that the language has only strings with two consecutive zeroes somewhere.

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

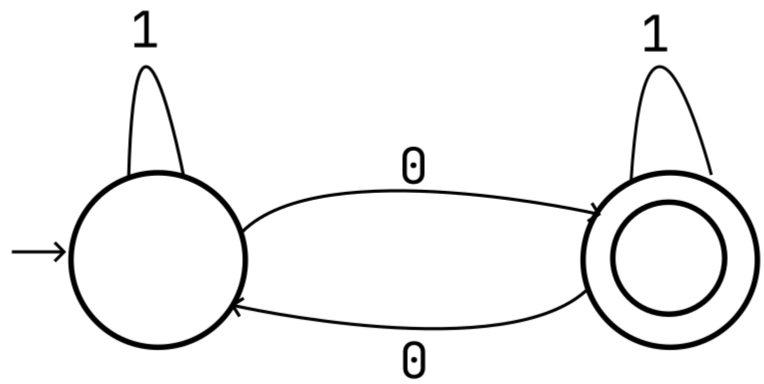


This one is pretty simple. If we have a zero, stay in the accepting state, if we see a one, toss it to the other non-accepting state, its not coming back.

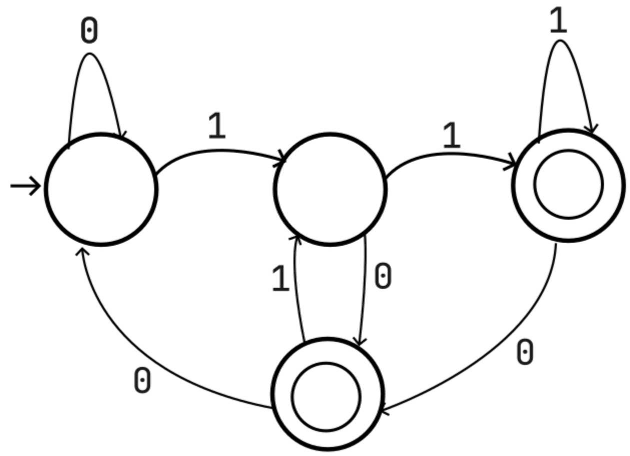
- **FA Example Three: The set of all strings that end in one** ($\Sigma = \{0, 1\}$):



- **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:

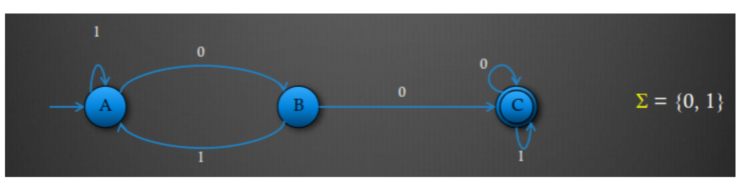
- **Finite State:** A finite automaton has a finite number of states. This means it can only "remember" a limited amount of information about the input it has processed. Once a finite automaton transitions to a new state, it forgets all previous information except for the current state.
- **No Stack 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*: $\quad Q = \{A, B, C\}$
  - Σ is an alphabet of *input symbols*: $\quad \Sigma = \{0, 1\}$
  - $q_0 \in Q$, is the *start state*: $\quad q_0 = A$
  - $T \subseteq Q$, is the set of *accepting states*: $\quad T = \{C\}$

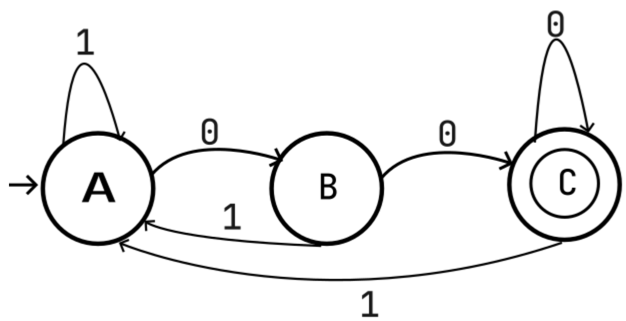
- δ is the *transition function* $\delta : Q \times \Sigma \to Q$

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

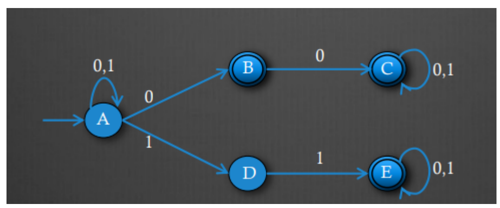
δ	0	1
A	B	A
B	C	A
C	C	A

.

1.2.3 nondeterministic Finite automata (NFA)

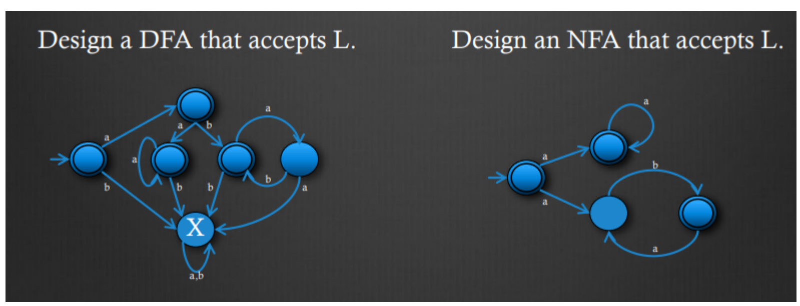
- **NFA definition**:
  - If an automaton gets to a state where there is more than one possible transition corresponding to the symbol read from the tape, the automaton may choose any of those paths. (nondeterminism) We say it **branches**
  - if an automaton gets to a state where there is no transition 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 $\Sigma = \{0, 1\}$



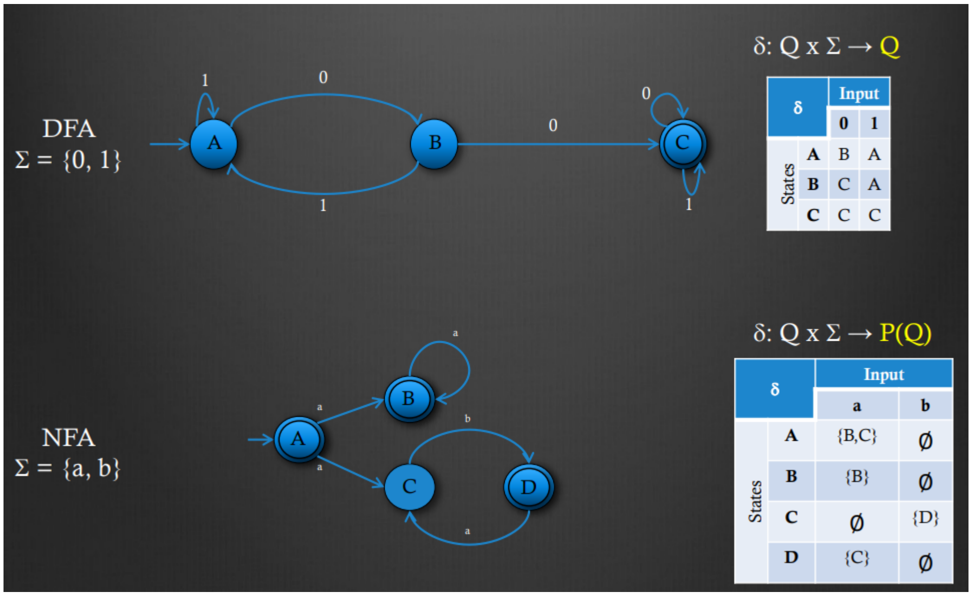
- **DFA or NFA?**: Consider the language L over $\Sigma = \{a, b\}$ which is defined by

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



- **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**:

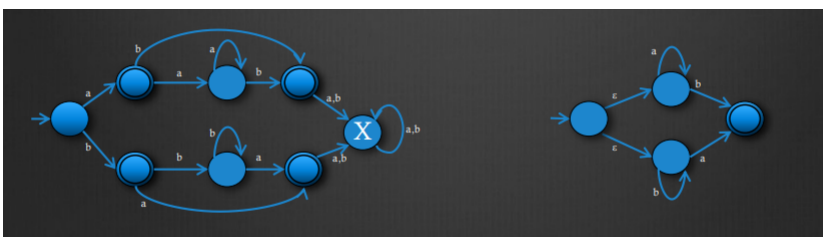


- **NFA with ϵ-transitions**: ϵ-transitions allow the automaton to change state without consuming an input symbol

  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

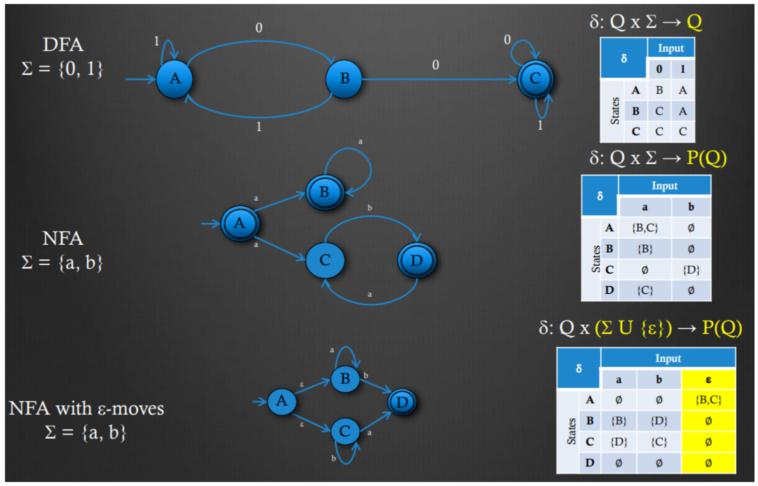
$$L = (b^*a) + (a^*b).$$



- **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

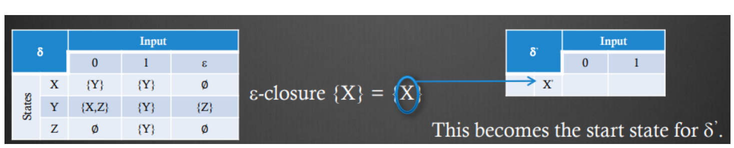
  Things to note about the conversion:

  - 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:

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

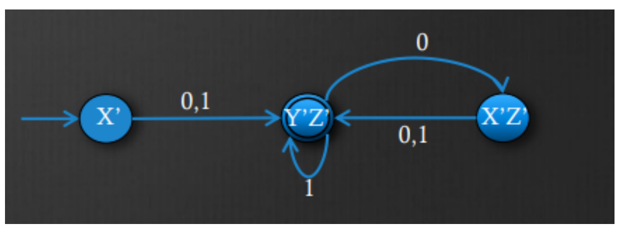
and since $X'Z'$ is a new state, we add it to δ'.

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,

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

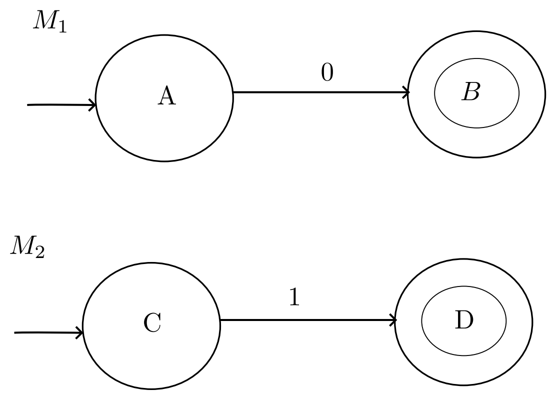
  3. **Define the transition function**: The transition function δ' of the new NFA will include:
     - All the transitions of M_1 and M_2
     - Two ϵ transitions from the new start state to the start states of the two original machines q_{0_1} and q_{0_2}. Thus,

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

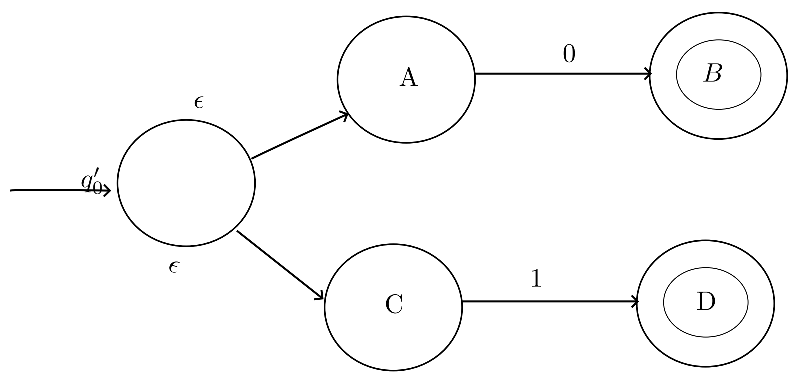
  4. **Define the set of accepting states**: The set of accepting states will be

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

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**

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

- **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 (·)

  - **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

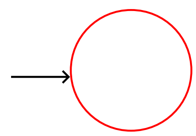
1. There must be exactly one accepting state.
2. No transitions (not even ϵ-moves) may leave the one accepting state.

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

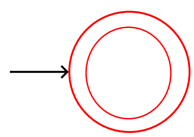
1. Adding a new accepting state.
2. Add an ϵ-move from each of the original accepting states to the newly added accepting state.
3. Convert all of the original accepting states to non-accepting states.

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

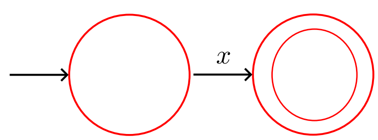
$\varnothing$:



$\{\epsilon\}$:



$x \in \Sigma$:



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

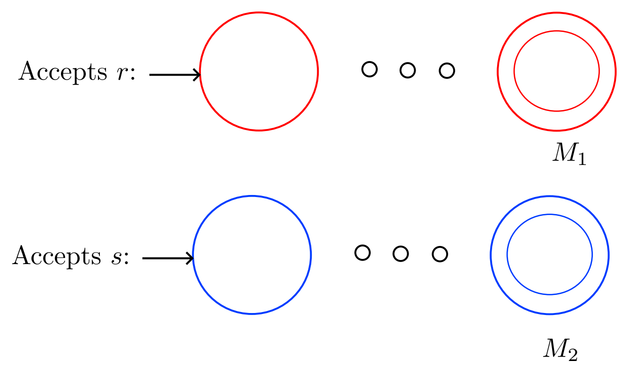
1. $r + s$ is a regular expression denoting the set $R + S$, (i.e,. union of languages),
2. rs is a regular expression denoting the set RS (i.e., concatenating languages), and
3. $r*$ is a regular expression denoting the set R^* (i.e., Kleene closure of a language).

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

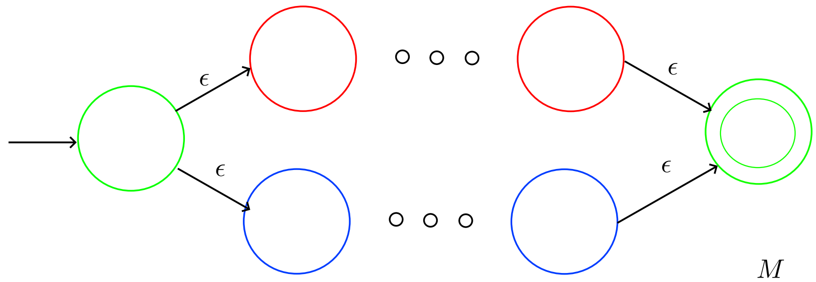
We then use these machines M_1 and M_2 to create a new machine M that accepts $r + s$

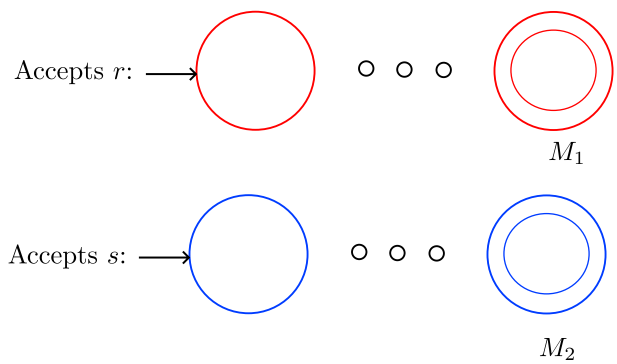
So what did we do here

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.

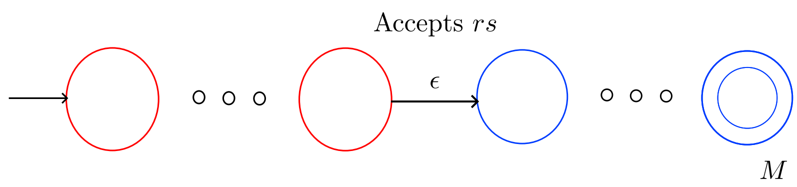
Accepts r or s $(r + s)$



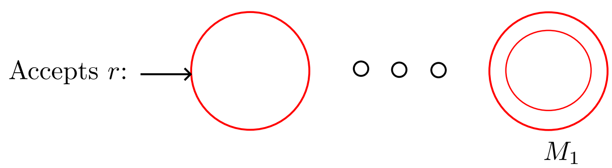


We use M_1 and M_2 to construct new NFA with ϵ-move M that accepts rs.

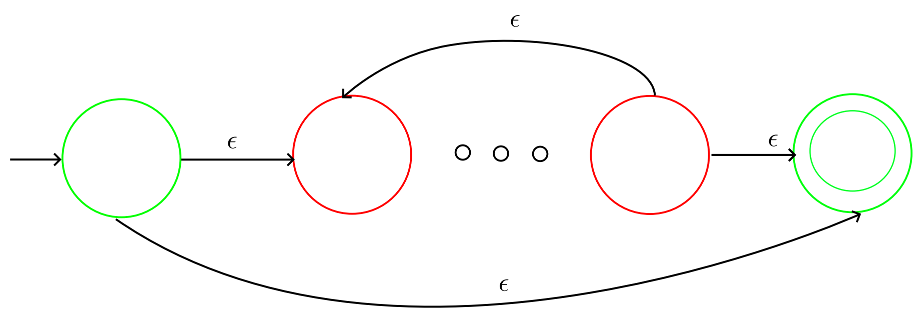
1. Add an ϵ-move from M_1's accepting state to M_2's start state.
2. Change M_1's accepting state to a nonaccepting state.



- **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^*

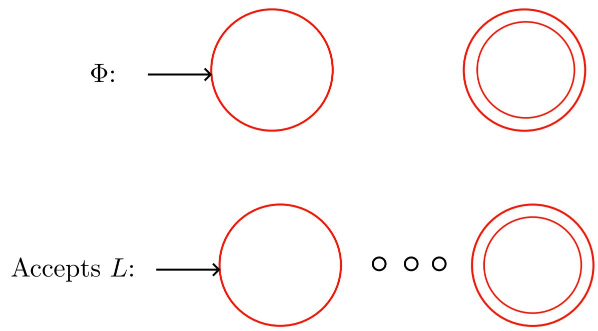


1. Create new start and accepting states.
2. Add ϵ-move from new start to M_1 start, M_1 accepting to new accepting, and new start to new accepting.
3. Add ϵ-move from M_1 accepting to M_1 start.
4. Change M_1's accepting state to a non accepting state.

- **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;

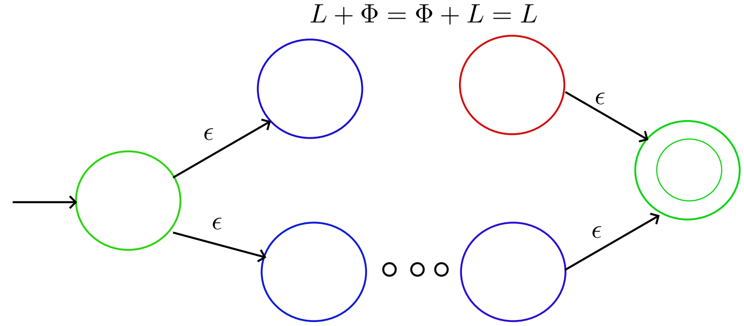
  1. $\Phi L = L\Phi = \Phi\Phi = \Phi \ \forall L$
  2. $L + \Phi = \Phi + L = L$
  3. $\Phi + \Phi = \Phi$
  4. $\Phi^* = \{\epsilon\}$

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

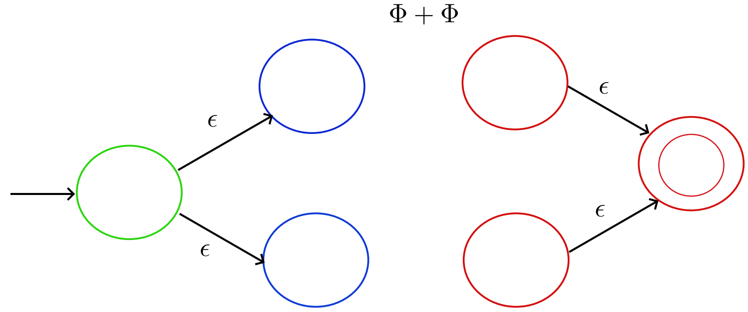


1. Confirming $L + \Phi = \Phi + L = L$ and $\Phi + \Phi = \Phi$:

$L + \Phi = \Phi + L = L$

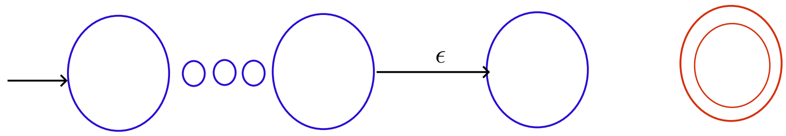


$\Phi + \Phi$

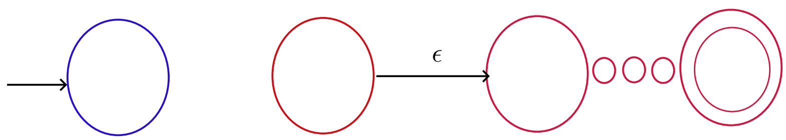


2. Confirming $\Phi L = L\Phi = \Phi\Phi = \Phi$

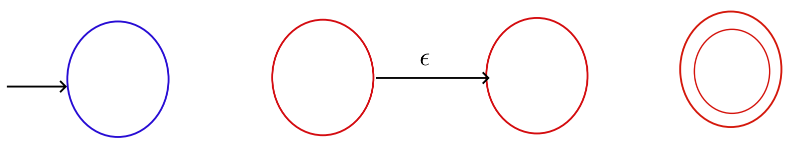
$L\Phi$



ΦL

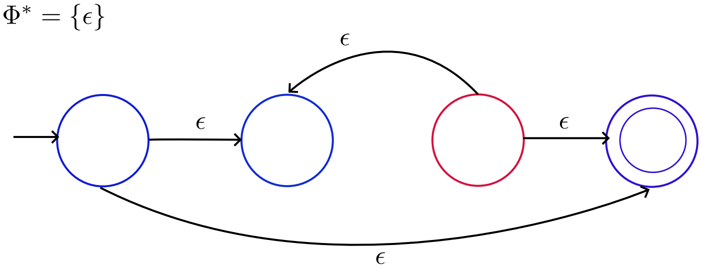


$\Phi\Phi$



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

$\Phi^* = \{\epsilon\}$



- **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.

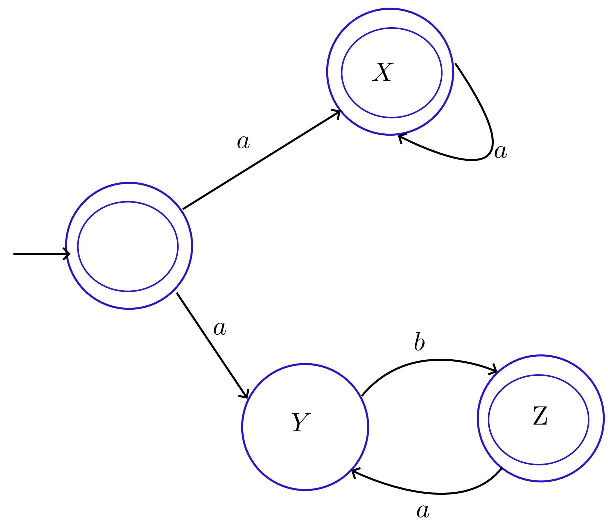
Some notes:

(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

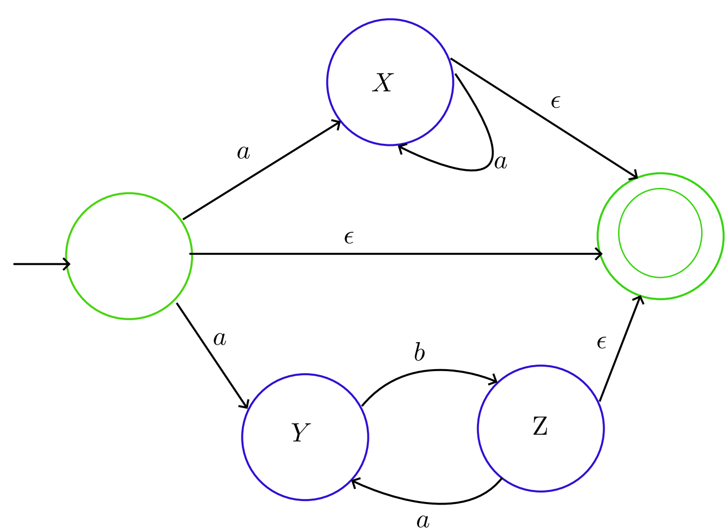
$$\text{New RE transitions} = N(\text{outgoing}) \times N(\text{incoming}).$$

Not including the loops

Example: Consider the NFA-ϵ

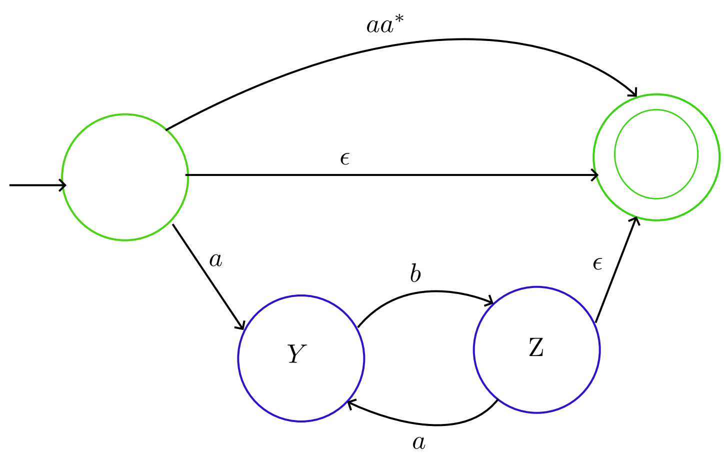


Before we begin eliminating states, we see that this FA does not obey the two constraints described above. So, we create a new accepting state such that there is only one accepting state. Each old accepting state has ϵ transitions to this new accepting state. This FA has no incoming transitions to the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

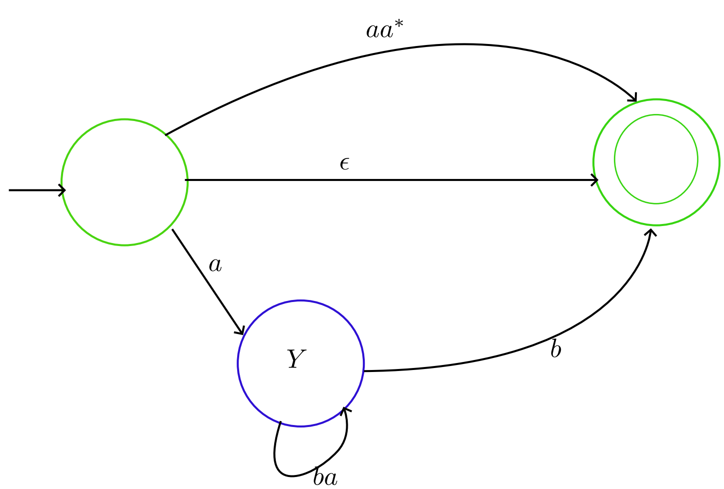


Now, we start eliminating states one at a time. We recall that it does not matter the order in which we eliminate them.

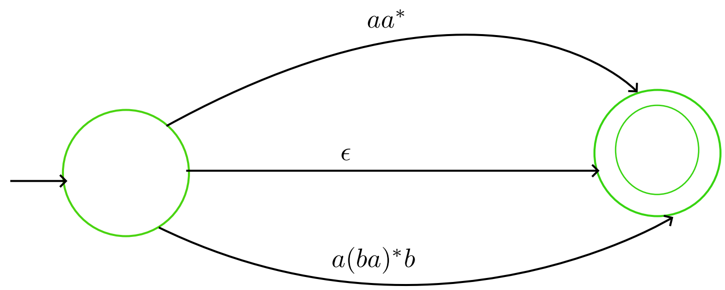
We start by eliminating state X. There is one incoming transition and one outgoing transition. Thus, there is $1 \times 1 = 1$ new transition. To get from the start state, through X, to the accepting state, the regular expression is $aa^*\Lambda = aa^*$. Thus,



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,

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

- **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

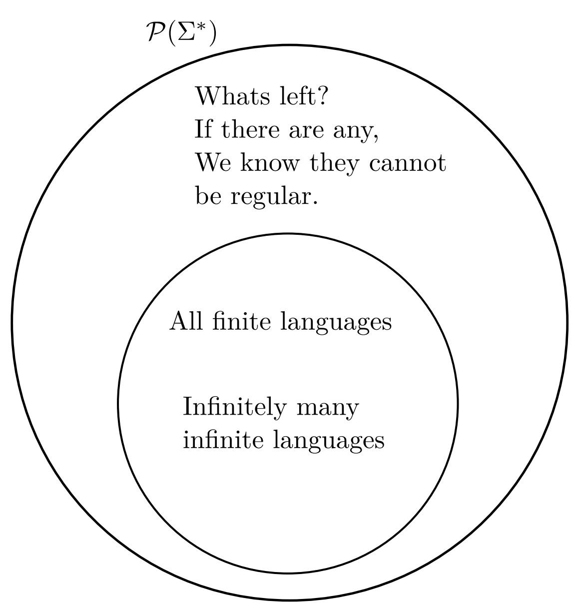
- **Intro to nonregular languages**: Suppose we have some alphabet Σ, we can then find Σ^*, which is the language consisting of all possible strings (including the empty string ϵ) using the symbols from Σ. The we can take the power set of Σ^* to get the set of all possible subsets of Σ^*. The alphabet 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. $\blacksquare$
  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.

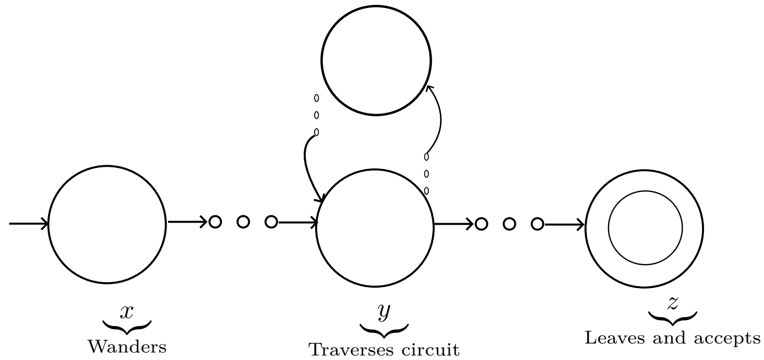
  Given that circuit and because we also selected $w \in L$, we note that we can modify w to create a new word w' pumping into w as many symbols as are needed and in just the right location in w that would cause M to traverse the circuit one more time than it did when processing w. We note that the new 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 $xy^nz \in L \ \forall \ n > 0$. 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 $xy^nz \in L \ \forall \ n > 0$. 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

  $$a \to b \not\Longrightarrow b \to a.$$

  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 $xy^iz \in L$ 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 $xy^iz \in L \ \forall \ i > 0$.
  - 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 $\blacksquare$

- **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. $\blacksquare$

- **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. $\blacksquare$.

- **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 $\blacksquare$

- **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. $\blacksquare$

- **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

- A **Context Free Grammar** (CFG) is a 4-tuple (V, Σ, S, P), where
  - V is a non-empty finite set of *variables*
  - Σ is a finite alphabet of *terminals*
    (we assume V and Σ are disjoint)
  - $S \in V$, is the *start variable*
  - P is a finite set of *productions* of the form $A \to \alpha$ where
    * A is a variable (i.e., $A \in V$) and
    * α is a string of symbols from $(V \cup \Sigma)^*$ (i.e., $\alpha \in (V \cup \Sigma)^*$).
- **CFG notational convention**: Variables are upper case letters with S always being the start variable.

  Terminals are lower case letters, symbols, or constants, including ϵ to denote the empty symbol.
- **A simple CFG**: Consider the following CFG that has two productions

$$S \to 0S1$$
$$S \to \epsilon.$$

- **Derivations**: We say that a finite string w, consisting only of terminals, is generated by a CFG if, starting with the start variable S, you can apply a sequence of productions that result in w.

  The sequence is called a derivation of w.
- **Derivation examples**: All derivations must start with the start variable S.

  As long as there is at least one variable in our string we must continue the derivation by

  1. Selecting a variable from our string,
  2. Selecting a production whose left side matches the variable we selected from our string, and
  3. Replacing the variable in our string with the right side of the production we selected.

  With the two productions

$$S \to 0S1$$
$$S \to \epsilon.$$

  We select the first production and apply it

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 00\epsilon 11 = 0011.$$

  This yields the string $w = 0011$. Thus, we have derived this string from the grammer. And we also note that there are some other strings for which there is no derivation using our CFG

In short, we observe that a CFG defines a language over its alphabet of terminals Σ.

- **Context Free Languages**: A language is a context free language if it is generated by some context free grammar.
- **Is palindrome context free?**: Recall that the language Palindrome (i.e., the set of all strings that read the same forward as they do backward) is a non-regular language.

  We can create a grammer for this language

$$
\begin{aligned}
&S \to aSa \\
&S \to bSb \\
&S \to a \\
&S \to b \\
&S \to \epsilon.
\end{aligned}
$$

  The CFG builds the string from the middle always pushing outward by inserting the same pair of characters each time.

- **Notational convenience**: We can express the grammer above simply as

$$S \to aSa \mid bSb \mid a \mid b \mid \epsilon.$$

  Where the pipe deliminates each production. Note that they are still separate productions, but since they share the same variable S, we can put them on the same line separated by a pipe.

- **Connection between regular and context free languages**: We assert that every regular language is generated by a context free grammar.

  Proof: There is a recursive algorithm that converts any regular expression to a context free grammar that accepts the same language.

  – If the overall language is the union of two or more pieces, then you can start with $S \to A_1|A_2|...|A_n$
  – If the overall language is the concatenation of two or more pieces, then you can start with $S \to A_1A_2 \cdots A$
  – If the overall language is the Kleene star of a piece, say generated by E, then you can start with $S \to ES \ \mid \ \epsilon$
  – Recursively and similarly generate productions for variables until you are left with only terminals.

  Example: Consider the regular expression $(11 + 00)^*11$

  At its top level it is a concatenation of $(11+00)^*$ and 11, so we start with the following production:

$$S \to AB$$

  We continue by generating productions for each new variable we introduced A and B.

A is a Kleene Star and B is straightforward so we add the following productions:

$$A \to CA \mid \varepsilon$$

$$B \to 11$$

All that remains is production for the variable C which is a union, so we add this final production.

$$C \to 11 \mid 00$$

Thus, all regular languages are context free, but not all context free languages are regular, we know that there exist some nonregular languages that are context free, like palindrome.

- **Left and rightmost derivations**:
  - **Leftmost derivation:** At each stage one replaces the leftmost variable.
  - **Rightmost derivation:** At each stage one replaces the rightmost variable.
- **Derivation (parse) trees**: It is sometimes useful to display derivations as trees called derivation trees or parse trees. Where a parse tree satisfies
  - The root of the tree is always the CFG start symbol S
  - All of the internal nodes (i.e., non-leaf nodes) are variables.
  - All of the leaf nodes are terminals.
  - The children of a node are ordered left to right and appear in the same order as they do in the right-hand-side of a production whose left-hand-side matches variable in the parent node.
  - The word generated by the derivation tree is the sequence of terminals read from the leaf nodes in left-to-right order.
- **Ambiguous parsing**: If for some words in a language there are more than one parse trees, we say it has ambiguous parsing
- **Ambiguous grammer**: A context free grammar (CFG) defining some language L is said to be an ambiguous grammar iff there exists some word $w \in L$ that has two different derivation trees.

  Because each derivation tree represents unique leftmost and rightmost derivations;

  - a CFG is ambiguous iff there exists some word $w \in L$ for which there are two different leftmost derivations or two different rightmost derivations.

  A CFG that is not ambiguous is said to be an unambiguous grammar

1.4.1 Pushdown automata (PDA)

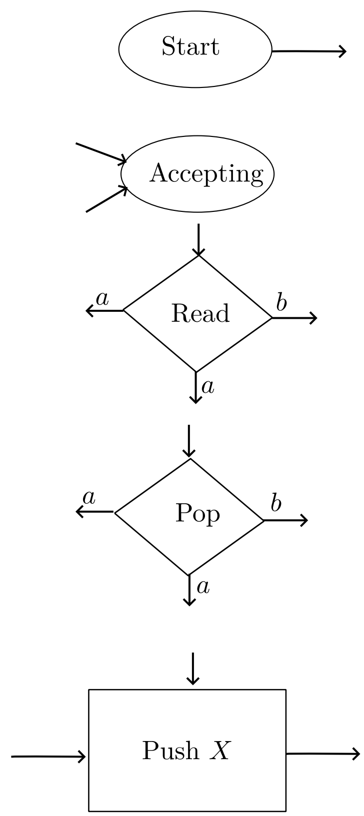
- **Intro to PDA**: PDA's are an abstract machine that is used as an acceptor for CFG's, similar to how FAs were used as acceptors for regular languages.
- **Comparison to NFA**: PDAs are closest to NFA's with ϵ-moves in that they both:
  - are nondeterministic (need only one set of choices to accept a string),
  - may crash (i.e., not every symbol must leave each state), and
  - allow for ϵ-moves

  Recall that NFA with ϵ-moves also:
  - Have finitely many states, some of which are accepting states.
  - Read string from an input tape, must read entire string to accept.
  - Reject by reaching the end of string while not in an accepting state.

  All the same is true for pushdown automata with the following minor revisions and one important addition:
  - [revision] There is only one accepting state, once entered accept string (whether the entire input string has been read or not).
  - [revision] There is no notion of reaching the "end of the string".
  - [addition] There is a stack of infinite capacity.
- **The Tape and Stack**: Pushdown automata have a stack of infinite capacity, initially empty.

  Because pushdown automata have both an input tape and stack they must distinguish which they are manipulating in each of their states; read, push, or pop.

- **State Symbols**:
  - **Start state**: No incoming edges, one outgoing edge
  - **Accepting**: state No outgoing edges
  - **Read**: (nondeterministic)
  - **Pop**: (nondeterministic)
  - **Push**: One outgoing edge



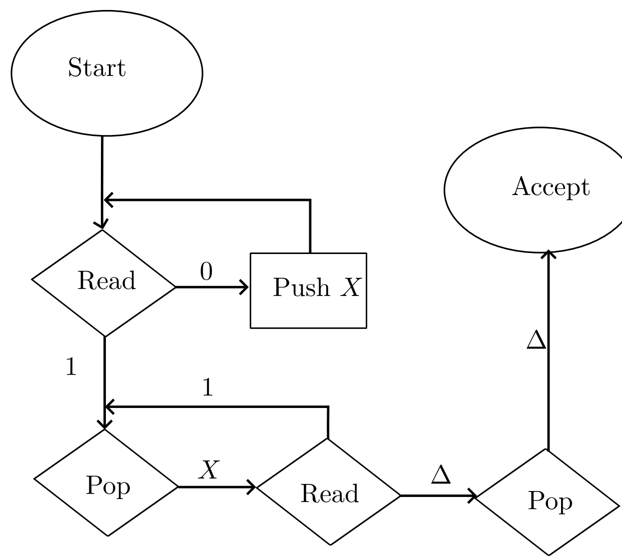
- **Initialization**: Like finite automata, we start pushdown automata by writing the string onto the input tape. Recall that the input tape has a beginning and an infinite capacity.

  In pushdown automata, after we write the string onto the input tape we append an infinite sequence of a special symbol Δ to denote the end of the string (i.e., assumed that Δ is not part of the input alphabet).

  We initialize the pushdown automaton's stack with an infinite supply of the special symbol Δ.

  We place the pushdown automaton in its START state

- **PDA Example**: Consider the PDA that accepts the language $0^n 1^n \; \forall \; n > 0$



It starts by reading all of the leading 0's and using the STACK to "count" how many 0's have been read (i.e., by pushing an x for each 0 that was read).

Once we encounter our first 1, we enter a different part of the PDA in which we confirm there is an x on the stack for every 1 we read from the tape.

If we READ a 1 and then pop something other than a x, then the PDA crashes in the POP state and rejects the string. This happens when there are more 1's than 0's.

Once we encounter our first Δ then we have reached the end of the input string and we are assured that there are at least as many 1's as there are 0's.

We still must POP the stack to confirm that there are no x's. If there was an x on the STACK then there were more 0's than 1's.

1.4.2 Chomsky Normal Form

- **Intro**: Noam Chomsky showed that it is possible to convert any CFG (i.e., all productions of the form $A \to v$ to another CFG that accepts the same language where all the productions are either

$$\begin{aligned} &A \to \text{ exactly two variables} \\ &\text{or} \\ &A \to \text{ one terminal (the terminal cannot be } \epsilon). \end{aligned}$$

  Which converts the CFG into Chomsky Normal Form (CNF)

  The conversion is done in four steps which must be in this sequence:

  1. Eliminate null productions
  2. Eliminate unit productions
  3. Almost CNF
  4. CNF

- **Eliminate Null Productions**: A null production is of the form $A \to \epsilon$.

  We start our conversion to Chomksy Normal Form (CNF) by first eliminating all null productions and replacing each eliminated null production with other new production(s).

  The basic idea is that the new production(s) we are adding to replace a null production $N \to \epsilon$ come from other productions where N appears on the right.

  If such a production exists, then we add new productions that have all possible subsets of N deleted, for example:

$$X \to aNb \quad \text{adds one production:} \quad X \to ab$$

$$X \to aNbNc \quad \text{adds three productions:} \quad X \to abNc \mid aNbc \mid abc$$

  If N appears k times in a production then add $2^k - 1$ new productions (i.e., every possible combination to remove N).

  How do we remove all null productions from a grammar? One possible strategy is to remove each one at a time ... but that can be problematic.

- **Nullable variable**: The solution is to eliminate all the null productions at the same time, but that requires a new definition.

  Given a context free grammar and a variable N in that grammar, we call N a nullable variable iff:

  1. There is a production $N \to \epsilon$ or
  2. There is a derivation that starts with N and leads to ϵ

- **Identify all nullable variables**: How do we identify all of the nullable variables in a context free grammar? By essentially taking a transitive closure.

  1. For every null production, "paint" the variable that appears on the left side of the production "red". These are all nullable variables.

2. "Paint" every occurrence of every nullable variable "red" throughout the entire grammar, including occurrences that appear on the right side of productions.
3. "Paint" any variable that appears on the left side of a production "red" if the right side of the production is all "red". These are nullable variables.
4. Repeat steps 2 and 3 until you have painted nothing else "red".

- **Eliminate the null productions**: How do we eliminate all null productions from a grammar?

  1. Identify all the nullable variables.
  2. Remove all the null productions.
  3. For every production that has a nullable variable in its right side, add new productions all with the same left side and new right sides with every possible subset of the nullable removed, but do not allow a new null production to be produced, even if the only symbol on the right side of the production is a nullable variable (i.e., do not add $Y \to \epsilon$ even if there is production $Y \to A$ and A was found to be nullable).

- **Eliminating null productions example**: Consider the following grammar

$$
\begin{aligned}
&S \to XY\\
&X \to Zb\\
&Y \to bW\\
&W \to Z\\
&Z \to AB\\
&A \to aA|bA|\epsilon\\
&B \to Ba|Bb|\epsilon.
\end{aligned}
$$

  The nullables are W, Z, A, B and the two null productions are $A \to \epsilon$ and $B \to \epsilon$

  We have identified the nullables. Now, we remove the null products $A \to \epsilon$, $B \to \epsilon$

$$
\begin{aligned}
&S \to XY\\
&X \to Zb\\
&Y \to bW\\
&W \to Z\\
&Z \to AB\\
&A \to aA|bA\\
&B \to Ba|Bb.
\end{aligned}
$$

  Finally, for every production that has a nullable variable in its right side, add new productions to the left side with every possible subset of the nullable removed.

$$
\begin{aligned}
&S \to XY\\
&X \to Zb \ | \ b\\
&Y \to bW\\
&W \to Z\\
&Z \to AB \ | \ A \ | \ B\\
&A \to aA \ | \ bA \ | \ a \ | \ b\\
&B \to Ba \ | \ Bb \ | \ a \ | \ b.
\end{aligned}
$$

- **Eliminate unit productions**: A production of the form *one variable* $\to$ *one variable* is called a unit production.

Given a context free grammar with no null productions, it is possible to create a new context free grammar that accepts the same language and that has no unit productions.

For every pair of variables A and B, if the CFG has a unit production $A \rightarrow B$ or if there is a derivation from A to B,

$$A \Rightarrow X_1 \Rightarrow \cdots \Rightarrow X_n \Rightarrow B$$

where each X_i is a single variable, then we introduce the following productions according to the following rule:

If all the non-unit productions from B are

$$B \rightarrow S_1 \mid S_2 \mid \cdots \mid S_m$$

then add the productions

$$A \rightarrow S_1 \mid S_2 \mid \cdots \mid S_m$$

Conclude by removing all unit productions from the CFG.

Consider the following grammar

$$S \rightarrow A \qquad A \rightarrow a \mid B \mid b \mid CB \rightarrow b \mid C \mid AAa \qquad C \rightarrow A.$$

Split the grammar into unit productions and non-unit productions as follows:

$$\begin{array}{ll} S \rightarrow A & \\ A \rightarrow B \mid C & A \rightarrow a \mid b \\ B \rightarrow C & B \rightarrow b \mid AAa \\ C \rightarrow A. & \end{array}$$

We must consider each unit production, one at a time, and make longer and longer derivation chains that end in a single variable without introducing any cycles (e.g., do not allow $A \Rightarrow B \Rightarrow ... \Rightarrow B$)

New grammar with unit productions removed and new productions added

$$\begin{array}{l} S \rightarrow a|b|AAa \\ A \rightarrow a|b|AAa \\ B \rightarrow a|b|AAa \\ C \rightarrow a|b|AAa. \end{array}$$

- **Almost CNF**: If L is a language generated by some CFG then there is another CFG that generates all the non null words in L all of whose productions are of one of two basic forms:

  $$\textit{Variable} \rightarrow \textit{string of only variables} \text{ or } \textit{Variable} \rightarrow \textit{one terminal}$$

  Start with a CFG having variables $S, X_1, \ldots, X_n$.

  For each terminal (e.g., a, b, c) add a new production by introducing a new variable:

  $$A' \rightarrow a, \quad B' \rightarrow b, \quad C' \rightarrow c$$

  In all of the *original* productions (i.e., only those that started with $S, X_1, \ldots, X_n$) replace the terminals *(not necessarily every terminal)* with the newly created associated variable.

We only rewrite productions with these new productions if the terminal in the production is not alone. For example $A \to Bc$ turns to $A \to BC'$, but $A \to c$ stays the same.

- **Chomsky Normal Form**: Going from Almost CNF to CNF is easy

  For every production of the form $A \to X_1 X_2 \dots X_n$ where $n > 2$:

  - Introduce a new variable R
  - Add new production $R \to X_2 \dots X_n$
    - The new production has $n - 1$ variables on the right side.
    - Because the original production had $n \geqslant 2$ variables, this new production will have $n \geqslant 2$ variables (i.e., cannot introduce unit productions) on the right side.
  - Re-write the original production $A \to X_1 X_2 \dots X_n$ as $A \to X_1 R$

  Repeat Step 1 as necessary, including applying it to any new productions that were generated by Step 1.

  Example:

$$A \to BCDE.$$

  Becomes

$$\begin{aligned} A &\to BR_1 \\ R_1 &\to CR_2 \\ R_2 &\to DE. \end{aligned}$$

- **Note about CNF**: A CFG converted to CNF will accept the exact same language *except* the empty string ϵ. If the original grammer accepted the empty string, the CNF grammer will not. If the original grammer did not accept the empty string, then the grammer in CNF will accept the exact same language.

1.5 Equivalence of Context-Free Grammars and Pushdown Automata

- **Recall**: A language is generated by a context free grammar if and only if it is accepted by a pushdown automaton.

- **Equivalence theorem**: The set of languages that can be defined by context free grammars – what we call the set of context free languages – is exactly the same set of languages that can be defined by pushdown automata.

  Claim: There is no language that can be defined by a CFG for which there is no PDA that accepts the same language.

  There is no language that can be defined by a PDA for which there is no CFG that accepts the same language.

- **Equivalence theorem proof**: The proof is in two parts and is similar in structure to the way we proved Kleene's Theorem – by constructive algorithm.

  1. Given any CFG that accepts some language L, then there is an algorithm to convert the CFG to a PDA that accepts the same language L.
  2. Given any PDA that accepts some language L, then there is an algorithm to convert the PDA to a CFG that accepts the same language L.

- **Converting CFG to PDA**: We can convert any CFG to another CFG that is in Chomsky Normal Form (CNF) that accepts the same language, most of the time; there is one noted exception (the empty string ϵ).

  Recall that CNF requires that every production be in one of following two forms: *Variable* $\rightarrow$ *exactly 2 variables* or *Variable* $\rightarrow$ *one terminal*

  Consider a CFG that defines language L_o and its conversion to another CFG in CNF that defines language L_c,

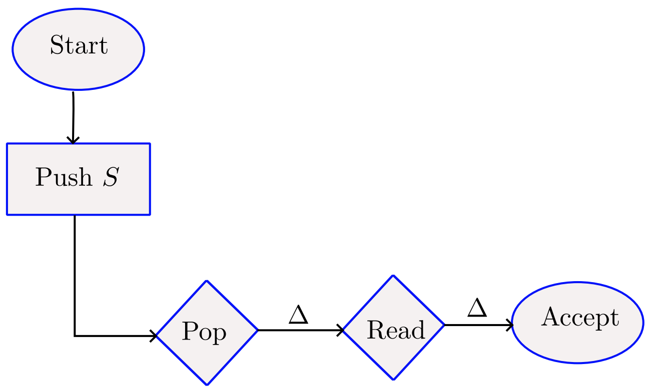
  $$\text{If } \epsilon \in L_o \text{ then } L_c = L_o - \{\epsilon\}, \text{ otherwise } L_c = L_o.$$

  It is a simple matter to determine if the empty string Λ is in the original language;

  $$\epsilon \in L_o \text{ if and only if the start symbol } S \text{ in the original CFG is nullable.}$$

  To convert any CFG to a PDA that accepts the same language we start by converting the CFG to CNF and taking note if we lost the empty string ϵ in the conversion.

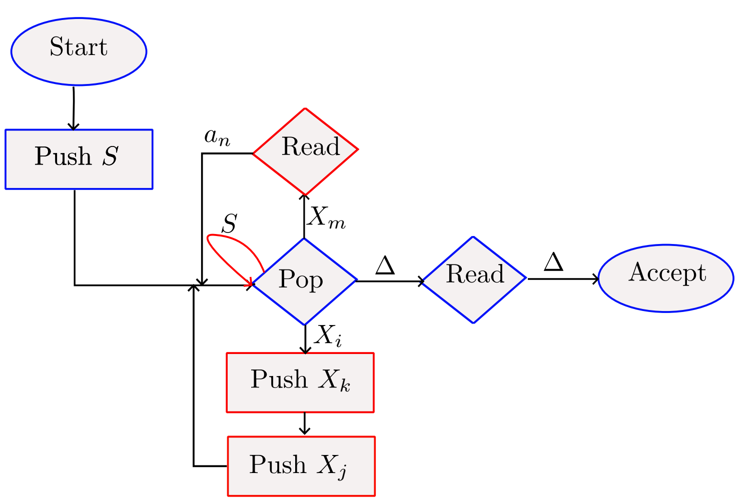
We then start building our PDA, which always starts with the same five states (i.e., no matter what the CFG).



For every production in the CNF CFG of the form $X_m \to a_n$ we add a transition from the POP state and a READ state.

For every production in the CNF CFG of the form $X_i \to X_j X_k$ we add a transition from the POP state and two PUSH states, first PUSH X_k and then PUSH X_j

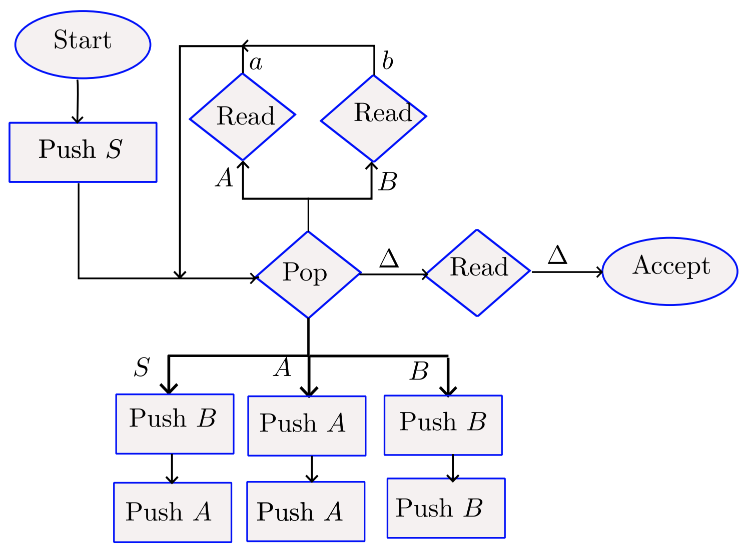
If the language defined by the original CFG accepted the empty string L, then we add one more transition labeled S from the POP state back to the POP state.



- **Example: Convert CNF CFG to PDA**: Consider the following CFG in CNF that accepts $a^n b^m$ for $n, m > 0$.

$$
\begin{aligned}
S &\to AB \\
A &\to AA \mid a \\
B &\to BB \mid b.
\end{aligned}
$$

Which converts to the PDA



We start building our PDA with the required five states.

We add the READ states for the two productions that have terminals on their right side.

We add the PUSH states for the three productions that have two variables on their right side.

Since the empty string ϵ is not in this language we do not have to add one more transition labeled S from the POP state back to the POP state, so we are done.

- **Conversion between PDA and CFG**: Given an arbitrary PDA there exists an algorithm to convert it to a CFG that accepts the same language.

The algorithm is complex, and so, we will take it on faith that it exists and that it is correct.

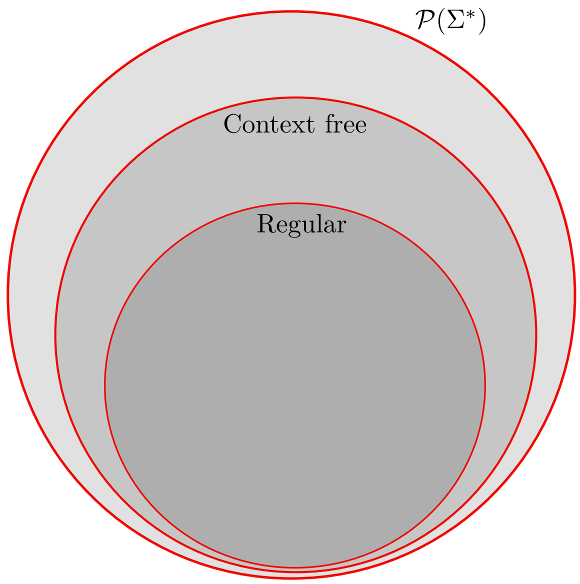
- **Equivalence theorem**: We accept the theorem

A language is generated by a context free grammar if and only if it is accepted by a pushdown automaton.

- **Current state of $\mathcal{P}(\Sigma^*)$**: Regular languages are accepted by regular expressions and finite automata.

Context free languages are accepted by context free grammars and pushdown automata.

Don't forget ... CFGs and PDAs can also accept all regular languages.



Where the regular languages are accepted by regular expressions and finite automata, and the context free languages are accepted by context free grammers and pushdown automata

1.6 Non context free languages

- **Intro**: We know that set of regular languages has all the finite languages, Σ^*, Φ, and infinitely many infinite languages.

  We know that the set of context-free languages contains all the regular languages, some other languages that are not regular, and that all the context-free languages that are not regular languages are infinite languages.

  But are there any non-context free languages? if there are any, we know they must all be infinite languages

  But what would that mean?

  A language is context free if and only if there exists a context-free grammar that accepts it, and

  a language is generated by a context-free grammar if and only if it is accepted by a pushdown automaton.

  If someone comes to us with a description of a language and asks us to develop a CFG or PDA that accepts the language, we work on it for a long time but fail to come with anything, what does that mean?

  It may simply mean that the problem is too hard or that we are not clever enough ... it does not mean that no CFG or PDA exists.

  To claim at a language is non-context free one must prove that no CFG or PDA exists (very different than trying for a time and failing).

- **Derivations From CNF CFGs**: Before we continue our conversation about the prospect of noncontext free languages we first look at derivations from context free grammars that are in Chomsky Normal Form (CNF).

  Consider the derivations that come from a CFG in CNF

  Every derivation starts with the grammar's start symbol (a variable) S and applies one production at each stage generating a working string until the final stage when the word $w \in L$ defined by the CFG is produced.

  Every application of a live production has the net effect of adding a variable to the working string.

  Every application of a dead production has the net effect of replacing a variable (i.e., removing a variable) with a terminal in the working string

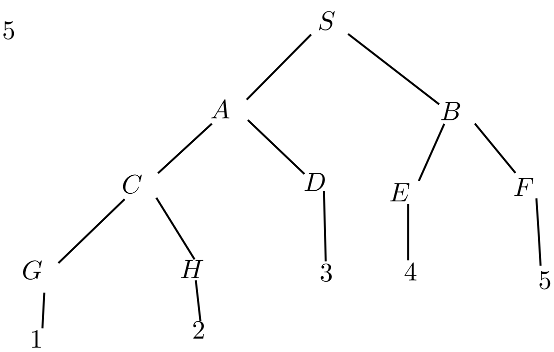
  Any derivation of a non-empty word $w \in L$ defined by the CFG in CNF whose length $|w| = n > 0$ must therefore always have exactly $2n - 1$ steps, or applications of productions, as follows:

  - $n - 1$ applications of *live productions* to convert the single variable (i.e., start symbol) S to the n variables needed for the subsequent *dead productions* and
  - n applications of *dead productions*; one to convert each variable to a single terminal.

We now turn our attention to derivation trees

Consider a finite language with exactly one word $L = \{12345\}$, the following two CNF CFGs each accepting L, and the derivation trees using each grammar.

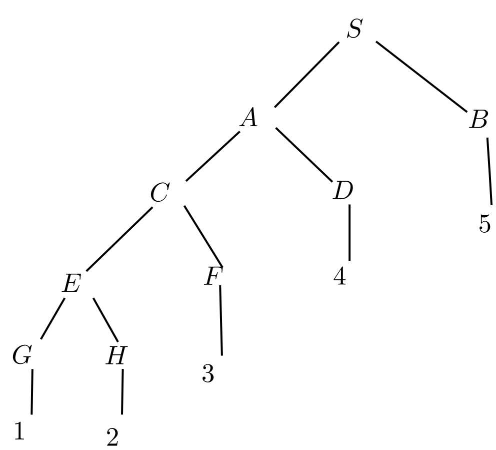
Depth 5



Using

$$
\begin{aligned}
S &\to AB \\
A &\to CD \\
B &\to EF \\
C &\to GH \\
D &\to 3 \\
E &\to 4 \\
F &\to 5 \\
G &\to 1 \\
H &\to 2.
\end{aligned}
$$

And



Depth 6

Using

$$
\begin{aligned}
S &\to AB \\
A &\to CD \\
B &\to 5 \\
C &\to EF \\
D &\to 4 \\
E &\to GH \\
F &\to 3 \\
G &\to 1 \\
H &\to 2.
\end{aligned}
$$

We first note that both derivation trees have

- The expected number of live productions ($n - 1 = 5 - 1 = 4$) and
- The expected number of dead productions ($n = 5$)

But we also note

- The shapes of the trees are different
- The depths of the trees are different

But what is the shallowest depth a derivation tree can be to produce a word with length n from a CFG in CNF?

In a derivation tree from a CFG in CNF,

- **Live productions**: The *branching factor*$= 2$, children are internal nodes
- **Dead productions**: The *branching factor*$= 1$, children are leaf nodes

The shallowest tree is one that first (i.e., from the root down) takes the fullest advantage of the live production's higher branching factor by pushing all the live productions to the top of the tree and pushing all the dead productions to the bottom.

So what is the shallowest depth a derivation tree can be to produce a word with length k from a CFG in CNF? The answer is

$$
\begin{aligned}
\lceil \log_2 (k) + 2 \rceil &= d \\
\implies 2^{d-2} &= k.
\end{aligned}
$$

Where d is the shallowest depth, with at least one path from root to leaf with $(\lceil \log_2 (k) \rceil + 2) - 1 = \lceil \log_2 (k) \rceil + 1$ variables.

- **The Pumping Lemma for Context Free Languages**: Let L be any context-free language. Then there is a constant n that depends on L such that if $z \in L$ and $|z| \geqslant n$, then we may re-write $z = uvwxy$ such that
  1. $|vwx| \leqslant n$,
  2. $|vx| > 0$, and
  3. for all $i \geqslant 0$, $uv^i wx^i y \in L$.
- **The pumping lemma for context free languages proof**: Let G be any context-free grammar that accepts a context-free language L.

Let G' be the CNF CFG that accepts $L - \{\Lambda\}$ having m variables.

If L is an infinite language, we can always find a word $z \in L$ that is longer than some arbitrary length, say the Pumping Lemma's constant n.

Set $n = 2^m$ and then select a word $z \in L$ such that $|z| \geqslant n$ (i.e., $|z| \geqslant 2^m$).

Consider the derivation tree of z using the CNF CFG G'.

Because $|z| \geqslant 2^m$, we know that there is at least one path from root to leaf with at least $\lfloor \log_2 2^m \rfloor + 1$, or at least $m + 1$, variables.

Because the CNF CFG G' has only m variables, we know that same path must repeat at least one variable at least once.

We traverse that path from the leaf node up to the root noting the lowest and second lowest occurrence of the first repeated variable.

Derivation trees for words $w \in L$ from a CNF CFG have a minimum depth that is based on $|w|$, and from that, we also know how many variables there must be in at least one path from the root to a leaf (i.e., terminal).

If you pick a $w \in L$ with $|w| \geqslant 2^m$, where m is the number of variables in the CNF CFG, then you are guaranteed that the derivation tree will have a path from root to terminal in which a variable is repeated … *the repeated variable is our "cycle"*, and

We can use that cycle to generate an infinite number of words that all must also be in the language L.

- **The Real Value of the Pumping Lemma for Context Free Languages**: The real value of the Pumping Lemma for CFLs is to prove that some infinite language is not context free ... here is the basic strategy – it is a proof by contradiction;

  You start with some infinite language that you suspect may be non context free.

  You assume that the language is a context free language.

  This is a crucial step in a proof by contradiction. You assume something, run the proof from that assumption, come to a contradiction (something that cannot possibly be true), and then that proves that your assumption must be false.

  You show that it is not possible to decompose a string into uvwxy such that … and so on (i.e., what The Pumping Lemma for CFLs assures us is always possible to do with infinite CFLs).

  This gives us our contradiction; we assume the language is a CFL, which means the Pumping Lemma for CFLs is in effect, which means we can always decompose a string into uvwxy such that … etc., but we find we cannot decompose the string into uvwxy , so contradiction.

  We conclude that our assumption that the original language is a CFL must be false, and therefore, it must be a non context free language.

- **Pumping lemma example**: Suppose $L = a^k b^k c^k \ \forall \ n \geqslant 0$

  Proof. Suppose L is context free, then $\exists \ n \in \mathbb{R} \ \ | \ \ \forall \ z \in L, \ |z| \geqslant n \ z = uvwxy$ for $u, v, w, x, y \in \Sigma^*$, with $|vx| > 0, \ |vwy| \leqslant n$ and $z' = uv^i wx^i y \in L \ \forall \ i \geqslant 0$.

  Let $k = n + 1$, then $z = a^{n+1} b^{n+1} c^{n+1}$. Since the middle portion of the decomposed string $z = uvwxy$, vwx must be weakly less than n in length, we have a couple cases for v, x. The first case is when v and x are all a's, all b's, or all c,s. In this case, pumping copies of v and x leads to an inbalance in one of the symbols. The second case is that v and x are homogenous in different symbols, for example v is all a's, and x is all b's. In this case, pumping would lead to an inbalance in two of the symbols. The final case is that v or x span some boundary. For example, $v = a^r b^\ell$. In this case, pumping would lead to more than one boundary, which leads strings not in L. Thus, there are no choices for v, x such that $z' = uv^i wx^i y \in L \ \forall \ i \geqslant 0$. Since we have a contradiction, L must not be context free.

- **PL example two**: Show that the language $L = a^k b^{k+1} a^k \ \forall \ n > 0$ is not context free.

  Proof: Assume L is context free, then $\exists \ n \in \mathbb{Z}^+ \ \ | \ \ \forall \ z \in L, \ |z| \geqslant n \ z = uvwxy$ for $u, v, w, x, y \in \Sigma^*$, with $|vx| > 0, \ |vwy| \leqslant n$ and $z' = uv^i wx^i y \in L \ \forall \ i \geqslant 0$.

  Let $z = a^{n+1} b^{2(n+1)} a^{n+1}$, for $k = n+1$. If v, x are homogenous substrings, for example $v = a^r, \ x = b^\ell$ for $0 \leqslant r < n + 1, \ 0 < \ell < n + 1 - r$, for all combinations of pairs of a, b, a. In this case, pumping would lead to one of the symbol sequences unchanged. For example, suppose $v = a^r, \ x = b^\ell$. Then, $v = a^{n+1-r} a^r b^\ell b^{2(n+1)} a^{n+1}$. Then $z' = a^{n+1-r+ir} b^{2(n+1)-\ell+i\ell} a^{n+1} \in L \ \forall \ i \geqslant 0$. Let $i = 2$, then $z' = a^{n+1+r} b^{2(n+1)+\ell} a^{n+1}$, we see that the a and b at the beginning grow, while the a at the end remains unchanged.

  The other case is that either v or x is non-homogenous, and span across some boundary. For example, $v = a^r b^\ell$, and x is some number of b's. In this case, pumping would lead to additional ab boundarys, which leads to strings not in the language. Since there are no choices for v, x that satisifies the property of context free languages, we have a contradiction and the language must not be context free. ■

- **PL example three**: Consider $L = \{ww : \ w \in (0 + 1)^*\}$. In other words, the left half of the string must be the same as the right half. For example, $z = 00110011 \in L$.

  Proof: Assume L is context free, then $\exists \ n \in \mathbb{Z}^+ \ \ | \ \ \forall \ z \in L, \ |z| \geqslant n \ z = uvwxy$ for $u, v, w, x, y \in \Sigma^*$, with $|vx| > 0, \ |vwy| \leqslant n$ and $z' = uv^i wx^i y \in L \ \forall \ i \geqslant 0$.

  Consider $z = 0^n 1^n 0^n 1^n$. Since $|vwx| \leqslant n$, the only hope of being able to satisfy the criterion is when v, x span the middle of the string. If v, x were contained entirely in the left or right half, then its clear that pumping would lead to a string not in L. Suppose then that v, x are contained within the middle portion such that v is 1^r, and $x = 0^\ell$, and their lengths satisfy $|uwx| \leqslant n$. Then, $z = 0^n 1^{n-r} 1^r 0^\ell 0^{n-\ell} 1^n$, and $z' = 0^n 1^{n-r+ir} 0^{n-\ell+i\ell} 1^n$, let $i = 0$, then $z' = 0^n 1^{n-r} 0^{n-\ell} 1^n \notin L$. We note that for any $i \geqslant 0$, $z' \notin L$. To find a v, x that satisfies the pumping lemma, we would need to be able to select a symbol in the first half as v, and then reach the same symbol in the second half for x, which is not possible given the condition $|vwx| \leqslant n$, since the lengths of each symbol are n characters long. The best we could do is reach the opposite (first) symbol in the second half for x, with v being the second symbol in the first half.

Thus, we have a contradiction and the language must not be context free.

- **PL example four**: Suppose $L = \{a^p : \ p \in \mathbb{P}\}$. Choose $z = a^\ell$ for $\ell \geqslant n$. Then $v = a^r$, $x = a^s$ for $0 < r+s \leqslant n$, which implies $z = a^r a^s a^{\ell-r-s}$, and $z' = a^{ir}a^{is}a^{\ell-r-s}$. Let $i = \ell+1$, then $z' = a^{\ell-r-s+r(\ell+1)+s(\ell+1)} = a^{\ell+r\ell+s\ell} = a^{\ell(1+r+s)}$. Since $\ell \in \mathbb{P}$, and $r+s > 0$, we know $p(1+r+s) \notin \mathbb{P}$. Thus, we have a contradiction and L must not be context free.

1.7 Turing machines

- **Intro**: In 1930's Alan Turing presented an abstract machine called the Turing Machine which serves to this day as the model for all computation (i.e., algorithms) and computing (i.e., computers)

- **TM General Description**: Turing Machines (TM) are similar to deterministic finite automata (DFA) in that a TM has:

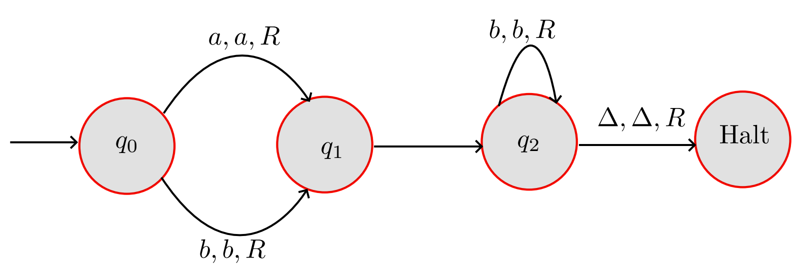
  - a tape that is infinite in one direction and a tape head
  - a finite number of states with exactly one that is designated as the start state, and
  - A set of transitions (i.e., directed edges) that deterministically take the TM from one state to the next based on what the tape head reads from the tap
  - Zero or more HALT state(s) that stops the TM and accepts the input string,
  - no requirement that every state must have an outgoing edge for every symbol (i.e., requirement of DFAs), with the same understanding that if the TM is in a state from which there is no edge for the symbol pointed at by the tape head, then the TM crashes and rejects the input string, and
  - no requirement that the entire input string must be read before accepting or rejecting the string.

  Turing Machines (TM) differ from FA and PDA in that a TM does the following ordered sequence on each transition:

  1. Read the symbol from the tape (same as FA and PDA).
  2. Write a symbol to the tape at the same location (may be the same symbol that was read).
  3. Move the tape head one slot either left or right with the understanding that an attempt to move left from the leftmost part of the tape crashes the TM (i.e., halts execution and rejects the input string).

  Accordingly, each transition in a TM is annotated with the three above

- **Turing machine example**:



In this example the TM

- always wrote the same symbol that it read
- always moved the tape head to the right

That is not always the the case with a TM.

- **Another TM example**: Consider the language $L = a^n b^n \; \forall \; n > 0$. We know that L is not a regular language and that it is a context free language, so there exists a pushdown automaton (PDA) that accepts L.

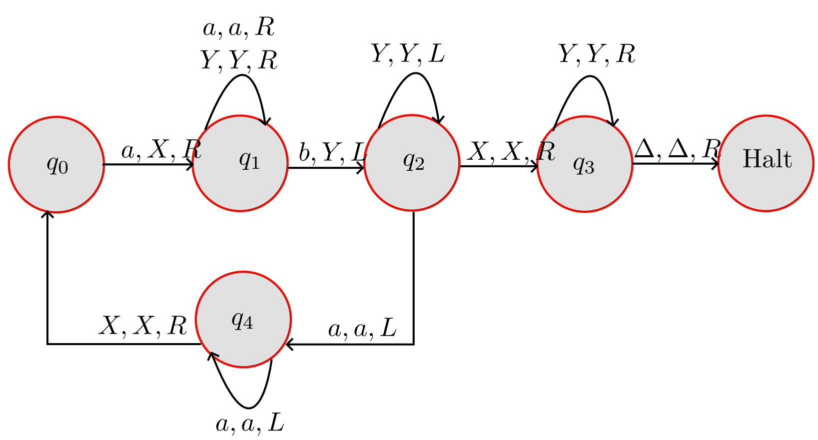
  A TM has no stack, but it has a tape head that can write and can move in either direction. What, then, is the strategy in designing a TM that accepts L?

  As you read each a

  - Cross the a off as an "already read" a (e.g., replace it with an X).
  - Move the tape head to the right until we encounter our first b.
  - Cross that b off as an "already read" b (e.g., replace it with a Y).
  - Move the tape head back to the left until we encounter the X we
  - had just written, then move the tape head one space to the right

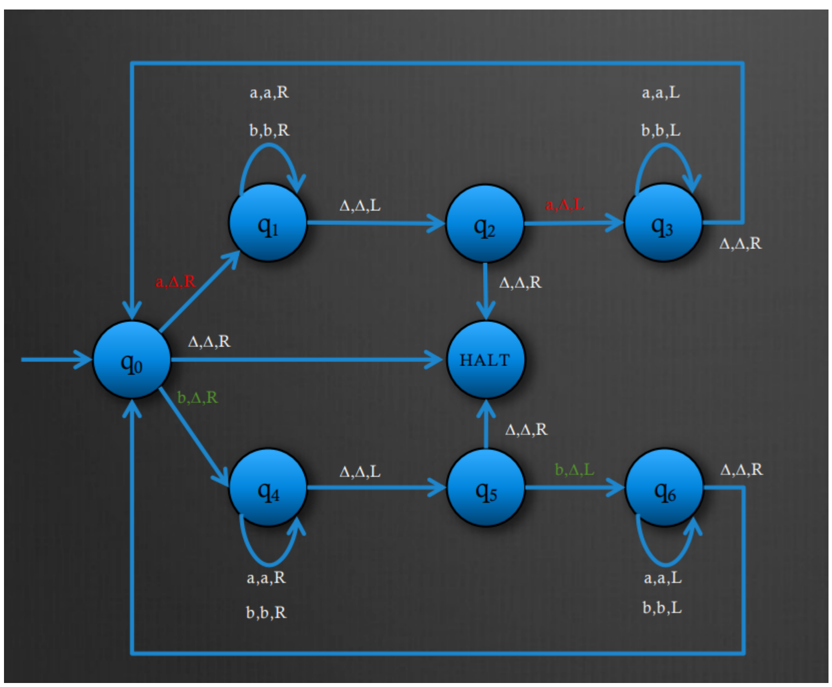
  At this point the tape head should point either to the next a or to the first Y that we wrote.

  - If the tape head points to the next a, then simply process the steps above again.
  - If the tape head points to the first Y, then move the tape head to the right making sure that it passes only over Y's until it reaches a blank Δ (to assure there were not more b's than a's).



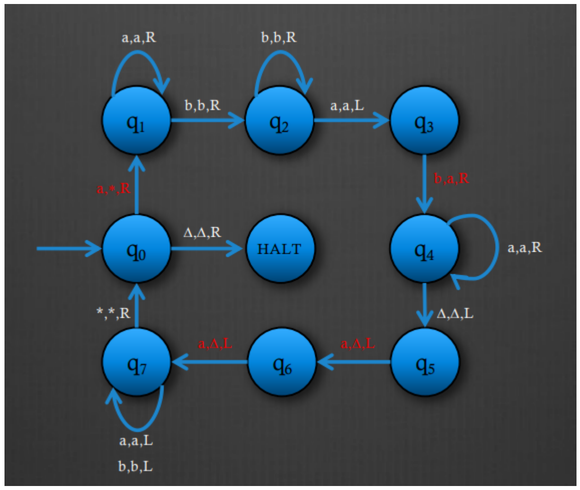
- **Palindrome**: We know that Palindrome is not a regular language and that it is a context free language, so there is a PDA that accepts it. The basic strategy in designing a PDA that accepts Palindrome relies on the PDA's non- determinism.
  - In the first portion of the PDA; read and push the first half of the string onto the stack.
  - Non-deterministically transition to the second portion of the PDA by either;
    - reading a single symbol off the tape without pushing it onto the stack (for odd-length palindromes) or
    - make a ϵ-move (for even-length palindromes).
  - In the second portion of the PDA; read the second half of the string and pop the stack for each symbol read to make sure that the two symbols (read and popped) match.
  - When you reach the end of the tape make sure that there are no more symbols in the stack.

  A TM has no stack and it is deterministic. What, then, is your strategy in designing a TM that accepts Palindrome ?
  - Process the string by reading and erasing characters from the ends taking care to make sure that the characters at each end of the string always match.
  - Take care to account for both even- and odd-length palindromes.



- **Turing machine example**: Consider the language $L = a^n b^n a^n \ \forall\ n \geqslant 0$. We know L is not context free
  - Start by marking the first a as read with a *
  - Move tape head to ba transition and backup to b
  - Replace that b with an a as read with a *

- Move tape head to end of input and backup to a
- Replace last two a's with blanks as read with a *
- Back tape head to * and advance to a
- Repeat



- **Formal Definition TM**: Formally, a Turing Machine (TM) is denoted

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \Delta, F).$$

  Where

  - Q is a finite set of states
  - Γ is a finite set of tape symbols
  - $\Delta \in \Gamma$ is the blank
  - $\Sigma \subset \Gamma$ such that $\delta \notin \Sigma$, is a set of input symbols
  - $q_0 \in Q$ is the start state
  - δ is the transition function that maps
    - a state in Q and a (read) tape symbol in Γ to
    - some (written) tape symbol in Γ, and a tape head move direction (L or R), and some state in Q
    - $\delta : Q \times \Gamma \to \Gamma \times \{L, R\} \times Q$

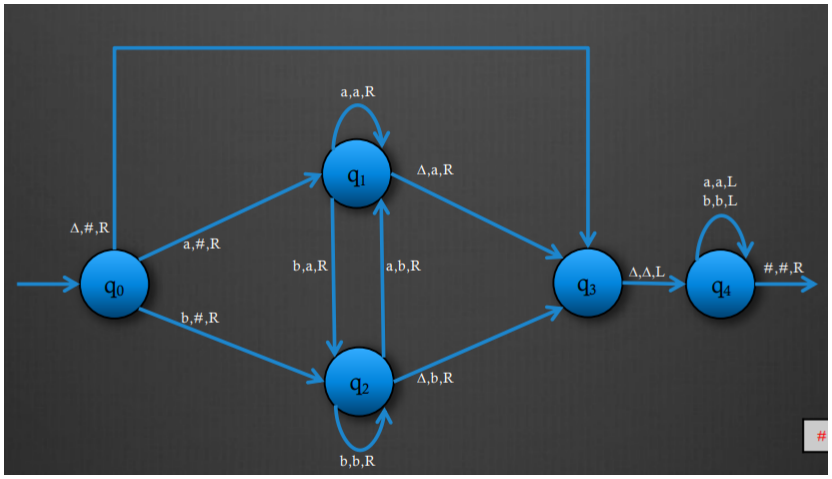
    Note: δ is a partial function in that it may be undefined for some values in $Q \times \Gamma$
  - $F \subseteq Q$ is the (possibly empty) set of HALT state(s)

- **Prepend #**: With a tape that is infinite in only one direction it is sometimes helpful to be able to shift the entire input string to the right one cell and then place in the leftmost cell of the tape a special symbol (e.g., #) with the understanding that the special symbol will never appear anywhere else on the tape

  In this way the rest of the TM can safely move the tape head left always taking care to not move left from #.

There is a TM that can be used at the beginning of any TM – it shifts the entire input string, prepends #, and positions the tape head at the first symbol of the original input string.

This TM is written for the input alphabet $\Sigma = \{a, b\}$ but it can be easily modified to work with any input alphabet.



- **Turing machines as transducers**: Thus far we have considered the use of Turing Machines as language acceptors.

  Because Turing Machines can write to the tape, we can also use them to perform calculations that transform value(s) that are initially written on the tape (i.e., input) to some other value (i.e., output) on the tape (e.g., the TM implements some function f(x)).

- We write the input variables onto the tape, and the TM must reach a HALT state (so the input string is in the language defined by the TM), but what is left on the tape is the output of the function.

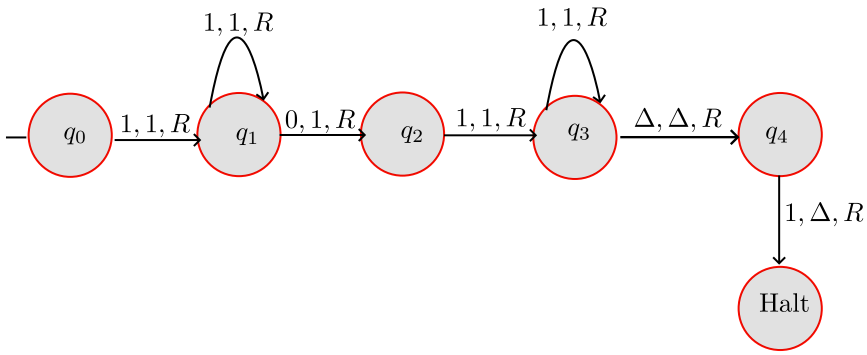
  When we use TMs in this way they act as a language acceptor (i.e., the set of valid inputs) but because the output is also valuable we say the TM is a transducer

- **TM transducer example**: Consider the language $L = 1^m01^n \; \forall \; m, n > 0$

  We can create a TM that accepts L, but can we also create a TM that leaves on the tape (i.e., as output) a string of the form 1^{m+n}?

  In other words, can we create a TM that computes

$$f(m, n) = m + n.$$

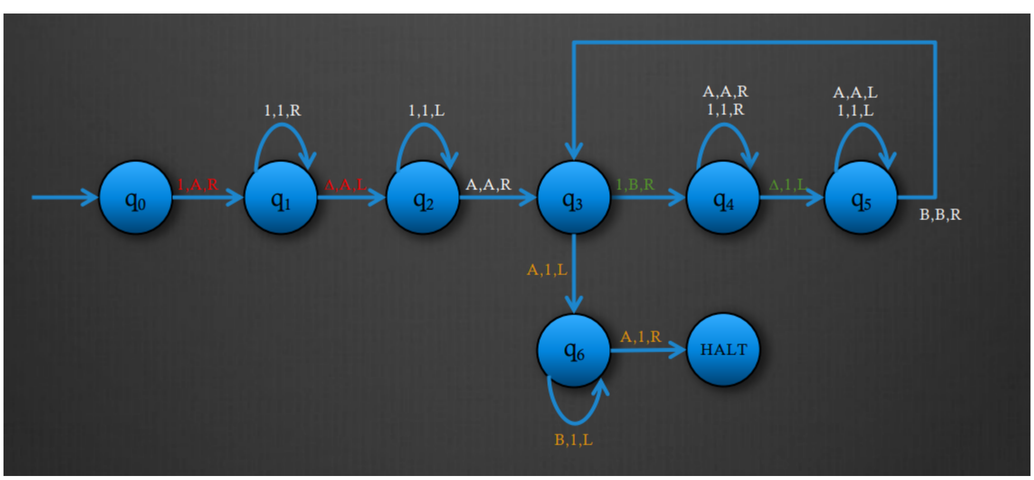


- **Another TM Transducer** : Consider the regular language $L = 1^n$ for $n > 0$

  Can we create a TM that accepts L and leaves on the tape a string of the form 1^{2n}

  In other words, can we create a TM that computes $f(n) = 2n$?

  – Start by marking the first 1 as read with a A
  – Move tape head to first D , replace with A
  – Back tape head to first A, then move R one
  – Now enter loop replacing each original 1 with B and appending a 1 for each, always backing up tape to last B and then moving R one
  – Eventually we will have processed the last original 1 which will position the tape head on the second A
  – All that remains is to move the tape left replacing all the A's and B's with 1's taking care to stop and move R when we reach the first A.



- **Final TM Transducer**: $f(m, n) = mn$

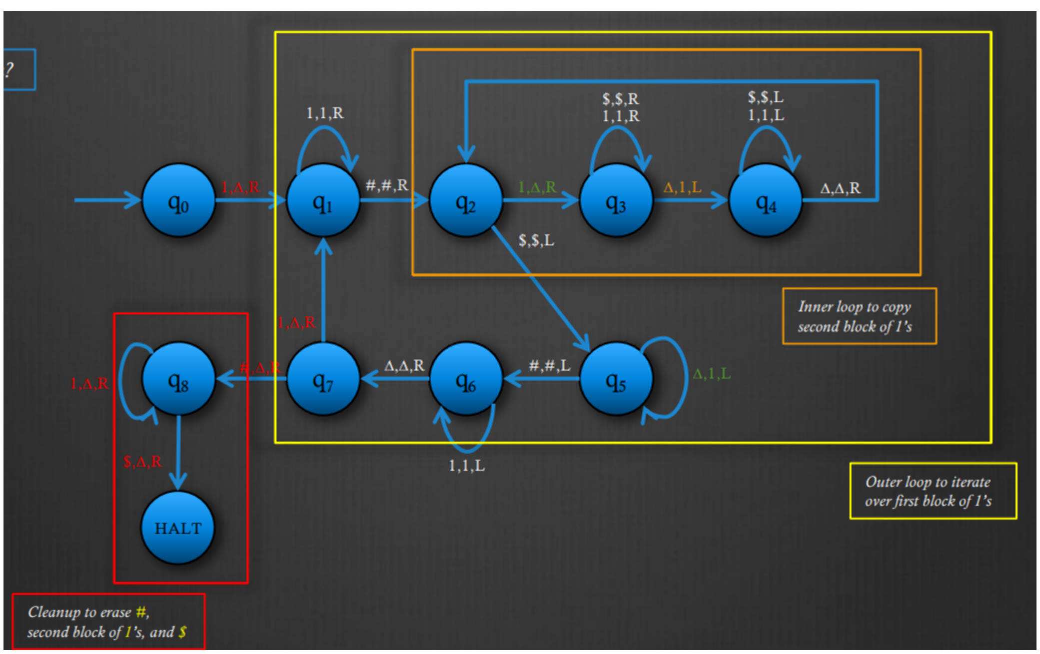
  In the previous two examples the output was written left-justified on the tape

  What if we temporarily relaxed that restriction by saying that the output may appear anywhere on the tape as long as it is contiguous (i.e., output may be preceded on tape by an arbitrary number of blank cells).

  Consider the regular language $L = 1^m 1^n\$$ for $m, n > 0$

  Can we create a TM that accepts L and leaves somewhere on the tape a string of the form 1^{mn} to compute $f(m, n) = mn$

  1. Start by marking the first 1 as "read" by erasing it and advancing the tape head to the first 1 after the #.
  2. Erase the first 1, append a Δ, return the tape head to the Δ you just wrote, and then move right.
  3. Repeat until you have erased and appended a copy of the entire second block of 1's; this will leave the tape head at the \$.
  4. Return the tape head to the first Δ we wrote and then move right, taking care to restore the second block of 1's.
  5. Repeat that process until you have appended a copy of the second block for the last 1 in the first block; this will leave tape at #.
  6. Finish by advancing the tape head to \$ and then one more cell right, erasing #, the second block of 1's, and \$.



1.7.1 Decidability and Languages Accepted by Turing Machines

- **Membership in Languages**: Given a language and an acceptor for that language (e.g., FA or PDA) one could ask the question... "Is a given string $w \in \Sigma^*$ in (or not in) the language?"

  Or to put the question in different ways

  1. For any $w \in \Sigma^*$ will the acceptor stop and determine whether or not w is in the language defined by the acceptor?
  2. Does the acceptor always stop and partition Σ^* into strings that are in the language and strings that are not in the language? For regular languages the answer is yes. We can see this from the definition of deterministic finite automata (DFA).

     In this way for any string $w \in \Sigma^*$ the DFA always stops and reports whether (or not) the w is in the language.

     In this way for any string $w \in \Sigma^*$ the DFA always stops and reports whether (or not) the w is in the language.

     For context free languages the answer is also yes. We can see this from the definition of context free grammars (CFG) that are in Chomsky Normal Form (CNF).

     Recall that any derivation of a non-empty word $w \in L$ defined by a CFG in CNF whose length $|w| = n > 0$ must always have exactly $2n - 1$ steps ($n - 1$ live productions plus n dead productions). Since any CFG has only finitely many productions, then for any string $w \in \Sigma^*$ with length $|w| = n > 0$, there are only finitely many derivations having exactly $2n - 1$ steps ... simply check them all to determine if any one derivation produces w.

     In this way the CNF CFG always stops and reports whether (or not) the w is in the language.

     In other words, the CNF CFG partitions Σ^* into strings that are in the language and strings that are not in the language.

- **Decidability**: So, for regular and context-free languages we can take any string $w \in \Sigma^*$ and determine whether or not w is in the language.

  In formal terms, we say that the question of membership for regular and context free languages is *decidable.*

  A question whose answer is boolean (i.e., yes or no) is decidable if there exists an effective method (or effective procedure or algorithm) that can determine the answer.

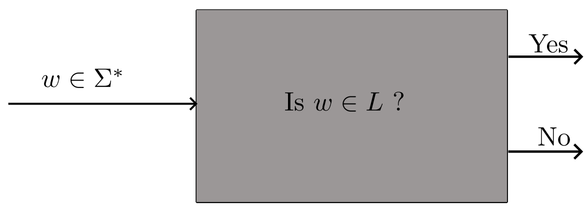
  An effective method is one that always halts and produces the correct answer

  A question whose answer is boolean and for which there is no algorithm (i.e., stops on all input and correctly answers "yes" or "no") is *undecidable.*

We can, therefore, say that the question of membership for regular and context free languages is decidable because we have an effective procedure that answers the question for both classes of languages.

– **Regular languages:** Use the DFA which always halts and gives correct answer.
– **Context free languages:** Use the CNF CFG which has only finitely many derivations that could possible produce a given string, simply check them all to see if one does.

It is sometimes convenient to represent a decidable question by depicting its decision procedure (or algorithm) as a "black box".



Note that in this representation the "black box" is not the language acceptor automaton (i.e., it is not a DFA or PDA)

For the decidable question, "Is $w \in L$?"

– **For Regular languages:** The "black box" represents the algorithm that uses the DFA which always halts and gives correct answer.
– **For Context free languages:** The "black box" represents the algorithm that uses the CNF CFG by checking all of the finitely many derivations that have exactly $2n - 1$ productions.

As we can see by the definitions of decidable/undecidable, the question of whether or not algorithm exists (i.e., a decision procedure that always stops and correctly answers "yes" or "no") is not limited to questions about formal languages (e.g., is membership in regular languages or CFLs decidable).

The question of decidability (i.e., whether an algorithm exists) can be, and is, applied to many domains.

Questions of decidability are at the core of all computability (i.e., does an algorithm exist?).

- **Euclid's conjecture**: Sometimes we answer a "yes/no" question by providing a proof, which answers the question entirely, and therefore, we do not need an algorithm

For example, there is Euclid's Conjecture which states:

There are infinitely many prime numbers.

Proof (by contradiction). Assume there is a finite list of all the prime numbers $p_1, p_2, ...p_r$

Consider $m = p_1 \cdot p_1 \cdot ... \cdot p_r + 1$

We observe that m cannot be any of the numbers in our original list $p_1, p_2, ..., p_r$ but that list was supposed to be all the primes, so contradiction.

If m is not prime, then it must be evenly divisible by some prime, call it p

We observe that p cannot be our original list $p_1, p_2, ..., p_r$, but that list was supposed to be all the primes, so contradiction.

- **Is a Number Prime?**: Other times we answer a "yes/no" question by providing an algorithm.

  Given some integer $n > 1$, is n prime?

  Algorithm:

  1. Divide n by every integer i in the range $2 \leqslant i \leqslant \sqrt{n}$
  2. If any i in that range evenly divides n then n is not prime; otherwise n is prime

- **Goldbach's Conjecture**: Sometimes we encounter a question for which there is no proof and there is no algorithm (i.e., always halts with "yes/no" answer).

  One of the oldest and best-known unsolved problems in number theory was posed on June 7, 1742 by the German mathematician Christian Goldbach in his letter to Leonhard Euler. It states:

  Every even integer greater than two can be expressed as the sum of two primes.

  1690-1764 One approach (not an algorithm):

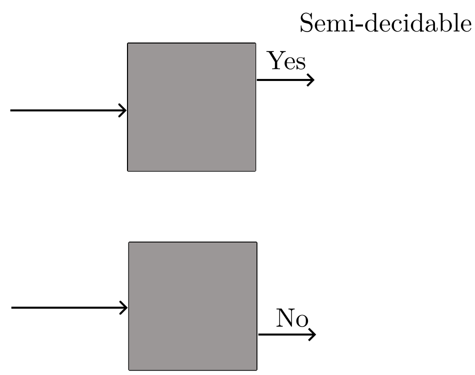
  1. Test every even number that is greater than two.
  2. For each number try to find two primes whose sum is equal to that number
  3. If we find two such primes, then proceed to the next even number; otherwise halt and declare Goldbach's Conjecture false.

  If Goldbach's Conjecture is false, then the proposed approach will eventually HALT and provide the answer "no"; otherwise it will run forever

  We say that Goldbach's Conjecture is undecidable.

- **Semi-decidable**: While it is true that Goldbach's Conjecture is undecidable (i.e., because there is no algorithm that always halts with "yes/no" answer), the fact that we have a method that always halts and gives one of the "yes/no" answers gives rise to a new definition.

A question whose answer is boolean (i.e., yes or no) is *semi-decidable* if there exists a method (or algorithm) that always halts when the answer is yes or no, but may run forever for the other answer



- **Summary**: Combining our new definition semi-decidable with our previous definition of decidable/undecidable we have the following:
  - **Decidable:** There is an algorithm that always halts with correct "yes/no" answer.
  - **Undecidable:** Any yes/no question that is not decidable.
    - **Semi-decidable:** an undecidable question for which there exists an algorithm that always halts with one of the answers "yes" or "no".
- **What About TMs?**: We know that Turing Machines (TMs) can be used as language acceptors.

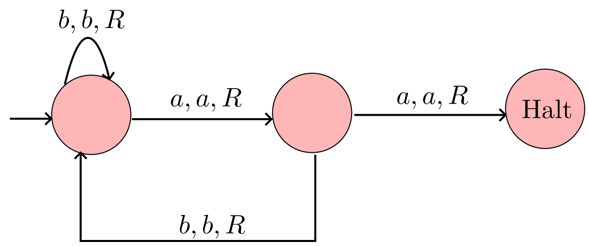
  In fact, TMs can accept all regular and context free languages plus other languages that are not context free

  But do TM's always partition Σ^* into strings that are in the language and strings that are not in the language?

  In other words, is the question of membership in languages accepted by Turing Machines decidable?

  The answer is not always

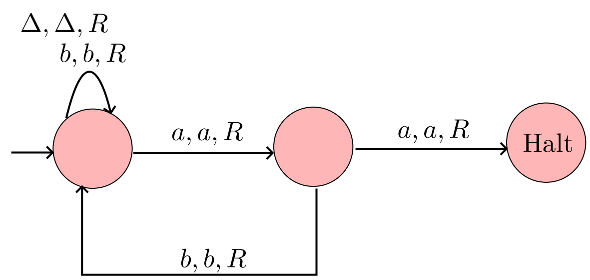
Consider the following TM that accepts all strings with a double a, (i.e., aa):



For all $w \in \Sigma^*$ this TM:

- For all $w \in L$, HALT and accept
- For all $w \notin L$, crash and reject

Consider this slightly modified TM that also accepts all strings with a double a, (i.e., aa):



For all $w \in \Sigma^*$ this TM:

- For all $w \in L$, HALT and accept
- For all $w \notin L$,
  * sometimes crash and reject (e.g., no double a but ends in a)
  * sometimes loop forever (e.g., no double a, ends in b)

You may ask... ”If there is an algorithm that always stops, then why not simply use that?”.

Because there are some problems for which the only algorithm(s) cannot eliminate the possibility of looping

Every Turing Machine (TM) partitions Σ^* into three classes;

1. HALT and accept
2. crash (i.e., stop) and reject
3. loop forever

For any specific TM one or more of those three classes may be empty, meaning, for any specific TM

- Might always either HALT or crash; might always crash; might always accept.
- [general case]: Sometimes HALT and accept, sometimes crash and reject, or sometimes loop forever

- **A Closer Look at a General TM** : In general, a TM will process strings by;
  - $w \in L$, always halt and accept any string that is in the language.
  - $w \notin L$, sometimes halt by crashing, other times loop forever.

In the black box depiciction, we might make the no line dashed

Recall, a decidable problem is one for which there exists an algorithm that always halts with the correct "yes/no" answer.

If there is a possibility that the algorithm may sometimes not halt, then that possibility alone renders the problem undecidable

This gives us the definition of two new languages accepted by Turing Machines

- **Languages defined by turing machines**: A language that is accepted by a Turing Machine is said to be a *recursively enumerable language.*

It is convenient to single out a subset of the recursively enumerable languages as follows:

A language that is accepted by at least one Turing Machine that halts on all inputs is said to be a recursive language.

Membership in recursive languages is decidable because, by the definition of recursive languages, there is a TM that always halts.

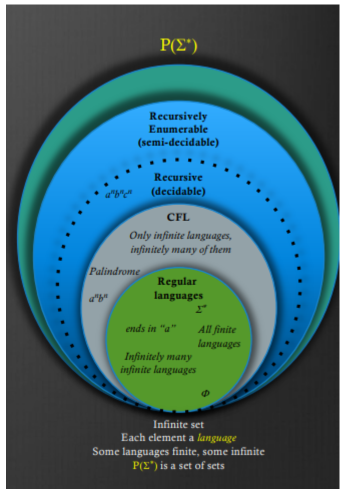
Membership in recursively enumerable languages is not decidable; it is semi-decidable. For recursively enumerable languages (that are not recursive) there are only Turing Machines that

- For any $w \in L$ always HALT and accept
- But for a $w \notin L$, sometimes crash and reject, other times loop forever

The existence of recursively enumerable languages raises the unsettling point ...

If a TM is running for a long time, how do we know if the machine will eventually stop (HALT+accept or crash+reject) or loop forever?

We don't know... Worst yet, we can't know



1.7.2 TM Variations

- **Intro**: As we have seen, different classes of automata have different power in their ability to define languages

  - **Finite automata**: Regular languages
  - **Pushdown automata**: Context free languages
  - **Turing Machines**: Recursively enumerable languages

  We have also experimented (and we can do other experiments) with different automata to see how such changes might impact the languages each accepts

  For Finite automata

  - Adding ϵ-moves and/or non-determinism didn't make a difference
  - Adding a stack makes a difference; equivalent to PDA

  For Pushdown automata

  - Removing non-determinism (i.e., forcing determinism) makes a difference; reduces the languages accepted
  - Adding one or more stacks (i.e., >1 stack) makes a difference; equivalent to Turing Machine

- **Can we do better?** In the cases that we have studied for which modifications make no difference, we demonstrate equivalence by presenting algorithmic conversions between the automaton with and without the modifications

  In such cases we never concern ourselves with efficiency; How much time or how many steps would one automaton take compared to another?

  We only concerned ourselves with whether or not both automata always produce the same result; Does the modification render the automaton more restrictive, less restrictive, or keep it exactly the same.

  We now consider variations on the Turing Machine with the same question in mind; Will the variation change the set of languages the TM can accept?

  We ignore questions of efficiency

- **Stay Option**: What if we allowed the TM to not move the tape head on a transition, instead of just L or R also allow S, which means "stay"?

  On the surface this may sound trivial, but it adds the ability to change state without disturbing the tape or tape head

  While it is obvious that adding the stay option does not reduce the languages a Turing Machine can accept one might ask ... Does the stay option give Turing Machines any extra real power? The answer is no it does not

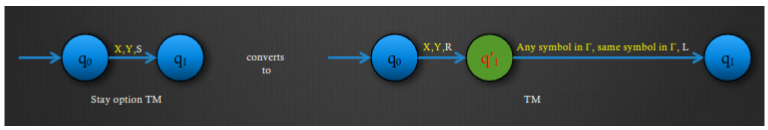
  Although the stay option can be useful in reducing the number of states a TM may require, it adds no new power to TMs.

Theorem: For any Turing Machine with the stay option there is some Turing Machine that acts the same way on all inputs; looping, crashing, or accepting, while leaving the same data on the tape, and vice versa

First, $TM \subseteq TM$ with stay option. Every TM is a TM with a stay option that simply does not have any stay option transitions

Part II... TM with stay option $\rightarrow$ equivalent TM

Construct a new TM from the TM with the stay option by first copying the machine and then converting every transition with a stay option as follows



$\therefore$ Turing machine with stay option = Turing Machine

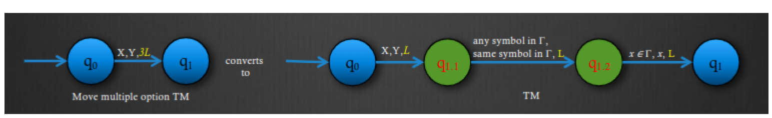
- **Move Multiple**: What if we allowed the TM to move the tape head multiple cells on a transition, instead of just L or R (one cell)?

  Theorem: For any Turing Machine with the move multiple option there is some Turing Machine that acts the same way on all inputs; looping, crashing, or accepting, while leaving the same data on the tape, and vice versa.

  Proof. Part I: TM $\subseteq$ TM with move multiple option. Every TM is a TM with a move multiple option where $n = 1$ on all move transitions

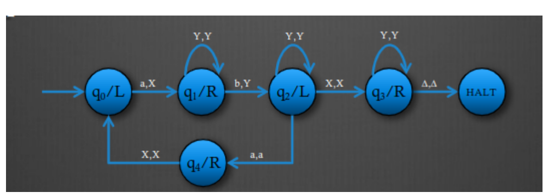
  Part II. TM with move multiple option $\rightarrow$ equivalent TM

  Construct a new TM from the TM with the move multiple option by first copying the machine and then converting every transition where $n > 1$ as in the the following $3L$ example (do likewise for nR):



  $\therefore$ Turing machine with move multiple option = Turing Machine

- **Move-in-State**: Consider the following variation to Turing Machines that specifies tape head movement in states rather than on transitions:



Transitions labeled p, q meaning read p and replace with q on tape

States (except HALT) labeled $q_i/\{L \text{ or } R\}$ meaning move tape head either L or R upon entering state.

Do not move tape head at beginning in q_0 but if you re-enter q_0 then move the tape head as indicated.

In original TMs when we entered a state like q_j above we moved the tape head, sometimes L other times R, before entering the state. How do we handle this with a move-in-state TM?

Does this make the TM or the move-in-state TM more powerful than the other? ... No, they are equally powerful.

Theorem. For any Turing Machine with the move-in-state option there is some Turing Machine that acts the same way on all inputs; looping, crashing, or accepting, while leaving the same data on the tape, and vice versa.

Proof. Part I: TM with move-in-state $\rightarrow$ TM

1.7.3 Encoding TMs

- **Recall**: Recall that a Turing Machine is formally defined as a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \Delta, F).$$

  And that we conveniently depict it as a state diagram.

- **Encoding**: Rather than using the 7-tuple to draw a picture we could, instead, represent the TM as a string of characters (i.e., encode the TM)

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \Delta, F) \rightarrow \text{ some string } w \text{ that represents } M.$$

  There are many ways to encode a Turing Machine, we present one.

  Recall that Q is a finite set of states where

  – $q_0 \in Q$ is the start state
  – $F \subseteq Q$ is the (possibly empty) set of HALT state(s)

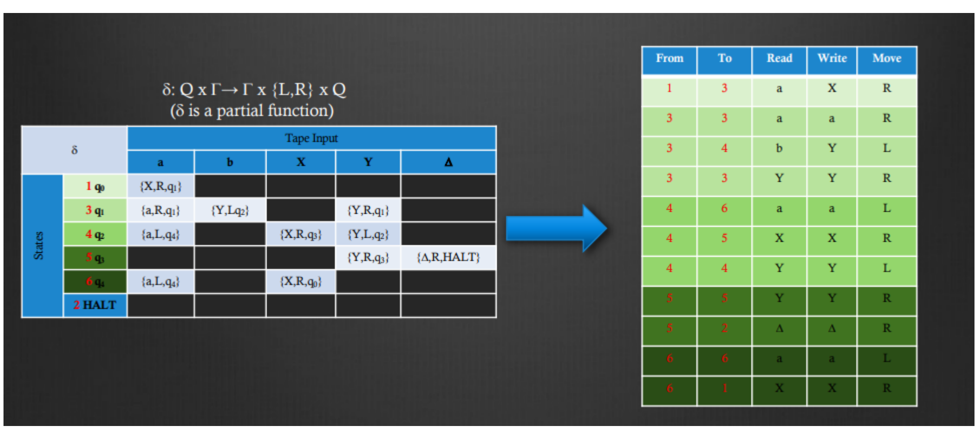
  Without loss of generality we can modify M by "collapsing" any and all HALT states to a single HALT state.

  In other words, if F was originally non-empty, then F will be left with a single state (and δ modified accordingly), otherwise F will remain empty and δ will be left unchanged.

  We continue by assigning numbers to each of the states in Q as follows:

  – The start state q_0 is assigned 1.
  – The (now single) HALT state, if one exists, is assigned 2.
  – The remaining states are assigned arbitrary numbers other than 1 or 2 and such that each state is assigned a unique number.

  Use the numbered states and the transition function δ to create a new table that has a row for each transition (i.e., cell in δ).



δ: Q x Γ→ Γ x {L,R} x Q
(δ is a partial function)

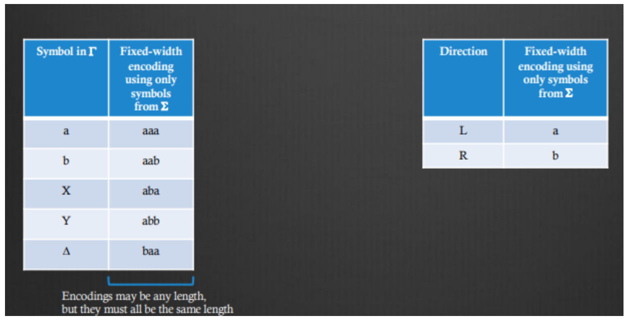
δ		Tape Input				
		a	b	X	Y	Δ
States	1 q0	{X,R,q1}				
	3 q1	{a,R,q1}	{Y,L,q2}		{Y,R,q1}	
	4 q2	{a,L,q4}		{X,R,q3}	{Y,L,q2}	
	5 q3				{Y,R,q3}	{Δ,R,HALT}
	6 q4	{a,L,q4}		{X,R,q0}		
	2 HALT					

From	To	Read	Write	Move
1	3	a	X	R
3	3	a	a	R
3	4	b	Y	L
3	3	Y	Y	R
4	6	a	a	L
4	5	X	X	R
4	4	Y	Y	L
5	5	Y	Y	R
5	2	Δ	Δ	R
6	6	a	a	L
6	1	X	X	R

We next, use input symbols in $\Sigma \subset \gamma$ to create:

- A fixed-length encoding for each tape symbol in Γ and
- An encoding for each direction (L or R)

Note, in order to do this Σ must have at least two symbols



Symbol in Γ	Fixed-width encoding using only symbols from Σ
a	aaa
b	aab
X	aba
Y	abb
Δ	baa

Direction	Fixed-width encoding using only symbols from Σ
L	a
R	b

We now encode each row of the new table as follows;

- **Transition from state**: $i \to j:\ a^i b a^j b$
- **Read/Write:** use fixed-width encoding
- **Direction**: use fixed-width encoding

From	To	Read	Write	Move	Encoding
1	3	a	X	R	abaaabaaaabab
3	3	a	a	R	aaabaaabaaaaaab
3	4	b	Y	L	aaabaaaabaababba
3	3	Y	Y	R	aaabaaababbabbb
4	6	a	a	L	aaaabaaaaaabaaaaaaa
4	5	X	X	R	aaaabaaaaababaabab
4	4	Y	Y	L	aaaabaaaababbabba
5	5	Y	Y	R	aaaaabaaaaababbabbb
5	2	Δ	Δ	R	aaaaabaabbaabaab
6	6	a	a	L	aaaaaabaaaaaabaaaaaaa
6	1	X	X	R	aaaaaabababaabab

The encoding of the TM concludes by concatenating the encoded rows (in any order) to create a single string. (with the understanding that start state = 1 and halt state = 2.)

- **Notes and the code word language (CWL):** Every TM with at least two symbols in Σ can be encoded to a string

  Not every string

  - Can be decoded to a TM (e.g., strings that begin with 'b' do not represent a TM)
  - **Decodes to a valid TM**: Might decode to a TM that is non-deterministic, has transitions from the HALT state, have READ, WRITE, or Move encodings that are invalid, etc.

In fact, using our last example where each symbol in Γ had an encoding length=3, from Σ^* we see that anything outside $(a^+ba^+b(a+b)^7)^*$ is definitely not a valid TM, and even some of those strings are not a valid TM.

Nonetheless, that regular expression comes pretty close to defining the set of valid TM's so we call it the Code Word Language (CWL)

$$\text{CWL } = (a^+ba^+b(a+b)^7)^*.$$

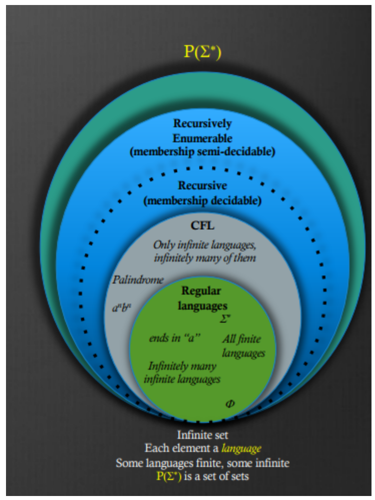
And we note

$$\text{valid TM encodings } \subset \text{ CWL } \subset \Sigma^*.$$

1.7.4 Revisiting Recursively Enumerable Languages

- **Recall**: The definitions of recursively enumerable and recursive languages: A language that is accepted by a Turing Machine is said to be a recursively enumerable language

  A language that is accepted by at least one Turing Machine that halts on all inputs is said to be a recursive language.



  We've seen an example of a recursive language that is not a context free language: $a^nb^nc^n$

  But is there a language that is recursively enumerable that is not recursive? Is there a language that is not even recursively enumerable?

  Yes to both

- **Turing Machines and Their Encodings**: Recall that we can take a Turing Machine (having at least two input symbols in Σ) and encode it using its own input alphabet Σ

  That means we can use a TM to process its own encoding.

  Thinking about this same prospect in a different way, consider the strings in CWL $= (a^+ba^+b(a+b)^7)^*$, which we can partition into the following three sets:

  - Invalid TMs
  - Valid TMs that accept their own encoding
  - Valid TMs that do not accept their own encoding

And given that partitioning we can define two new languages, Acc,NotAcc $\subseteq$ CWL, as follows:

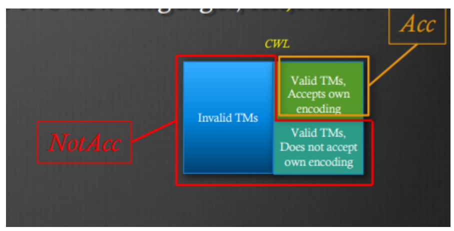
$$Acc = \begin{cases} w \in \text{ CWL such that:} \\ w \text{ is a valid TM that is} \\ \text{accepted by the TM it encodes} \end{cases}$$

$$NotAcc = \begin{cases} w \in \text{ CWL such that:} \\ \text{1. w is not a valid TM or} \\ \text{2. w is a valid TM that is not accepted by the TM it encodes} \end{cases}.$$

What kind of language is NotAcc? Regular? Context-free? Recursive? Recursively enumerable? none of the above, meaning, there does not exist a TM that accepts NotAcc.

Theorem. There exists a language that is not recursively enumerable.

Proof (by contradiction). Assume that the language NotAcc is recursively enumerable.:



This means that there exists some Turing Machine, T, that accepts NotAcc.

If we have a Turing Machine, T, then we can create a string w from T's input alphabet Σ that encodes T. Because w was built from T's input alphabet Σ, we can process w using T (i.e., have a TM process its own encoding).

There are only two possibilities; either T accepts w, or it does not.

Case I – T accepts w

Because w is a valid encoding of T, and (in this case) T accepts w, we see that $w \in$ "Valid TM's, Accepts own Encoding", which means $w \notin$ NotAcc. However, (in this case) T accepts w, which means $w \in$ NotAcc. (contradiction)

Case II – T does not accept w

Because w is a valid encoding of T and (in this case) T does not accept w, we see that $w \in$ "Valid TM's, Does not accept own Encoding", which means $w \in$ NotAcc.

However, T (in this case) T did not accept w, which means $w \notin$ NotAcc (contradiction)

Since the only two possible cases each led to a contradiction, we conclude that our assumption that NotAcc is a recursively enumerable language must be false.

$\therefore$ NotAcc is not a recursively enumerable language.

1.7.5 Universal turing machine

- **Turing Machines and Their Encodings**: We have seen that NotAcc is not a recursively enumerable language, meaning one cannot create a TM that accepts NotAcc.

  What about the remaining partition?

  What kind of language is Acc? Regular? Context-free? Recursive? Recursively enumerable? ... before we can answer that we need to create a special TM.

- **UTM**: We will create a special TM and call it a UTM. We will describe our new UTM in general terms rather than drawing it explicitly.

  First, the UTM takes as input some w which is an encoding of some TM.

  The basic function of the UTM is to simulate the TM that is encoded as its input, that is, the UTM should;

  - accept strings the encoded TM would accept
  - Reject (i.e., crash) strings the encoded TM would reject
  - Loop on strings on which the encoded TM would loop

  The first thing the UTM does is

  1. Shift the input string to the right one cell inserting a # in the leftmost cell
  2. Append a $ to the end of the input string
  3. Append a copy of the original input string just after the $

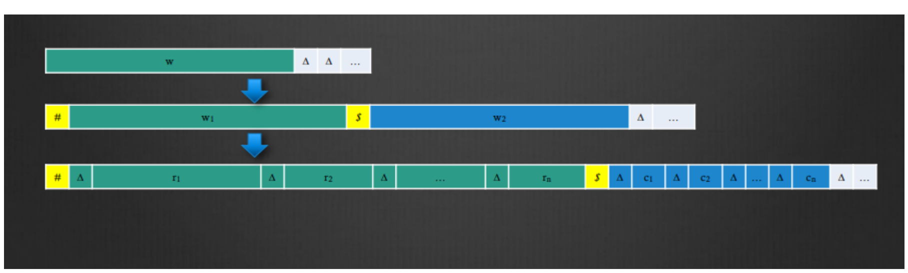
  The general idea is that the UTM will use

  1. w_1 to simulate the encoded TM (never changing it)
  2. w_2 as the input to the encoded TM (changing often)

  Recall that an encoded TM is a concatenation of rows from a table (i.e., one row per transition in the encoded TM).

  The UTM next inserts a blank (Δ)

  1. Before each row in w_1
  2. Before each symbol in w_2



To complete its initialization phase, the UTM Inserts a “dummy row” at the beginning of the tape whose only value is its “To” state which must indicate the start state = 1

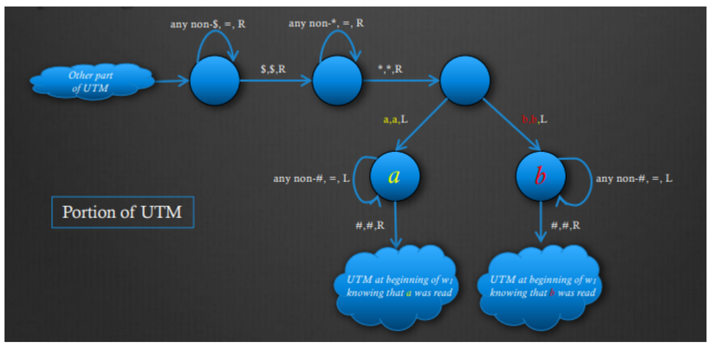
Places an * before

- the newly inserted “dummy row” (to indicate that was the last transition processed)
- The first character in w_2 to indicate the simulated tape head location

This ends the setup phase for the UTM. It is now ready to enter its next phase in which it simulates the TM encoded by w.

The UTM runs right down the tape to read the next character from w_2

It next runs left back up the tape down a different branch in the UTM depending on the character that it read.



The UTM runs right down the tape to find the * in w_1 to discover what state the simulated TM is in (i.e., the To state from the row r_i last processed).

Then, Find the row r_j in w_1 whose

- From state matches the simulated TM’s current state
- Read character matches the last character read by the simulated TM

If no such row r_j in w_1 can be found, then the simulated TM crashes, so crash the UTM. Otherwise

- Move the * in w_1 to precede the matching row r_j
- Update the character in w_2 (i.e., WRITE)
- Move the * in w_2 (L or R).

If the To state in matching row r_j is 2 (i.e., HALT), then the simulated TM HALTs (accepts) its input, so the UTM should also HALT (accept).

In this way the UTM accurately simulates the TM that is encoded as its input in that the UTM

1. accepts strings the encoded TM would accept
2. Rejects (i.e., crash) strings the encoded TM would reject
3. Loops on strings on which the encoded TM would loop

In other words, the UTM is a TM that accepts the language Acc. Therefore, Acc is a recursively enumerable language.

Is Acc a recursive language?

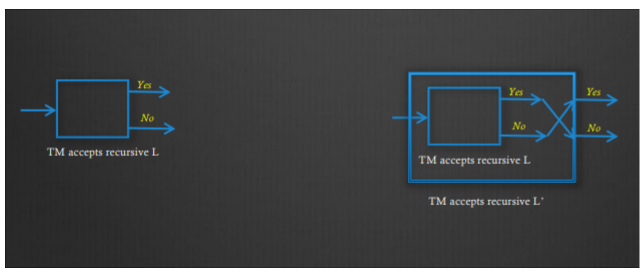
Before we can answer that question, we must first take a closer look at recursive languages by revisiting regular languages.

Recall that if L is a regular language, then its compliment L' is also a regular language.

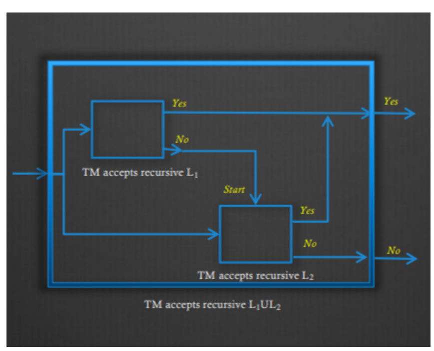
We accomplish this by taking a DFA that accepts L and changing all accepting states to non-accepting, and vice versa.

We can do something with recursive languages to show that if L is a recursive language then its compliment L' is also a recursive language.

We accomplish this by constructing a new TM that swaps "yes/no" decisions.



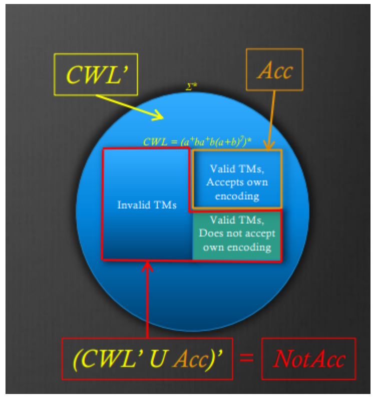
Similarly, the union of two recursive languages is also a recursive language. We accomplish this by constructing a new TM from the two recursive TMs.



Returning to Acc ... we know

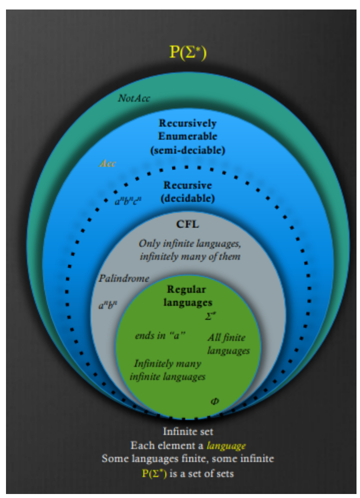
- CWL is a regular language
- CWL′ is a regular language
- CWL′ is a recursive language

  - (CWL $'\cup$ Acc) is a recursive language
  - (CWL $'\cup$ Acc)$'$ is a recursive language



However, (CWL$'\cup$ Acc)$'$ $=$ NotAcc, which we know is not recursively enumerable (much less recursive) ... contradiction.

$\therefore$ Acc is a recursively enumerable language that is not recursive.



- **More on UTM**: Consider the UTM we just described to accept Acc. UTM took as input an encoding of some TM and started by making a copy of that encoding;
  - w_1 – encoding of TM that UTM did not change
  - w_2 – copy of encoded TM (w1) that served as input to simulated TM – UTM frequently changed

  After the initialization phase (i.e., making a copy of w_1) and the UTM entered the simulation phase, it didn't matter that that w_2 was a copy of w_1.

  In other words, we could have placed any input for w_2 and the UTM would have simulated the TM encoded by w_1 processing the arbitrary input we placed as w_2

Now the UTM becomes our *first stored-program computer*

The UTM takes as input (a) an encoded TM and (b) input for that encoded TM, and simulates the TM (a) as it processes input (b)

The UTM is a TM that can simulate any TM processing any data, and hence its name, The Universal Turing Machine (Turing, 1936).

The Universal Turing Machine is the foundation of all computing theory and was the conceptual archetype of the early computer.

It is not a computer's operating system, but rather, the logic that guides a computer's instruction fetch and execution cycle (typically implemented in a computer's hardware).

The first stored-program computer was built by John von Neumann and his colleagues within years of Turing's work.

Contemporary computers implement the UTM differently (for efficiency) but they are all based on the UTM.

1.7.6 The halting problem

- **Revisiting Turing Machines**: If a TM has been processing some string for a long time, how do we know if this is a case when the TM will be looping forever?

  If the TM is going to loop forever, then let's just stop now and declare the input string outside the language accepted by the TM!

- **The halting problem**: There is a problem, The Halting Problem, that states:

  If we are given a Turing Machine T and input string w, can we tell whether T halts on w?

  The Halting Problem question does not care if T accepts/rejects w, only whether T eventually stops (accept or crash/reject) or loops forever.

  Obviously, we cannot simply start processing w with T to find the answer.

  The UTM we presented cannot answer the question because it will loop forever if T loops on w.

  What The Halting Problem is calling for is an algorithm (a TM) that

  - Accepts as input an encoded TM T and input string w
  - The Halting Problem TM would never loop forever (i.e., must be recursive, not merely recursively enumerable)
  - On all possible inputs it would always stop and report either "halts" or "loops" to indicate what T would do when processing w.

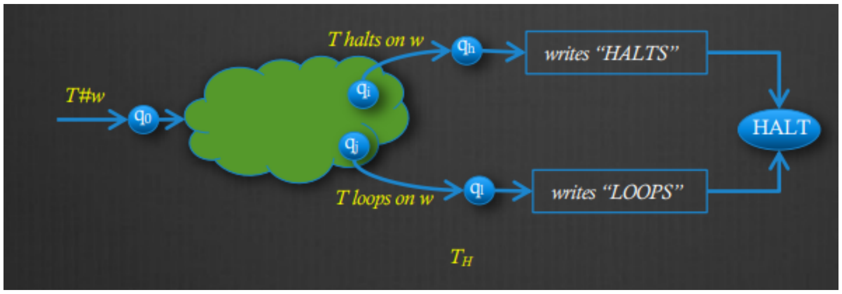
  So the question becomes does such an algorithm exist?... No!

- **Halting problem proof**: Assume a TM exists that solves The Halting Problem, call it T_H.

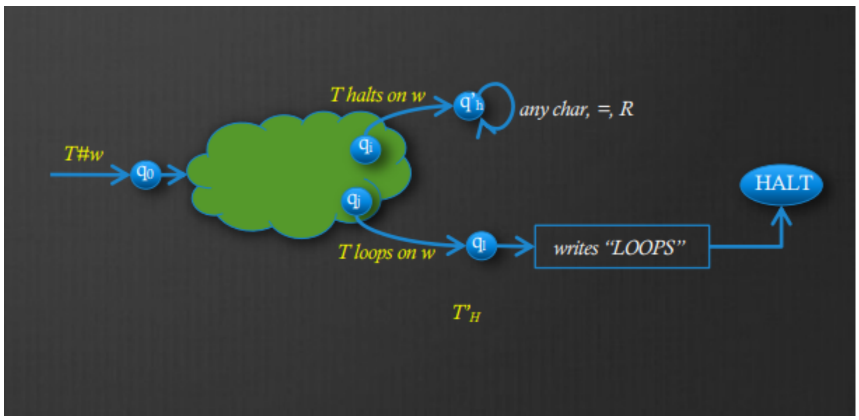
  That is, T_H is a TM that takes as input and encoded TM T, followed by a , followed by some input string w.

  T_H would always terminate and print as its output on its tape

  - "HALTS" if T will halt when processing w, or
  - "LOOPS" if T will loop forever when processing w



We modify T_H to create a T'_H by replacing the TH path from q_i through its "writes HALTS" states, to HALT with a new state that loops forever.



T'_H will either

- Loop forever if input T would halt on its input w, or
- Halt with "LOOPS" written on the tape when input T would loop forever on its input w

Consider how T'_H would process input $T'_H\#T'_H$, that is, ask T'_H how it would process an encoding of itself

As we've seen, there are only two possible responses from T'_H.

Case I: T'_H actually loops forever when processing $T'_H\#T'_H$ (i.e., enters q'_h). T'_H only loops forever when its input T would halt on T's input w

Since we processed $T'_H\#T'_H$ that would mean that T'_H must halt when processing its own input, contradiction.

Case II: T'_H actually halts with "LOOPS" when processing on $T'_H\#T'_H$ (i.e., enters q_1). T'_H only halts with "LOOPS" when its input T would loop forever on T's input w.

Since we processed $T'_H\#T'_H$ that would mean that T'_H must loop forever when processing its own input, contradiction.

This tells us that T'_H cannot exist, which in itself is not a proof that there is no solution to The Halting Problem

However, T'_H was a legitimate modification of T_H which we assumed to exist as a solution to The Halting Problem.

Since T'_H cannot exist and it was based on a legitimate modification of T_H, we conclude that T_H cannot exist

$\therefore$ There is no algorithm that can solve The Halting Problem.

1.8 Complexity theory

- **Intro**: The field of computational complexity looks at the resources required to solve problems. Or, more generally, how much resources does it take: time, memory space, number of processors, bandwidth, and so fourth. It is a given that the problem is solvable.

- $\mathcal{P}$ **and** $\mathcal{NP}$: Regarding time, we define two sets of languages. The set $\mathcal{P}$ is those languages that can be solved in polynomial time, and $\mathcal{NP}$ is the set of those languages that can be solved in polynomial time on a non-deterministic TM. The set $\mathcal{P}$ is somewhat viewed as the reasonably tractable problems, buy many problems of practical interest have been shown to be in $\mathcal{NP}$. Thus, the question of whether all of $\mathcal{NP}$ is in $\mathcal{P}$ (that is, whether $\mathcal{P} = \mathcal{NP}$) is of fundemental importance. This question remains unsolved.

1.8.1 Time complexity

- **Time complexity**: The time complexity of a set of problems is how much item is needed to solve them. The time complexity of a language is how much time is needed to decide membership in it.

  The goal is to odetermine how the resources required depend on the size of the input. n always denotes the size of the input. Then, the running time is the number of steps as a function of n.

  It is important to note that we analyze the **worst case**. It doesn't matter that some instances can be done quickly, what matters is if *every* instance can be done quickly. A TM is said to run in time $T(n)$ if for all inputs w, it halts within $T(|w|)$ steps.

- **Big-O**: It is important to note that constants do not matter. If a machine runs in $2n^2, 7n^2$, or $1000n^2$ steps is immaterial. In fact, we cannot care about the constants. For, the exact time will depend very heavily on what the atomic operations are. And nobody uses a laptop TM anyway. If the number of steps in a certain TM is proportional to n^2, we say the TM runs in $O(n^2)$ time, or "*order* n^2" time, meaning there exists some constant c such that the Tm runs in at most cn^2 steps for any input of length n. The order notation says how the worst case running time grows as n gets large.

- **Polynomial time**: The collection of all problems that can be solved in polynomial time is called $\mathcal{P}$. That is, a language L is in $\mathcal{P}$ if there exists a constant k and a TM that decides L that runs in time $O(n^k)$.

- **Complexity class**: It is common to call a set of related languages a *complexity class*, or just a class. The class $\mathcal{P}$ roughly captures the collection of practically solvable problems.

- **Polynomially related**: Two models of computation are polynomially related if there is a polynomial p such that if a language is decidable in $T(n)$ time on one model, the nit is decccccidable in time $p(T(n))$ on the other.

- **Nondeterministic time**: The collection of all problems that can be solved in polynomial time by a non-deterministic machine is called $\mathcal{NP}$. That is, a language L is in $\mathcal{NP}$ if there exists a constant k and an NTM that decides L that runs in time $O(n^k)$. It should be clear that

$$\mathcal{P} \subseteq \mathcal{NP}.$$

- **Theorem**: Let L be a recursive language. If there is an NTM for L that runs in time $T(n)$, then there is a deterministic TM for L that runs in time $O(C^{T(n)})$ for some constant C
- **Conjecture**: $\mathcal{P} \neq \mathcal{NP}$, however, we do not know.

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 sub-problems

  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. $\Longrightarrow$ `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,

    ```
    #include "./dir/headerfile" // Without -I
    #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[*libraryname*]**: 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.

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

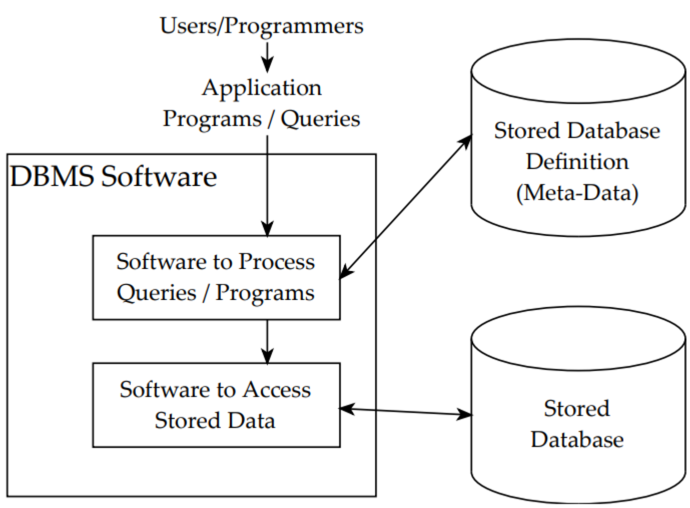
- **Manipulating a Database**
  - retrieve specific information in a query
  - update the database to include changes
  - generate reports from the data

  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

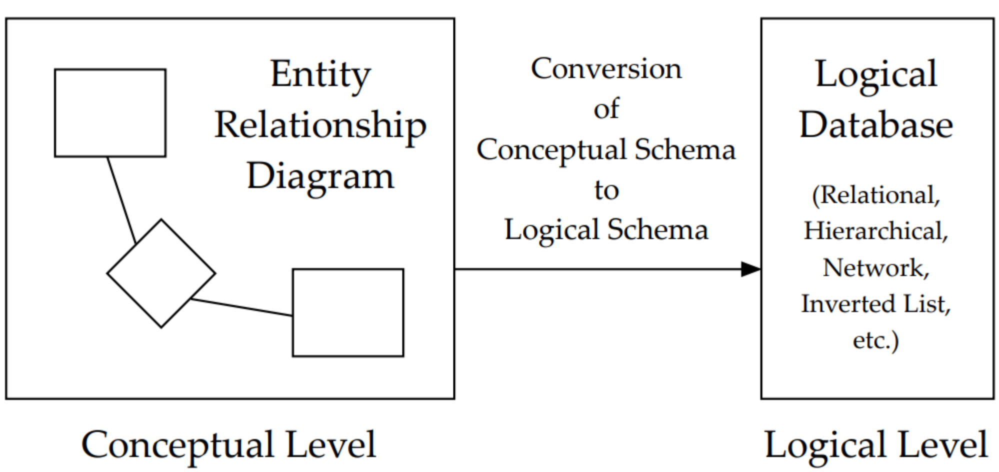
- efficient file access which allows us to “find the boss of Susie Jones”
- allows us to “navigate” within the data
- allows us to combine values in 2 or more databases to obtain “information”

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

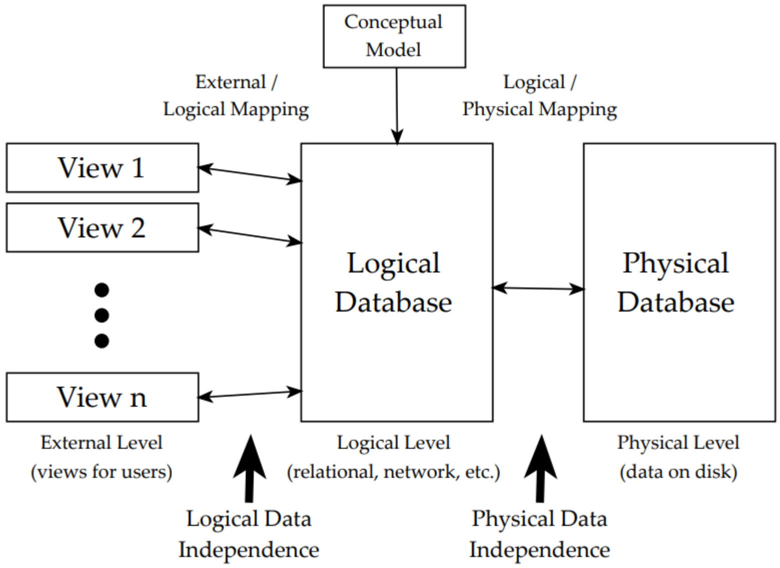
- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Data Control Language (DCL)
- query language access data
- operations such as add, delete, and replace

- **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.

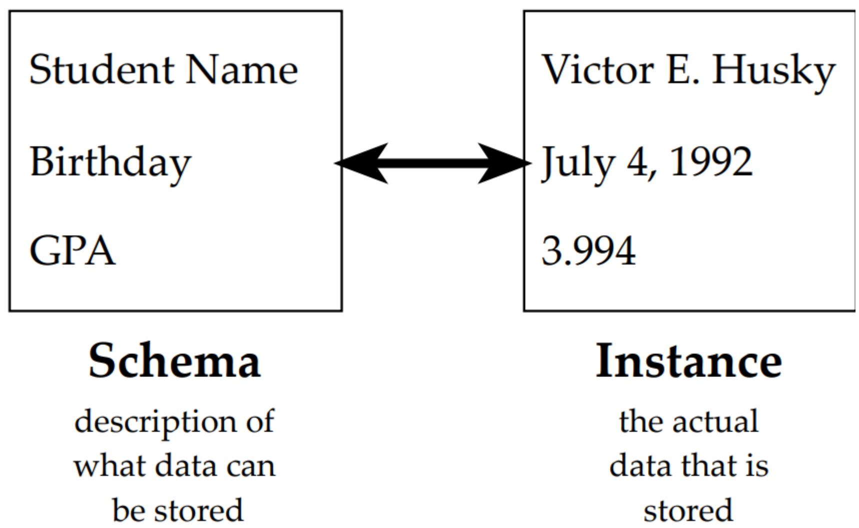
  Note: Obviously, some of these would require more than just software - offsite back-ups, etc
- **Use of Conceptual Modeling**:



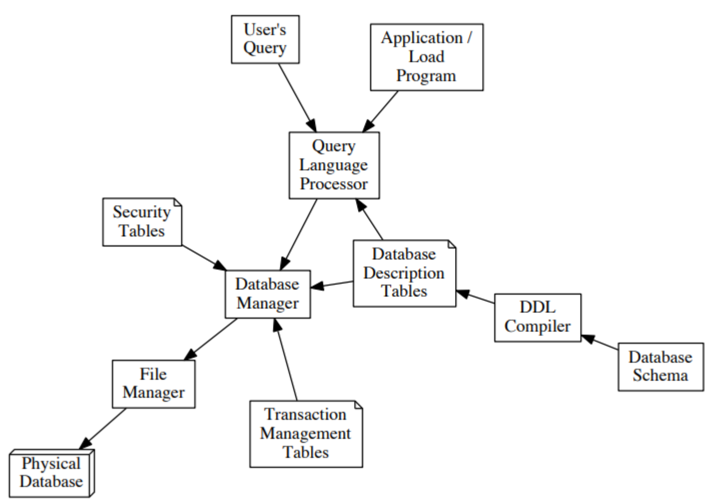
- **Leveled Architecture of a DBMS**:



- **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**:



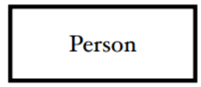
- **Overall DBMS Usage Scenario**: Database Administrator (DBA) define the conceptual, logical, and physical levels using DDL. DBMS software stores instances of these in schemas. User defines views (External Schema) in DDL. User accesses database using DML
- **Advantages of a Database**:
  - Controlled 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

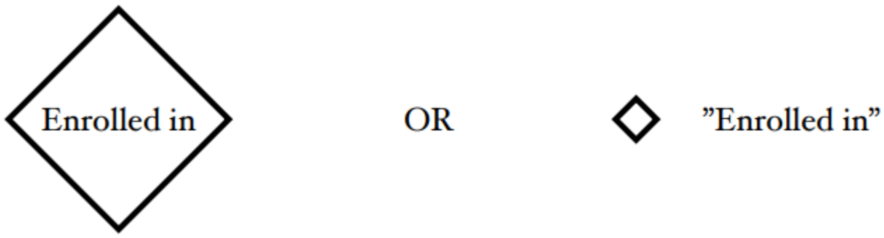
3.2.1 Definitions and theorems

- **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.

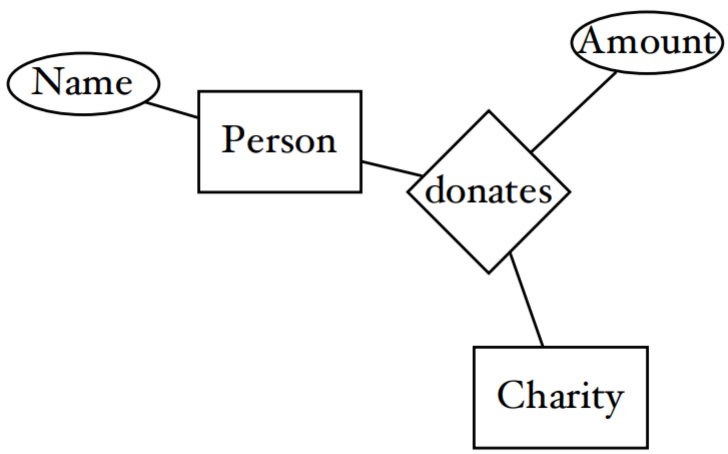
[1] "Non-atomic data values" refer to data structures or values that are composed of multiple components, as opposed to atomic data values, which are indivisible and represent a single value.



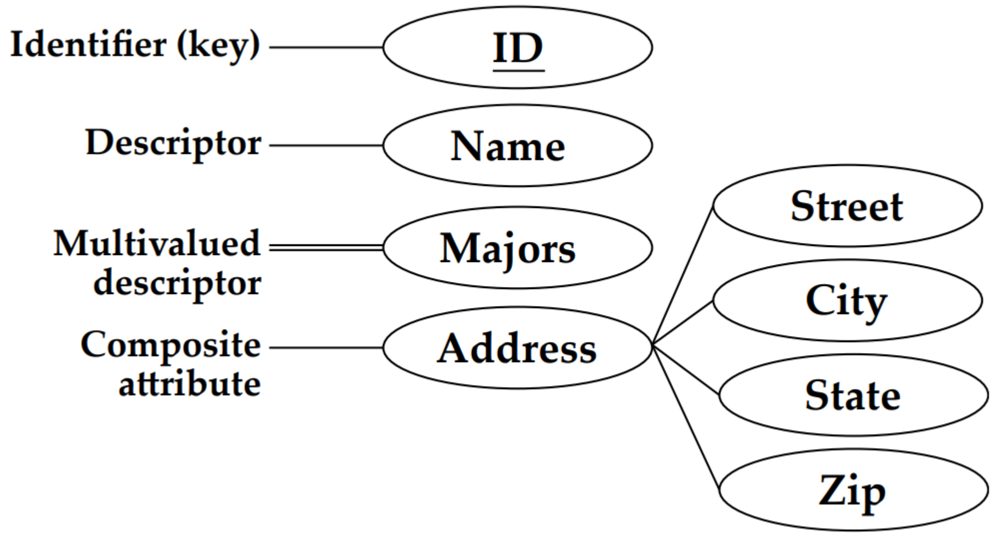
- **Relationships**: Relationships connect one or more entities together to show an association. A relationship *cannot* exist without at least one associated entity. Graphically represented as a diamond with the name of the relationship inside, or just beside it



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

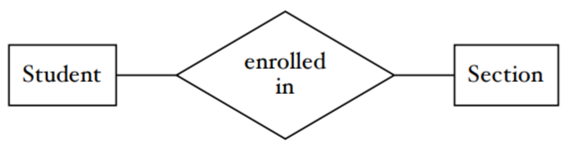


- **Attributes on Entities**: When an attribute is attached to an entity, it is expected to have a value for every instance of that entity, unless it is allowed to be null. For instance, in the diagram above, Name was an attribute of Person. Every person that we store data about will have a value for 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**:

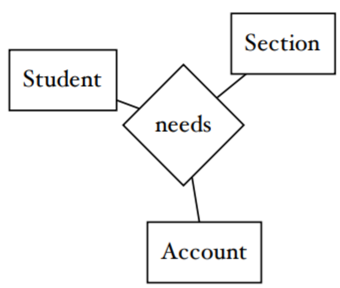


- **Degree of a Relationship**: The degree of a relationship is defined as how many entities it associates. If one entity is associated more than once (such as with a recursive relationship), then the degree counts each time it is referenced.

  - binary



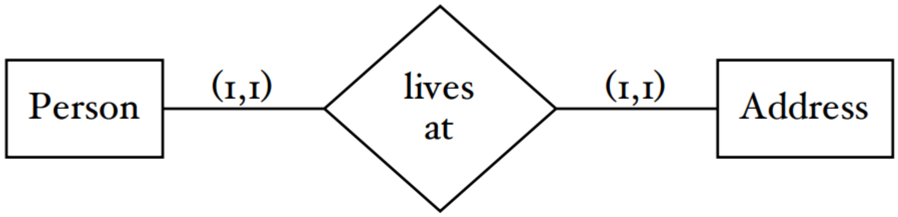
  - ternary



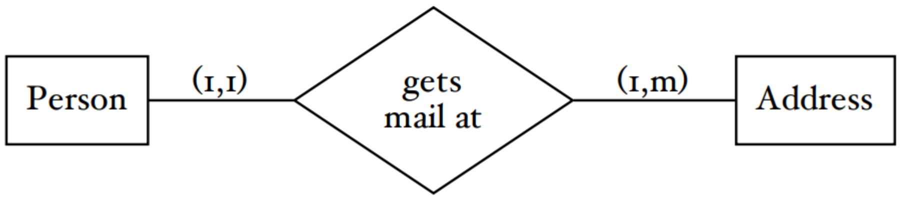
Note: There is no limit to how many entities there can be in a relationship. After binary, and ternary, we start to call the relationships n-ary, where n is the degree

- **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

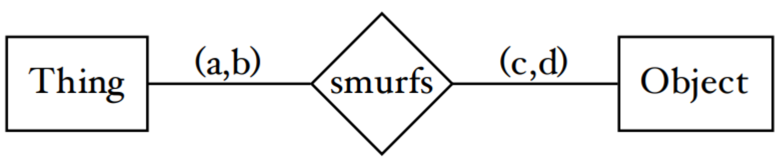


- one-to-many



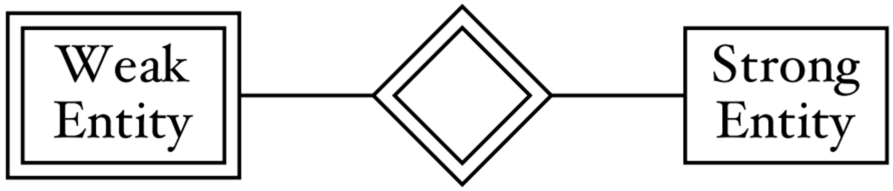
Together (from the image) both sides make up the connectivity, to refer to a single side, we use the term ”cardinality”, ie the cardinality of a person is (1,1). If we hold Address constant (We know a specific address and are therefore refering to that), how many persons may live at that address, in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

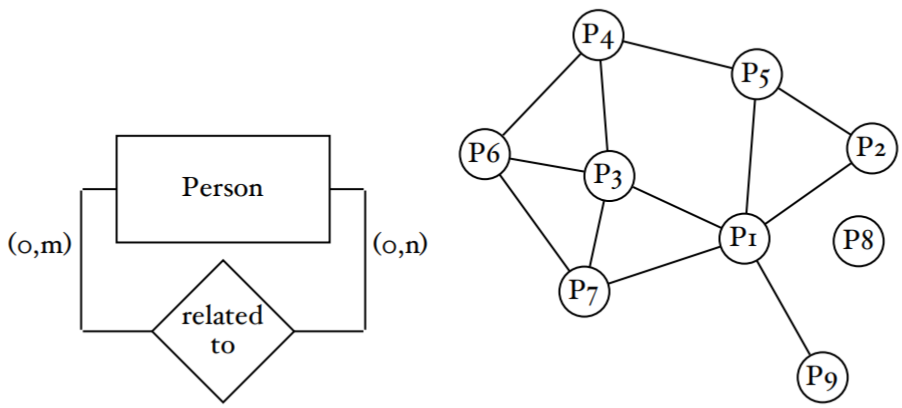


- **Weak Entities**: Sometimes you may run into an entity that depends upon another entity for its existence. The weak entity is a tool you can use to represent this.:w

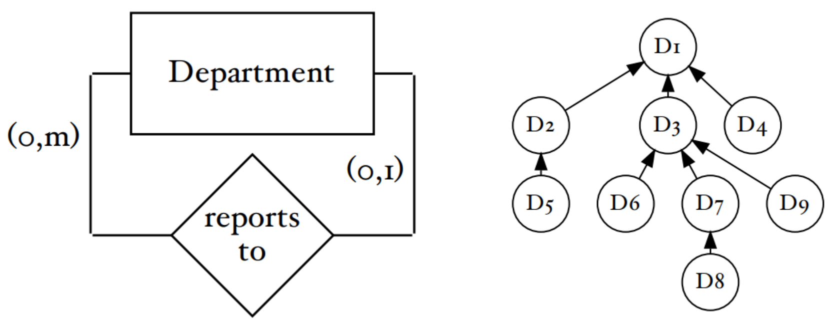
  Weak entities are written like normal entities, except that they have a double rectangle outline. The relationship that connects the weak entity to the strong entity it depends upon will be written with a double diamond. This does not mean that the relationship is weak. It is just to indicate upon which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

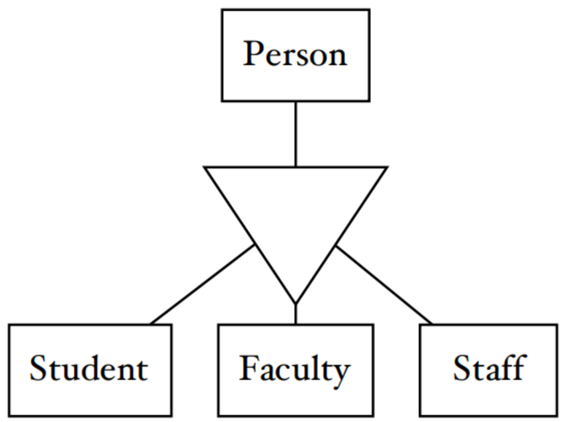


- **Entity or Attribute?**: Sometimes it isn't clear whether something should be an entity or an attribute of some other entity. Usually the decision will come down to how complicated it is to store the data, and how important it is. If it ends up being used in multiple places, it might be a clue that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

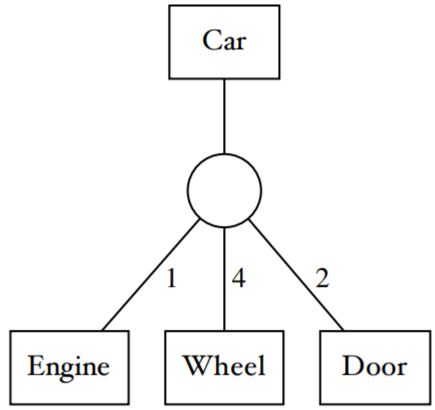
- **IS A Inheritance**: This type of inheritance happens when you have a supertype and one or more subtypes that are members of the supertype. Denoted by an upside-down triangle, with the supertype on top, and the subtypes coming out the bottom.



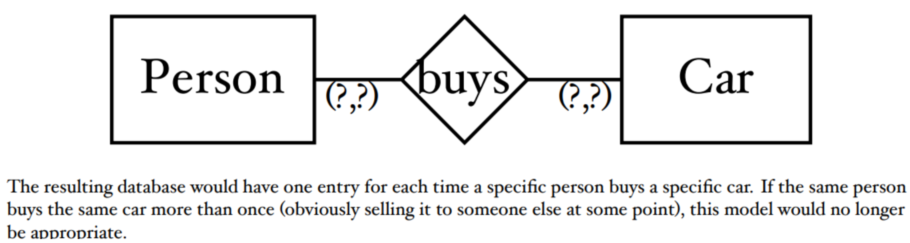
- **Defining IS-A inheritance**: There are two things you need to choose when using IS-A inheritance:
  - **Generalization (no) vs. specialization (yes)**: can the supertype occur without being a member of the specified subtypes?
  - **Overlapped (yes) vs. disjoint subtypes (no)**: is it possible for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

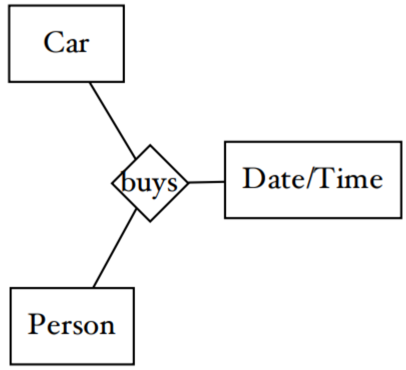


- **Warning about IS-PART-OF**: The IS PART OF inheritance operator does have its uses, but it is not very commonly used, If you see something involving a certain number of things being present, there are several possibilities
  - Sometimes a number is specified that isn't actually important for what 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.

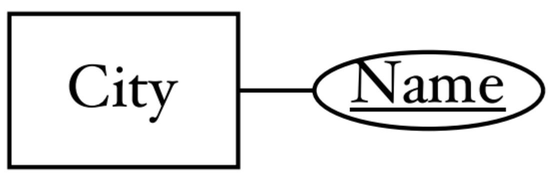
Adding a new entity to the relationship for the date/time of the purchase can fix this problem.



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

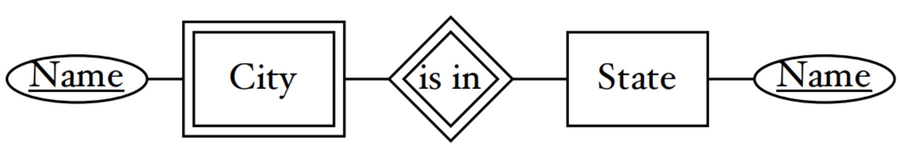
- **Weak Entities - Introduction**: So far, all of the entities we have used have been things that stand on their own. There are some situations where we are modeling an object for which we certainly need to store data, but the items exist only in the context of some other entity. Many of these examples 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	$\dots$	A_n
x_1	x_2	x_3	$\dots$	x_n
y_1	y_2	y_3	$\dots$	y_n
$\dots$	$\dots$	$\dots$		$\dots$

- **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:

  $$X \to Y.$$

  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 $\to$ StudentFirstName, StudentLastName, Birthday**: If I identify a student using their ZID, that student has one first name, last name, and birthday
  – **StudentFirstName $\not\to$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 $\to$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**($\underline{\text{a}}$,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**(<u>EmpID</u>, <u>Project</u>, 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:

$$\text{EmpID, Project} \rightarrow \text{Supv, Dept, Case}$$
$$\text{EmpID} \rightarrow \text{Supv, Dept}$$
$$\text{Supv} \rightarrow \text{Dept}$$

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(<u>EmpID</u>, <u>Project</u>, 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 $\rightarrow$ Supv, Dept, Case
    EmpID $\rightarrow$ Supv, Dept
    Supv $\rightarrow$ Dept.

    In this case, the FD that causes our relations to violate 3NF is (Supv $\rightarrow$ 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 $\rightarrow$ 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

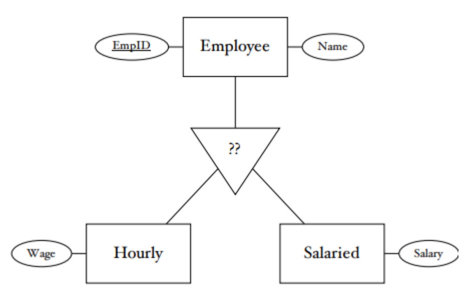
- **Entities like date**: there is no reason to make a relation for a "Date" entity or similar. The single value for the date is enough to determine it, and any other data associated with it is generally happening through a relationship anyway. Think about what data would go into such a table and how 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(<u>EmpID</u>, Name)

   Would become something like:

   Employee(<u>EmpID</u>, 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(<u>EmpID</u>, 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(<u>EmpID</u>, 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

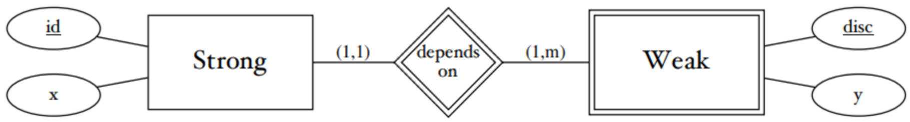
   Employee(<u>EmpID</u>, Name

   Hourly(<u>EmpID†</u>,Wage)

   Salaried(<u>EmpID†</u>, 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(<u>id</u>, 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(<u>id</u>†, <u>disc</u>, y)

The <u>id</u> 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

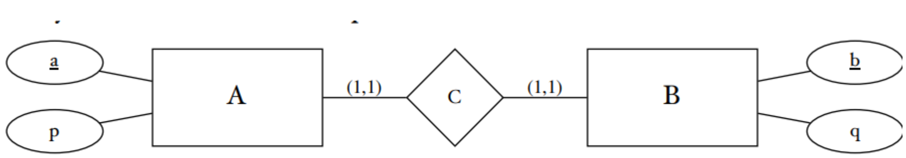
- **Handle relationships**: The relationships will be handled in order from lowest degree to highest degree, and within that, from simplest cardinality (one-to-one) to more complicated cardinalities (many-to-many, etc.).

  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(<u>a</u>,p) *and* **B**(<u>b</u>, 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:

A(<u>a</u>,p,b†) *or* **B**(<u>b</u>, q, a†)

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.

   A(<u>a</u>,p) *and* **B**(<u>b</u>, 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:

   B(<u>b</u>,q,a†)

   The only FD is

   $$b \to a.$$

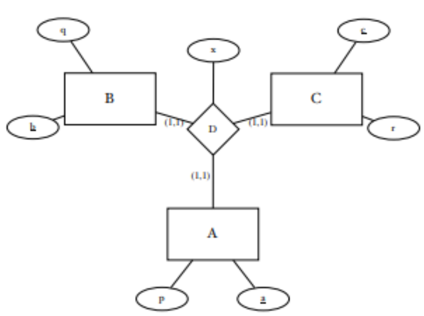
3. **Binary many-to-many Relationships**: In a binary relationship, we will already have made a relation for each of the entities involved.

   A(<u>a</u>,p) *and* **B**(<u>b</u>, 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.

   C(<u>a</u>†, <u>b</u>†, x)

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



So we have

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

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

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

would just have less functional dependencies and therefore less options to choose from (see table above)

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:

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

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.

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

.

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.

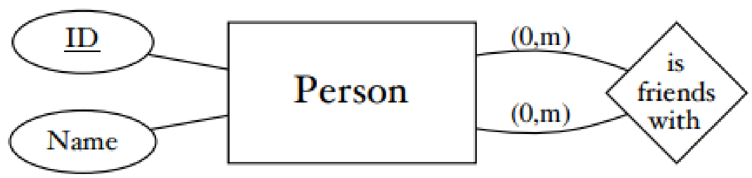
$$\mathbf{Department}(\underline{\text{DeptNo}}, \text{Name})$$

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(SSN, 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(SSN†, CLSYEAR, GPA, TOTALHRS)

```sql
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

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

  To add multiple columns/attributes:

  ```sql
  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.

  ```sql
  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.

  ```sql
  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:

```sql
SHOW TABLES;
```

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

```sql
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.

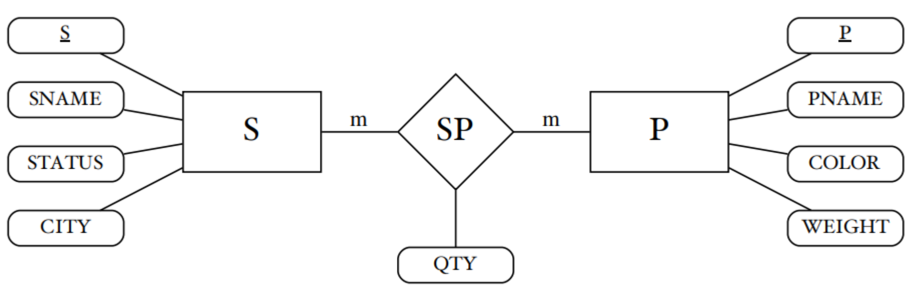
```
SELECT [DISTINCT|ALL] <column_list> # most common, show row data
    FROM <table_list>
    [ WHERE <where_exp> ]
    [ GROUP BY <group_key> ]
    [ HAVING <having_exp> ]
    [ ORDER BY <sortcols> ] ;

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

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

List the full details of all suppliers.

```sql
SELECT * FROM S;
```

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

```sql
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.

```sql
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.

```sql
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

```sql
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.)

```sql
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.

```sql
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

```sql
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

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

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

```sql
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:

```sql
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.

```sql
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 Average	Retail	Overall
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'**:

```sql
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.

```sql
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

```sql
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.

  ```sql
  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.

  ```sql
  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.

```sql
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.

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

We can of course negate this with NOT

```sql
SELECT ...
    WHERE ... NOT BETWEEN val1 AND val2;
```

3.7 Frontend: Html

- **Why HTML for databases?**: HTML is a relatively easy way to provide a graphical user interface to a user. This interface can solicit from the user the data that is needed to perform operations on the database, as well as present to the user the data that is returned in a structured way
- **HTML vs. XHTML**: As a web browser goes through evaluating all of the HTML code it encounters, it has several possible modes of operation.
  - The HTML-only parser mode is very loose. If you make a mistake, it will try to guess what you meant and continue anyway. This can cause weird behavior with no warning that anything was wrong.
  - The XHTML parser performs more checks, which can help you locate and fix your problem areas more quickly.
- **Tree-Like Structure**: A well-formed HTML document should have a structure based upon the data structure known as a tree. This tree is formed by the elements present in the document, which can be used as containers of one or more other elements.
- **Elements**: There are three basic types of elements that you will see in HTML.
  1. **Plain Text**: This is just normal text, in whatever encoding (ASCII, EBCDIC, Unicode) the document is using. Note that HTML-specific characters will need to be properly escaped.
  2. **Comments**: These start with `<!--` and end with `-->`
  3. **HTML tags**: These are special keywords, surrounded by angle brackets (< and >). For each tag element, there will be a start tag and an end tag

     The start, or open tag is, in its most simple form, just the keyword in angle brackets, eg. <tag>.

     The end, or close tag, when written separately, has a forward slash / after the first angle bracket, eg. </tag>.
- **Plain Text**: Plain text elements will be displayed (or not displayed) in a manner appropriate to where they occur in the document.
- **A Note on Comments**: Comments are nice, and should still be used when they would be helpful. However, since the comments are sent to the user from the server every time the page is loaded, it might be beneficial, from a performance standpoint, to keep them to a minimum.
- **HTML Tags**: They can be written separately, usually when they are surrounding something, such as <div>Text Here</div> - note the / at the beginning of the end tag.

  If there is nothing between the start and end tag, there is a shorthand, <meta/> – the / at the end of the tag means that the end tag is included with the opening tag, so it is equivalent to <meta></meta>, with nothing inside.

  If the element is generally expected to have content, you should not use this shortcut.
- **Attributes**: It is possible to specify attributes of an HTML tag. These get written as part of the start, or opening tag, and they can either be name=value pairs (attrname1 or attrname2 below), or just the name of an attribute if it is boolean if it is desired to set it to true (boolattr below).

```
<tag attrname1="attr1value" attrname2="attr2value" boolattr>
```

Each type of tag will have a different set of applicable attributes. The values specified for these attributes will control the behavior of the element.

- **Basic HTML Document**:

```
<html>
    <head>
        <title>Page Title Here</title>
        <meta charset="UTF-8"/>
    </head>
    <body>
        <h1>Big Heading Here</h1>
        <p>A paragraph here.</p>
    </body>
</html>
```

- **<html> Element**: The <html> element should be the one of the first elements to appear, and will contain all of the other elements in your document.

  There should be only one per document.

- **<head> Element**: The <head> element is meant to contain header information. Among other things, this can include:
  - <title> The title that will be displayed for this document in either a tab button or the window caption. The title is the text between the start and the end tag, not an attribute.
  - <meta> Various information about the document that you would like to make available to search engines, etc.

  Notice that any text you put in the <head> tag will not show up when the document is rendered.

- **<body> Element**: The <body> element will contain the body of your document. Any text that you would like to display to the user should be contained by this element

- **Working with Text**: Before we start working with text, there are a few things we should address.
  1. whitespace (spaces, tabs) is mostly ignored after the first space
  2. newline character in the source do not get shown as newlines when the document is displayed
  3. characters that could be interpreted as part of the HTML markup may need to be escaped to show up as text-only

- **Special Characters and Escape Codes**:
  - `>` greater than symbol (because used to identify tags)
  - `<` less than symbol (because used to identify tags)
  - `&` ampersand symbol (because used for these codes)

  - ` ` non-breaking space (this is a space that will not be ignored)
  - `©` copyright symbol, © (not a part of ASCII alphabet)
  - `✏` Unicode character with decimal value 9999
  - `&#xffff;` Unicode character with hexadecimal value ffff

- **Headings – <h1> to <h6>**: The tags, <h1> through <h6>, provide various levels of headings. <h1> is the first heading, and will have the largest text, with the others getting progressively smaller in importance.
- **Paragraphs – <p>**: The <p> element is used to indicate that the text it contains is meant to be a paragraph. In general, this means that it will be printed as a single block of text, wrapping automatically as its text reaches the end of the line.

  What if we wanted to format an address, with the name on one line, the street address on another, and the city/state/zipcode below that?

```
<p>Northern Illinois University</p>
<p>1425 West Lincoln Highway</p>
<p>DeKalb, IL 60115</p>
```

- **Line breaks**: There is a special tag,
, the line break, that can help in this situation.

```
<p>Northern Illinois University<br />
    1425 West Lincoln Highway<br />
    DeKalb, IL 60115</p>
```

  Note: Some purists will argue that the
 tag should not be used, because it's purely visual, and the purpose of HTML is to denote structure. They are not entirely wrong, but it's more work to do it other ways, so it will work in a pinch.

- **Unordered Lists – **: It is possible to create unordered (bulleted) lists with <ul> and <li> elements.

```
<ul>
    <li>No</li>
    <li>Particular</li>
    <li>Order</li>
</ul>
```

- **Ordered Lists – **: It is possible to create ordered (numbered) lists with <ol> and <li> elements.

```
<ol>
    <li>First</li>
    <li>Second</li>
    <li>Third</li>
</ol>
```

The type of numeral used for ordered lists can be controlled by CSS, or via the type attribute of the <ol> tag. Its possible values are (1, A, a, I, and i).

- **Images – <img>**: It is possible to embed images into your HTML documents. This is done with the <img> tag. On its own, the <img> element cannot do much, because it needs more information. It will be necessary to supply the path to the file containing the image to display as the src attribute.

  In HTML, paths can be specified in any of three ways:

  - **relative paths** – these are paths starting from the directory the current HTML document is being loaded from. These will never begin with a forward slash, `/`.
  - **absolute paths (local)** – these are paths starting from the root of the webserver that is serving the current document, these will always begin with a forward slash, `/`.
  - **absolute paths (remote)** – these are paths that are on other servers. They will begin with the protocol specifier of the server. (`http://`, `https://`, `ftp://`, etc.)

  With all three of these types of paths, the steps along the path will be separated by a forward slash, `/`.

- **Images - Useful Attributes**: Some useful attributes for the `<img>` element:

  - **height** – height of image (in pixels) or as a percentage of screen (with a `%` at the end)
  - **width** – width of image (in pixels) or as a percentage of screen (with a `%` at the end)
  - **alt** – alternate text, which will show when hovering, or may be read aloud for accessibility purposes

  The `height` and `width` attributes can be used to scale the image, but the same image file is downloaded either way. It is best to make the image exactly the size you intend to display it using some image editing tool, rather than resizing it on the page. The best use for `height` and `width` is to allow the browser to know the size of the image before downloading it, so the page doesn't have to resize as it loads.

- **Links**: One of the features that originally made HTML popular was the idea of the hyperlink. Now we just call them links.

  They are portions of the document that can be clicked to navigate to somewhere else.

  They are added to the HTML using the anchor tag, <a>. The path for the destination of the link is specified with the href attribute, and the text that forms the link will be whatever is contained by the anchor element.

```html
<a href="other.html">Relative Path Link</a>
<a href="/some.html">Absolute Path Link</a>
<a href="http://www.niu.edu"> Link on another server</a>
```

- **Tables**: Something that comes up very often when dealing with relational databases is the table. There is support for drawing them in HTML. This is done with a set of four elements that work together:

- `<table>` – This element starts a table, the others should only occur inside one of these.
- `<tr> (table row)` – This element is a row within a table. It will contain one or more of either of the following two.
- `<th> (table heading)` – This element is a table heading cell. The text to display as a column label should go inside.
- `<td> (table data)` – This element is a table data cell. The text inside should be the data to be shown in the cell.

Tables in HTML are always specified in row-major order. The cells go from left to right inside of rows. There is not a way to specify it columnwise.

```
<table>
    <tr>
        <th>Topping</th>
        <th>Price</th>
    </tr>
    <tr>
        <td>Sausage</td>
        <td>$0.90</td>
    </tr>
    <tr>
        <td>Pepperoni</td>
        <td>$1.00</td>
    </tr>
</table>
```

- **Forms**: In HTML, forms are used to collect user input and submit it to a server for processing. They are fundamental to creating interactive web applications, allowing users to input information, like login credentials, feedback, or search queries

```
<form action="submit_page.php" method="post">
    <!-- Form elements like input fields, buttons, etc. go here -->
</form>
```

  - **action:** Specifies the URL where form data should be submitted.
  - **method:** Defines the HTTP method for submitting the form. Common values are:
    1. **GET:** Sends form data as URL parameters (query strings), typically used for non-sensitive data.
    2. **POST:** Sends form data in the request body, more secure for sensitive data.

- **Input**: The <input> tag in HTML is used to create interactive fields in forms that collect user input. The <input> tag is versatile and supports various types of inputs by setting the type attribute to specify the kind of data the field should accept.

  1. **Text Input:** Collects single-line text input, like names or emails.

```html
<input type="text" name="username" placeholder="Enter your name">
```

2. **Password Input:** Similar to text but hides characters as they're typed.

```html
<input type="password" name="password" placeholder="Enter your password">
```

3. **Radio Buttons:** Allow users to select a single option from a set.

```html
<input type="radio" name="gender" value="male"> Male
<input type="radio" name="gender" value="female"> Female
```

4. **Checkboxes:** Let users select multiple options independently.

```html
<input type="checkbox" name="subscribe" value="newsletter"> Subscribe to newsletter
```

5. **Submit Button:** Sends the form data to the server.

```html
<input type="submit" value="Submit">
```

6. **Color Picker:** Allows users to pick a color from a color wheel or enter a color code.

```html
<input type="color" name="favcolor" value="#ff0000">
```

7. **Date Picker:** Provides a calendar interface for selecting a date.

```html
<input type="date" name="birthdate">
```

8. **Email Input:** Accepts and validates email addresses.

```html
<input type="email" name="email" placeholder="Enter your email" required>
```

9. **File Upload:** Allows users to upload files from their device.

```html
<input type="file" name="profilePicture">
```

10. **Hidden Input:** Stores data that isn't visible to the user but is submitted with the form.

```
<input type="hidden" name="userID" value="12345">
```

11. **Image Button:** Acts as a submit button but uses an image instead of text.

```
<input type="image" src="submit_button.png"
↪ alt="Submit" width="50" height="50">
```

12. **Month Picker:** Allows users to select a specific month and year.

```
<input type="month" name="startmonth">
```

13. **Number Input:** Accepts numeric values, optionally within a specified range.

```
<input type="number" name="quantity" min="1" max="10"
↪ step="1">
```

14. **Range Slider:** Allows users to select a number within a specified range by sliding a handle.

```
<input type="range" name="volume" min="0" max="100"
↪ step="10">
```

15. **Reset Button:** Clears all input fields in the form to their default values.

```
<input type="reset" value="Reset">
```

16. **Search Field:** Provides a field for entering search queries.

```
<input type="search" name="query"
↪ placeholder="Search...">
```

17. **Telephone Input:** Used for entering telephone numbers, with optional pattern for specific formats.

```
<input type="tel" name="phone"
↪ placeholder="123-456-7890"
↪ pattern="[0-9]{3}-[0-9]{3}-[0-9]{4}">
```

18. **URL Input:** Accepts and validates URLs.

```
<input type="url" name="website"
↪ placeholder="https://example.com">
```

- **Form validation**: HTML5 provides several attributes for form validation:
  - required: Ensures a field must be filled out.

```html
<input type="email" name="email" required>
```

– pattern: Specifies a regular expression pattern for input validation.

```html
<input type="text" name="username"
↪ pattern="[A-Za-z]{3,}">
```

– min and max: Define minimum and maximum values for number fields.

```html
<input type="number" name="age" min="1" max="100">
```

- **Dropdown List (Select):** Allows users to select from a list of options.

```html
<select name="country">
    <option value="usa">USA</option>
    <option value="canada">Canada</option>
</select>
```

- **Textarea:** For multi-line text input, such as comments or descriptions.

```html
<textarea name="comments" placeholder="Enter your
↪ comments"></textarea>
```

- **Buttons in html**: We can create buttons with the button tag

```html
<button> </button>
```

- **Nav tags**: The <nav> tag in HTML is used to define a section of navigation links on a web page. It's a semantic element introduced in HTML5 that helps improve the structure and accessibility of a website by explicitly marking sections dedicated to navigation. This tag is typically used for primary navigation elements like menus, tables of contents, or other links that users can use to navigate a site.

```html
<nav>
    <ul>
        <li><a href="index.html">Home</a></li>
        <li><a href="about.html">About Us</a></li>
        <li><a href="services.html">Services</a></li>
        <li><a href="contact.html">Contact</a></li>
    </ul>
</nav>
```

- **Div containers**: The most commonly used container in HTML, <div> stands for "division."

It's a generic container that can hold any other HTML elements.

Used for grouping elements to apply CSS styles or JavaScript functionality.

```
<div class="container">
    <h1>Welcome to My Website</h1>
    <p>This is a sample container.</p>
</div>
```

- **Section container**: A semantic container used to group related content.

  Useful for sections like articles, services, or product listings.

```
<section class="features">
    <h2>Features</h2>
    <p>Our product offers the following features:</p>
</section>
```

- **<header>, <footer>, and <aside> Tags:**
  - **<header>:** For header content, like logos or navigation.
  - **<footer>:** For footer content, like copyright or contact information.
  - **<aside>:** For content related to the main content, like sidebars or ads.
- **Span**: The <span> element, can be used to group inline elements only. So, if you have a part of a sentence or paragraph which you want to group together

3.8 PHP

- **Components of web services**:
  - **Web browser**: formats and displays Web pages
  - **Web server**: sends Web pages to browsers and lets site visitors enter and request information
  - **Information server**: accepts requests from the Web server and uses its stored data to respond appropriately

- **Database server**: A database server is a kind of information server, it stores information in databases

  Web servers, etc. connect to the database server to send queries or update data

- **Web pages and HTML**: An HTML document is used to create the format and structure of a Web page

  HTTP is a communication protocol that specifies how two or more things are expected to interact on the Web

- **Web server**: a computer system that responds to requests for Web pages by processing the requests and returning a new Web page to the browser

- **Preparing to use php**: The php language was designed to help developers create dynamic and data-driven web pages

  php interacts with one main external tool, the MySQL database management system, to access data stored in a database

  MySQL must be installed on a functional Web server to interact with php, but this is a relatively easy step in setting up the php environment

  php is a server-side scripting language that you can embed into HTML documents. You can also embed HTML in php scripts

  php scripts are parsed and interpreted on the server side of a Web application

  php is popular with web developers and web designers alike, and is powerful and easy to use

  To start working with php, you can create a script that contains HTML code

- **Methods of using php in html**: There are two ways

```
<?php
    ...
?>

<SCRIPT LANGUAGE="php"
    ...
</SCRIPT>
```

- **PHP example**: We can create a file called example.php, that is essentially an html file, but can contain php code in <?php ... ?> blocks. For example,

```
<body>
    <?php
        print("Hello world");
    ?>
</body>
```

Note that lines in php are terminated with a semicolon

- **Displaying php Output**: php has two functions that allow display: echo and print

  The only difference between echo and print is that the print function returns a 1 or 0 integer (denoting success or failure, respectively), for the contents of the function being displayed

  Also, be aware that if you want to send php reserved characters (such as double quotations) to the Web browser within the echo command, you must use the backslash character

- **Defining php Variables**: Variables in php are preceded with a dollar sign ($) and contain either letters or numbers

  php is called a loosely typed programming language, meaning that you don't have to predefine your variables; you can define and use them as needed

  You do have to follow certain rules for naming a variable:

  – Precede the variable name with a dollar sign
  – Assign the variable a meaningful name that you can remember in the future
  – Name the variable with uppercase or lowercase letters, numbers, or the underscore (_) character
  – Do not allow the first character after the ($) to be a number
  – Variable names are case sensitive
  – Assign the variable an initial value with a single equals (=) sign

```
<?php
    $x = 3.14;
    echo $x;
    echo "<br/>";
    print($x);
?>
```

- **Variable scope**: If a variable is defined at the start of a php file, it stays in memory until the end of that file. This is known as the variable's scope

  If a variable is assigned a value of 5 in one php file, and that file calls another php file that has a variable of the same name, then the first variable is terminated and its value is lost

One major distinction that relates to a variable's scope involves the processing of web-based forms

Any variables that are defined within a php/HTML form and sent to the server with the form's Post method are automatically sent with the called Post action and named in php by the same name used in the HTML form

- **Variable data types**:
  - **Boolean**:
  - **Integer**:
  - **Float or double**:
  - **String** :
  - **Object**: An instance of a class
  - **Array**: Ordered set of keys and values
  - **Resource**: Reference to a thirdparty resource (a database for example)
  - **NULL**: An unintialized variable
- **Gettype function**: Test the type of a variable by using the built-in php function gettype().
- **Functions for numbers**: There are also many functions you can use with numbers. Two nice ones are round() and number_format().
  - **round()**: rounds a decimal to either the nearest integer or to a specified number of digits. Round($n,2) will give 2 digits to the right of the decimal point.
  - **number_format()**: makes a number appear in the more commonly written format (adding commas where appropriate) and you can specify digits to the right of the decimal point.
- **Super Global Variables**: Super Global Variables – pre defined in php, these are always present and their values available to all your scripts
  - **$_GET**: contains any variables provided to a script through the GET method
  - **$_POST**: contains any variables provided to a script through the POST method
  - **$_COOKIE**: contains any variables provided to a script through a cookie
  - **$_FILES**: contains any variables provided to a script through file uploads
  - **$_SERVER**: contains information such as
  - **$_ENV**: contains any variables provided to a script as part of the server environment
  - **$_REQUEST**: contains any variables provided to a script via any user input mechanism
  - **$_SESSION**: contains any variables that are currently registered to a session
- **php operators**:
  - **Arithmetic Operators** (used to perform mathematical calculations)
    * `+` (Addition): Adds two values, e.g., `5 + 3` results in `8`.
    * `-` (Subtraction): Subtracts one value from another, e.g., `5 - 3` results in `2`.
    * `*` (Multiplication): Multiplies two values, e.g., `5 * 3` results in `15`.
    * `/` (Division): Divides one value by another, e.g., `15 / 3` results in `5`.

    * `%` (Modulus): Returns the remainder of division, e.g., `5 % 2` results in `1`.
    * `**` (Exponentiation): Raises a number to the power of another, e.g., `2 ** 3` results in `8`.
  - **Assignment Operators** (used to assign values to variables)
    * `=` (Basic assignment): Assigns a value to a variable, e.g., `$x = 5`.
    * `+=` (Addition assignment): Adds and assigns a value, e.g., `$x += 5` is equivalent to `$x = $x + 5`.
    * `-=` (Subtraction assignment): Subtracts and assigns a value, e.g., `$x -= 5` is equivalent to `$x = $x - 5`.
    * `*=` (Multiplication assignment): Multiplies and assigns a value, e.g., `$x *= 5` is equivalent to `$x = $x * 5`.
    * `/=` (Division assignment): Divides and assigns a value, e.g., `$x /= 5` is equivalent to `$x = $x / 5`.
    * `%=` (Modulus assignment): Takes modulus and assigns a value, e.g., `$x %= 5` is equivalent to `$x = $x % 5`.
  - **Comparison Operators** (used to compare two values)
    * `==` (Equal): Checks if values are equal, e.g., `5 == 5` results in `true`.
    * `===` (Identical): Checks if values and types are identical, e.g., `5 === "5"` results in `false`.
    * `!=` (Not equal): Checks if values are not equal, e.g., `5 != 3` results in `true`.
    * `<>` (Not equal): Alternative to `!=`.
    * `!==` (Not identical): Checks if values and types are not identical, e.g., `5 !== "5"` results in `true`.
    * `>` (Greater than): Checks if one value is greater than another, e.g., `5 > 3` results in `true`.
    * `<` (Less than): Checks if one value is less than another, e.g., `3 < 5` results in `true`.
    * `>=` (Greater than or equal to): Checks if one value is greater than or equal to another, e.g., `5 >= 5` results in `true`.
    * `<=` (Less than or equal to): Checks if one value is less than or equal to another, e.g., `3 <= 5` results in `true`.
    * `<=>` (Spaceship operator): Returns `-1` if left is less, `0` if equal, `1` if greater, e.g., `5 <=> 3` results in `1`.
  - **Logical Operators** (used to combine boolean expressions)
    * `&&` (And): Returns `true` if both operands are true, e.g., `true && false` results in `false`.
    * `||` (Or): Returns `true` if either operand is true, e.g., `true || false` results in `true`.
    * `!` (Not): Inverts the boolean value, e.g., `!true` results in `false`.
    * `xor` (Exclusive or): Returns `true` if only one operand is true, e.g., `true xor false` results in `true`.
  - **Increment/Decrement Operators** (used to increase or decrease a variable's value by 1)
    * `++` (Increment): Increases the variable's value by 1, e.g., `$x++`.
    * `--` (Decrement): Decreases the variable's value by 1, e.g., `$x--`.
  - **String Operators** (used to work with strings)
    * `.` (Concatenation): Joins two strings, e.g., `"Hello " . "world"` results in `"Hello world"`.

    * `.=` (Concatenation assignment): Appends a string to a variable, e.g., `$x .= "world"`.
  - **Array Operators** (used to work with arrays)
    * `+` (Union): Combines two arrays, e.g., `$a + $b` merges arrays $a and $b.
    * `==` (Equality): Checks if arrays have the same key-value pairs, e.g., `$a == $b`.
    * `===` (Identity): Checks if arrays have identical key-value pairs in the same order and of the same types.
    * `!=` (Inequality): Checks if arrays do not have the same key-value pairs.
    * `!==` (Non-identity): Checks if arrays are not identical in either key-value pairs, order, or types.
  - **Type Operators** (used to check or specify types)
    * `instanceof`: Checks if an object is an instance of a specific class, e.g., `$object instanceof ClassName`.
  - **Error Control Operator** (used to suppress errors)
    * `@` (Error suppression): Suppresses error messages for an expression, e.g., `@file_get_contents("nonexistentfile.txt")`.

- **Comments**: Like most computer languages, php allows you to add explanations to the code in the form of comments These comments are ignored by the php parser

  Comments should be added whenever necessary to explain code that is hard to follow

  To insert a comment in a single line of php code, you preface the comment with either a pound symbol (#) or two forward slashes (//)

- **Define()**: use php's builtin define() function

```
define("YOUR_CONSTANT_NAME", value)
```

  You can set your constant to a number, a string, or a boolean.

  By convention, use all caps for name of a constant.

  You don't use a $ when accessing a constant.

- **If statements**: If statements in php are the same as c++

```
if (expression) {
    ...
} else if (expression) {
    ...
} else {
    ...
}
```

- **Switch-Case**: Switch statements are also the same as c++

```
switch (expression) {
    case 1:
        ...
        break;
    case 2:
        ...
        break;
    default: // If no break was encountered
        ...
        break;
}
```

- **For Loop**: Same as c++

```
for (init expression; test expression; modification
↪  expression) {
    //code to execute
}
```

Note: zero, an undefined variable or an empty string will all evaluate to false, all others will evaluate to true.

- **False evaluations**: In PHP, several values evaluate to false when used in a boolean context.
  - **Boolean:** false itself.
  - **Integer:** 0 (zero).
  - **Float:** 0.0 (zero as a floating-point number).
  - **String:** An empty string ”” and the string ”0”.
  - **Array:** An empty array [].
  - **NULL:** The NULL type.
  - **Objects without methods or properties:** Objects that do not have any methods or properties defined in them may also evaluate to false when checked with empty().
- **While and do while**: Same as c++

```
while (expression) {
    ...
}

do {
    ...
} while(expression);
```

- **Arrays**: There are two ways to define an array in php.

```php
$colors = array("red","green","blue");

$colors[0] = "red";
$colors[1] = "green";
$colors[2] = "blue";
```

These are both numerically indexed arrays.

- **Associative arrays**: You can also have associative arrays which have named keys.

```php
$character = array(
    "name" => "Monk",
    "occupation" => "detective"
);
```

You access an element of an associative array by using the key name rather than a number.

- **Array functions**: There are approximately 60 array functions built into php. Some important ones are
  - count() and sizeof() return the number of elements in the array.
  - foreach() steps through an array
  - each() and list() usually appear together in the context of stepping through an array and returning keys and values
  - reset() rewinds the pointer to the beginning of the array
  - array_push() adds elements at the end of an existing array
  - array_pop() removes and returns the last element in an existing array
  - array_merge() combines two or more existing arrays
  - shuffle() randomizes the elements of a given

- **Including Files**: When developing more than a single home page for the Internet, you probably want the pages to have a common look and feel

  To make this possible, php has provided a method called include files

  These files let you incorporate common artwork, contact information, and menu and link options into your Web pages with a minimum of code

```php
// vars.php
<?php
    $color = 'green';
    $fruit = 'apple';
?>

// test.php
<?php
    echo "A $color $fruit"; // Output - A
    include 'vars.php';
    echo "A $color $fruit"; // Output - A green apple
?>
```

- **Try except**: Same as c++

```
try {
    ...
}  catch (exceptionname e) {
    ...
}
```

- **Creating objects with the new keyword**: In PHP, the new keyword is used to create an instance of a class, which is known as an object. When you use new, PHP allocates memory for the object and calls its constructor method (if defined) to initialize it

```php
$object = new ClassName();
```

- **Scope resolution operator (::)**: In PHP, the :: operator is called the Scope Resolution Operator, also known as the Paamayim Nekudotayim (Hebrew for "double colon"). It is used to access static, constant, or overridden properties and methods of a class, without needing to instantiate the class.

- **arrow operator (->)**: In PHP, the -> operator is used to access properties and methods of an object instance. It allows you to call non-static methods and access non-static properties on an instantiated object.

- **print_r**: The print_r function in PHP is used for outputting human-readable information about a variable. It's particularly useful for displaying the contents of arrays and objects. Unlike echo or print, which are mainly used to display strings, print_r provides a structured format for complex data types, making it easier to visualize the nested structure of an array or object.

```php
print_r(mixed $expression, bool $return = false): mixed
```

```php
$array = array("name" => "John", "age" => 30, "city" => "New
↪  York");
print_r($array);
```

- **foreach()**:

```
$a = array(1 => 1 ,2,3);
foreach($a as $item) {
    echo $item;

}

foreach($a as $key => $value) {
    echo "key: $key    value: $value";
    echo "<br/>";
}
```

- **Functions**: Functions in php are made with the *function* keyword

```
function functionName($param1, $param2) {
    // Code to execute
    return $result; // Optional
}
```

- **Anonymous functions**: Functions with no name, often used as variables or passed as arguments to other functions.

```
$sayHello = function($name) {
    return "Hello, $name!";
};

echo $sayHello("Bob"); // Outputs: Hello, Bob!
```

- **Arrow Functions**: Shorter syntax for one-liner anonymous functions.

```
$multiply = fn($a, $b) => $a * $b;
echo $multiply(2, 3); // Outputs: 6
```

- **Optional Parameters:** Parameters with default values, which are used if no argument is provided.

```
function greet($name = "Guest") {
    return "Hello, $name!";
}
echo greet(); // Outputs: Hello, Guest!
```

- **Passing by reference**: By default, arguments are passed by value, meaning changes within the function don't affect the original variable. You can pass by reference using &.

```php
function addOne(&$number) {
    $number += 1;
}
$num = 5;
addOne($num);
echo $num; // Outputs: 6
```

3.9 SQL via PHP: PDO

- **Why SQL via PHP?**: Can provide an interface for the user that does not require them to worry about database design specifics.

  Although there are other ways to provide this interface, the web-based interface is very common, and PHP is a common and relatively easy way of making it work.

- **Application Programming Interface (API)**: In order for our PHP application to interface with our DBMS, we will need to use an appropriate API (Application Programming Interface) An API is the set of function calls and other resources that are provided to allow you to interface with a given application via your program code.

- **Which API?**: Even for a given DBMS, there can be many API's present. PHP has been around for a while, so many things have evolved and died out. Many of the API's still work. Some work but are considered deprecated, and others are no longer supported at all. In this class, we will be working with the PDO library.

- **PDO Library**: The PHP Data Objects (PDO) library is an object oriented API to connect PHP to SQL servers. It allows you to use a common interface to interact with many different DBMS programs.

  It supports most of the popular relational DBMSes, including MySQL, Postgresql, and SQLite, in a fairly transparent way, so it is more portable than using the other, DBMS specific API's. **Note:** Because PDO is object oriented, it requires at least version 5 of PHP. If you need to use a lower version, you'll need to look into the other API's available.

- **PHP Data Objects**: The PDO library is object oriented. That means that our interactions with the database will be done using objects and their members. There are three basic objects that we will be concerned with:
  - **PDO**: this object handles the connection to the database
  - **PDOStatement**: this object handles prepared statements, and is used to work with result sets
  - **PDOException**: this object is used to store information on errors that have occurred

- **Specifying a Database with a DSN**: Although, once properly initialized, PDO should function the same for any DBMS, you need to properly initialize it by telling it which type of server you are connecting to. To do this, you need to make a DSN string.

DBMS	DSN Format
MySQL	mysql:host=HOSTNAME;dbname=DBNAME
Postgresql	pgsql:host=HOSTNAME;dbname=DBNAME
SQLite 3	sqlite:PATHTODB
SQLite 2	sqlite2:PATHTODB

  MariaDB will use the MySQL interface.

- **Initializing PDO**: Once you've chosen the DBMS you'll be using and you've chosen an appropriate DSN string, you can use that DSN to construct an instance of the PDO class. This object represents a connection to the database specified by the DSN

```
try { // if something goes wrong, an exception is thrown
    $dsn = "mysql:host=courses;dbname=z123456";
    $pdo = new PDO($dsn, $username, $password);
}
catch(PDOexception $e) { // handle that exception
    echo "Connection to database failed: " . $e->getMessage();
}
```

- **Using PDO to Talk to DBMS**: The PDO library provides three basic ways of running queries for a database, once connected:
  - the PDO::exec() function is used to execute an SQL query that does not return a result (INSERT, UPDATE, etc.)
  - the PDO::query() function is used to execute an SQL query that will return a result (SELECT)
  - the PDO::prepare() function should be used when constructing a query from user input.
- **Using PDO::exec()**: PDO::exec() is used to run a query that does not return any results. Instead of returning a result set, it will tell you how many rows were affected by your query.

```
// Three examples follow.
$n = $pdo->exec("INSERT INTO Students (FName) VALUES ('Victor');");
$n = $pdo->exec("UPDATE Students SET LName='Husky' WHERE FName='Victor';");
$n = $pdo->exec("DELETE FROM OldJunk;");
```

- **Using PDO::query()**: PDO::query() is used to run a query that does return a result. The result set is returned as a PDOStatement object.

```

$sql = "SELECT phone FROM Customer;";


$result = $pdo->query($sql);
```

- **Using PDO::prepare()**: The third option is to use the PDO::prepare() command. This is useful for situations where the same query is run multiple times in the same script, and can also help you to avoid some SQL Injection issues. Once prepare() succeeds, you run the query with the execute() method of the PDOStatement returned by prepare()

  You can use a colon before a value name in your query to denote where the execute statement will insert the value of the given name:

```php
<?php
    # Notice that we use :color below in our SQL template
    $sql = 'SELECT name, color, calories
    FROM fruit
    WHERE calories < :calories AND color = :color';

    $prepared = $pdo->prepare($sql, array(PDO::ATTR_CURSOR => PDO::CURSOR_FWDONLY));
    # The value associated with the :color key will be used when executing

    $success = $prepared->execute(array(':calories' => 150, ':color' => 'red'));
    # if(success == true)thenprepared will be ready to be used as a result set
    # with fetch() or fetchAll() -- just like the object returned by query()
?>
```

It is also possible to use a ? in your query as a positional parameter.

```php
<?php
    # Execute a prepared statement by passing an array of values
    $prepared = $pdo->prepare('SELECT name, color, calories
                               FROM fruit
                               WHERE calories < ? AND color = ?');

    # Here we execute the query twice with different parameters.
    # The ?'s will be replaced with the values in the array specified,
    # in the order they are specified.
    $prepared->execute(array(150, 'red'));
    $red = $prepared->fetchAll();
    $prepared->execute(array(175, 'yellow'));
    $yellow = $prepared->fetchAll();
?>
```

- **Named placeholders**: In PDO (PHP Data Objects), named placeholders are a way to represent dynamic values in SQL statements using identifiable names instead of anonymous ? placeholders. Named placeholders make SQL statements easier to read and maintain, especially when there are multiple variables or parameters.

  A named placeholder is written as :name in an SQL query, where name is an identifier you choose. When you prepare the SQL statement, you can bind actual values to these placeholders by their names, allowing you to execute the query with specific data values.

- **anonymous placeholders**: In PDO (PHP Data Objects), anonymous placeholders (also called positional or unnamed placeholders) are represented by question marks ? in SQL statements. These placeholders are used to represent values that will be dynamically bound to the SQL query at execution time.

  When using anonymous placeholders, each placeholder corresponds to a specific value based on its position in the SQL query. When you prepare and execute the query, you provide an array of values that will replace each ? in the order they appear.

- **Execute**: In PHP's PDO (PHP Data Objects) extension, the execute() method is used to run a prepared SQL statement with specific values for placeholders. It's part of the PDOStatement class and is essential for safely executing queries that include dynamic data, protecting against SQL injection attacks.

  After preparing a statement with PDO::prepare(), you call execute() to supply any values for placeholders (either named or anonymous) and run the query.

- **Dealing with result sets**: Once you have a result set (stored in a PDOStatement from query() you can use its PDOStatement::fetch() or PDOStatement::fetchAll() methods to get the data returned.

  If you would like to work on one row at a time, as if using the mysql_fetch_array() function from the original MySQL API, use fetch().

  To grab all of the rows at the same time, use fetchAll().

```php
<?php
    # FETCH_BOTH means that you will get both position
    indices and the column names
    # as keys in the array returned
    $row = $result->fetch(PDO::FETCH_BOTH);

    # this returns all of the rows at once in a
    two-dimensional array
    $allrows = $result->fetchAll();
?>
```

- **Handling Errors**: As of the time of this writing, there are three modes PDO can use to handle errors.
  - **PDO::ERRMODE_SILENT**: the default mode. PDO will set the error code for you to inspect using the errorCode and errorInfo methods on your PDO and PDOStatement objects
  - **PDO::ERRMODE_WARNING**: in addition to setting the error code, emits a traditional E_WARNING message. Use for debugging/testing when you just want to see what problems occurred without interrupting the flow of the application.
  - **PDO::ERRMODE_EXCEPTION**: in addition to setting the error code, also throw PDOException when an error occurs.

  You can set which one you'd like PDO to use with the setAttribute method of the PDO object you're using, as below:

```
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_SILENT);
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

- **Members of the PDO object**

```
PDO {
    // constructor has a bunch of optional parameters, check reference if needed
    public __construct ( string $dsn [, string $username [, string $password [,
    array $options ]]] )
    public bool beginTransaction ( void )
    public bool commit ( void )
    public mixed errorCode ( void )
    public array errorInfo ( void )
    public int exec ( string $statement )
    public mixed getAttribute ( int $attribute )
    public static array getAvailableDrivers ( void )
    public bool inTransaction ( void )
    public string lastInsertId ([ string $name = NULL ] )
    public PDOStatement prepare ( string $statement [, array $driver_options ] )
    public PDOStatement query ( string $statement )
    public string quote ( string $string [, int $parameter_type = PDO::PARAM_STR ] )
    public bool rollBack ( void )
    public bool setAttribute ( int $attribute , mixed $value )
}
```

- **Members of the PDOStatement object**:

```
PDOStatement implements Traversable {
    /* Properties */
    readonly string $queryString;
    /* Methods */
    public bool bindColumn ( mixed $column , mixed &$param [, int $type [, int $maxlen [, mixed $driverdata ]]] )
    public bool bindParam ( mixed $parameter , mixed &$variable [,
    int $data_type = PDO::PARAM_STR [, int $length [, mixed $driver_options ]]] )
    public bool bindValue ( mixed $parameter , mixed $value [, int $data_type = PDO::PARAM_STR ] )
    public bool closeCursor ( void )
    public int columnCount ( void )
    public void debugDumpParams ( void )
    public string errorCode ( void )
    public array errorInfo ( void )
    public bool execute ([ array $input_parameters ] )
    public mixed fetch ([ int $fetch_style [, int $cursor_orientation = PDO::FETCH_ORI_NEXT [,
    int $cursor_offset = 0 ]]] )
    public array fetchAll ([ int $fetch_style [, mixed $fetch_argument [, array $ctor_args = array() ]]] )
    public mixed fetchColumn ([ int $column_number = 0 ] )
    public mixed fetchObject ([ string $class_name = "stdClass" [, array $ctor_args ]] )
    public mixed getAttribute ( int $attribute )
    public array getColumnMeta ( int $column )
    public bool nextRowset ( void )
    public int rowCount ( void )
    public bool setAttribute ( int $attribute , mixed $value )
    public bool setFetchMode ( int $mode )
}
```

- **Members of the PDOException object**

```
PDOException extends RuntimeException {
    /* Properties */
    public array $errorInfo ;
    protected string $code ;
    /* Inherited properties */
    protected string $message ;
    protected int $code ;
    protected string $file ;
    protected int $line ;
    /* Inherited methods */
    final public string Exception::getMessage ( void )
    final public Throwable Exception::getPrevious ( void )
    final public mixed Exception::getCode ( void )
    final public string Exception::getFile ( void )
    final public int Exception::getLine ( void )
    final public array Exception::getTrace ( void )
    final public string Exception::getTraceAsString ( void )
    public string Exception::__toString ( void )
    final private void Exception::__clone ( void )
}
```

3.10 Transactions and concurrency control

- **Transaction**: A logical unit of work, a unit of program execution that access and possibly updates various data items

  is initiated by user program written in a high-level data-manipulation language like SQL, COBOL, C, C++, or Java

  Transactions are delimited by statements such as *begin transaction end transaction* or *COMMIT* or *ROLLBACK*

- **Sample Transaction**:

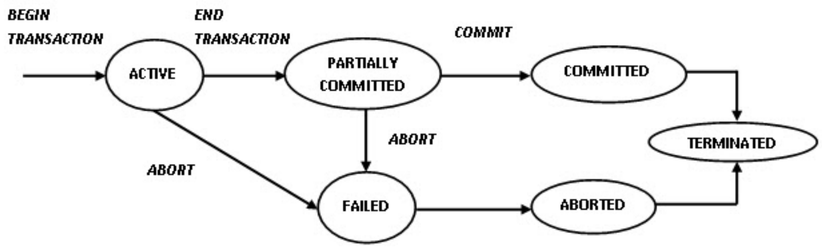
```
BEGIN TRANSACTION;
    UPDATE ACC 123 { BALANCE := BALANCE - 100 };
    IF (ANY ERROR) THEN
        GO TO UNDO;
    ENDIF;
    UPDATE ACC 456 { BALANCE := BALANCE + 100 };
    UPDATE ACC 456 { BALANCE := BALANCE + 100 };
    IF (ANY ERROR) THEN
        GO TO UNDO;
    ENDIF;
    COMMIT;
    GO TO FINISH;
UNDO:
    ROLLBACK;
FINISH:
    RETURN;
```

- **ACID properties of transactions:**
  - **Atomicity:** either all operations of the transaction are implemented properly or none are.
  - **Consistency:** execution of a transaction in isolation preserves the consistency of the database (with no other transaction executing concurrently).
  - **Isolation:** each transaction is unaware of other transactions executing concurrently in the system.
  - **Durability:** after a transaction completes successfully, the changes it has made to the database persist, even if the system fails.

- **Transaction States**: In the absence of any failures, all transactions complete successfully.

  However, there is not always an absence of any failures.

  Thus we have different "states" in which a transaction may reside.



- **Active:** the initial state and one in which the transaction stays while it is executing.
- **Partially committed:** a state after the final statement has been executed.
- **Failed:** the state after the discovery that normal execution can no longer proceed.
- **Aborted:** the state after the transaction has been rolled back and the database restored to its condition prior to the start of the transaction.
- **Committed:** the state after successful execution.

- **Committed Transaction**: A transaction is considered committed when it has performed updates that transforms the database into a new consistent state.

  Once a transaction is committed, its effects cannot be undone by a system failure.

  Only way to undo a committed transaction is to execute a compensating transaction.

  However it is not always possible to create a compensating transaction.

- **Failed & Aborted Transaction**: When a transaction cannot be completed (due to some kind of system failure), the transaction must be "rolled back".

  It also enters the aborted state where the system has two options:

  - Restart the transaction (but only if aborted due to some hardware or software error).
  - Kill the transaction which is usually done due to some internal logical error.

- **Concurrent Executions of Transactions**: Concurrent executions are good:

  - Improved throughput and resource utilization
  - Reduced waiting time

  In transaction processing multiple transactions are allowed to run concurrently.

  Updating within concurrent transactions causes several complications with consistency of the data.

- **Execution sequences**: Represent the chronological order in which instructions are executed in the system

- **Serial schedules**: consists of a sequence of instructions from various transactions where the instructions belonging to one single transaction appear together in that schedule

- **Concurrency control**: Concurrency control: the task of ensuring that any schedule that gets executed will leave the database in a consistent state

The concurrency-control component of the DBMS carries out setting up the schedules to ensure consistency during the execution of multiple transactions

- **Serializability**: a schedule that is equivalent to a serial schedule.

  In discussing serializability, only two operations are important

  - read(Q)
  - write(Q)

  We also assume that the transaction may perform an arbitrary sequence of operations on the copy of Q between the read and write operations

- **Concurrency control introduction**: Fundamental Property of a transaction is isolation

  To ensure that isolation property is preserved when several transactions are running concurrently, the system controls the interaction among the concurrent transactions

  These schemes are called concurrency control

- **Lock-Based Protocols**: One way to ensure serializability

  - Require data items be accessed in a mutually exclusive manner
  - That is when one transaction is accessing the data item, no other transaction can modify that data item
  - Common method to implement is that transaction must hold a lock on an item

- **Modes in which data item may be locked**:

  - **Shared**: If a transaction T1 has obtained a shared-mode lock on item Q, then T1 can read, but cannot write, Q
  - **Exclusive**: If a transaction T1 has obtained an exclusivemode lock on item Q, then T1 can both read or write Q

  Every transaction is required to request a lock in an appropriate mode on each data item

  Request is made to the concurrencycontrol manager

  Concurrency-control manager must grant the lock to the transaction before it can proceed.

- **Compatibility Function**: Given set of lock modes can define compatibility function

  - A & B represent arbitrary lock modes
  - Transaction T1 request a lock of mode A on item Q
  - Transaction T2 currently holds a lock of mode B
  - If T1 can be granted a lock on Q immediately in spite of the presence of the mode B lock, then mode A is compatible with mode B

  Can represent with a Lock-compatibility matrix

	shared	exclusive
shared	true	false
exclusive	false	false

When a transaction requests a lock that is incompatible, it enters a wait state until all incompatible locks have been released

Transactions cannot execute until concurrency-control manager grants the requested locks

- **Deadlock**: Locking can lead to an undesirable situation where no transaction can proceed with normal execution. This situation is called deadlock

  When this occurs

  - The system must roll back one of the transactions
  - Unlocking transactions until execution can be continued

  If locking is not used, or if a data item is unlocked as soon as possible after reading or writing, inconsistent states may occur

  On the other hand, if a data item is not unlocked before requesting a lock on some other data item deadlocks may occur

  In general, deadlocks are a necessary evil associated with locking which is necessary to avoid inconsistent states

  Deadlocks are preferable to inconsistent states

  - since they can be handled via rolling back transactions
  - inconsistent states cannot be handled by database

- **Locking Protocol**: a set of rules that each transaction must follow

  Indicates when a transaction may lock or unlock each data item

  A schedule is legal under a given locking protocol if it follows the rules

  A locking protocol ensures conflict serializability if and only if all legal schedules are conflict serializable

- **Granting of Locks**: Grant can take place if

  - a transaction requests a lock on a data item in some mode
  - and no other transaction has a lock on the same data item in a conflicting mode

  Care must be take to avoid certain situations

  Suppose transaction T2 has a shared-mode lock on a data item

  Transaction T1 requests an exclusivemode lock on the data item

Clearly, T1 has to wait for T2 to release the shared-mode lock

Meanwhile, transaction T3 may request a shared-mode lock on the same data item

The lock request is compatible with the lock granted to T2 so T3 may be granted the shared-mode lock

At this point, T2 releases the lock but T1 has to still wait for T3 to finish

But again, there are other transactions Ti, that requests a shared-mode lock

In fact it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item and T1 NEVER gets the exclusive-mode lock

T1 is then said to be starved

- **Avoiding starvation of transactions**: T1 requests a lock on a data item Q in some mode M. the lock is granted provided that
  - there is no other transaction holding a lock on Q in a mode that conflicts with M
  - there is no other transaction that is waiting for a lock on Q and that made its lock request before T1
- **Two-Phase Locking Protocol**: A protocol that ensures serializability is the two-phase locking protoco

  Each transaction issues lock and unlock requests in two phases

  1. **Growing phase**: a transaction may obtain locks but may not release any lock
  2. **Shrinking phase**: a transaction may release locks but may not obtain any new locks

  Initially a transaction is in the growing phase

  Once the transaction releases a lock, it enters shrinking phase and cannot request more locks

  The Two-phase locking protocol

  1. Ensures conflict serializability
  2. Does not ensure freedom from deadlock
- **Variations of two-phase locking protocol**:
  1. **Strict two-phase locking protocol**: requires not only that locking be two phase, but that all exclusive-mode locks taken by a transaction be held until that transaction commits
  2. **Rigorous two-phase locking protocol**: requires that all locks be held until the transaction commits
- **Refinement of basic two-phase locking protocol**: lock conversions are allowed
  - a mechanism is allowed for upgrading a shared lock to an exclusive lock
  - a mechanism is allowed for downgrading an exclusive lock to a shared lock

Strict two-phase and rigorous twophase locking (with lock conversions) are extensively used in DBMSs

A simple but widely used scheme automatically generates the appropriate lock and unlock instructions for a transaction on the basis of read and write requests

When a transaction T1 issues a read(Q), the system issues a lock-S(Q) instruction followed by the read(Q) instruction

When T1 issues a write(Q) operation, the system checks to see whether T1 already holds a shared lock on Q.

- If yes, system issues an upgrade(Q) followed by a write(Q)
- If no, system issues a lock-X(Q) followed by a write(Q)

All locks obtained by a transaction are unlocked after that transaction commits or aborts

- **Other Locking Protocols**:
  - **Graph-based protocols**
  - **Timestamp-based protocols**
  - **Validation-based protocols**: majority of transactions are read-only
  - **Multiversion schemes**: each write(Q) creates a new version of Q
- **Deadlock Handling**: Deadlock state... there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set

  Two principal methods for dealing with deadlocks

  1. deadlock prevention
  2. deadlock detection and deadlock recovery
- **Deadlock prevention**: two approaches
  1. **one approach:** ensure that no cyclic waits can occur by
     - ordering the requests for locks
     - or requiring all locks be acquired together
  2. **second approach:** impose an ordering of all data items and require that a transaction lock data items only in a sequence consistent with the ordering
- **Deadlock detection and recovery**: an algorithm that examines the state of the system is invoked periodically to see if a deadlock has occurred, if one has, then the system must recover

  To recover from a deadlock the system must

  - maintain information about the current allocation of data items to transactions as well as any outstanding data item requests
  - provide an algorithm that uses this information to determine whether the system has entered a deadlock state
  - recover from the deadlock when the detection algorithm determines that a deadlock exists.

3.11 Mariadb in C++

- **Motivation**: We have already looked at how to interact with a MariaDB Database from PHP using the PHP Data Objects (PDO) API. Now we will look at the API you can use to do the same thing in C/C++ programs.

  The MariaDB API is implemented in C, so it is provided as a set of functions to call, as opposed to a set of classes, as may be found in more object oriented programming

- **Header file**: To use any of these functions, you need to include the appropriate header file.

```
#include <mysql/mysql.h>
```

  This file provides declarations for the functions and new types that make up the MariaDB API. Later, during the linking stage, you will specify the mariadb library, where the implementations of the functions are found.

- **Some new variable types**: The MariaDB library defines some new types. Some of them are data structures that store more complicated data, which is dealt with using the library functions, others are just a different way of referring to types you already know
  - **MYSQL**: a data structure containing data about a connection to a MariaDB server
  - **MYSQL_RES**: a data structure that contains data about a result set
  - **MYSQL_ROW** a data structure that contains data about a single row from a result set
  - **my_ulonglong**: used to store large unsigned integers

- **Handling errors**: Various functions in the MariaDB API have specific return codes for when an error has occurred. In other situations, it may not be clear.

  When an error occurs, there are two functions you can use to find out what happened:

```
unsigned int mysql_errno(MYSQL *mysql);
const char *mysql_error(MYSQL *mysql);
```

  - mysql_errno - returns a numeric error code that identifies the last error that occurred
  - mysql_error - returns a human-readable error message describing the last error that occurred.

  The mysql parameter is a pointer to the connection object you would like to check the error for. We will discuss that later.

- **Library Initialization**:

```
int mysql_library_init(int argc, char **argv, char **groups)
```

This function initializes the MySQL library. It is generally only needed if you want your application to be threadsafe, but you should call it in your programs for this class anyway.

  – **argc**: argument count, use 0 unless you have a reason not to
  – **argv**: argument vector, use NULL unless you have a reason not to
  – **groups**: NULL-terminated array of strings listing which options to read, use NULL unless you have a reason not to

- **Library Deinitialization**:

```
void mysql_library_end(void)
```

This function cleans up any memory allocated during the use of the MySQL library. You should make sure to call it after you are completely finished using the library to communicate with the DBMS.

- **Initializing a Connection Object**:

```
MYSQL *mysql_init(MYSQL *mysql)
```

This function initializes (allocating memory as well, if needed) a MYSQL object, which is suitable for use with mysql_real_connect(). If mysql is NULL, then function will dynamically allocate memory for the object. If it is a pointer to an already allocated MYSQL object, it will set it up in that place. Either way, it will return a pointer to where the initialized object is. If there is an error, NULL will be returned.

  – **mysql**: a pointer to a MYSQL object to initialize, or NULL

- **Establishing a DB Connection**:

```
MYSQL *mysql_real_connect(MYSQL *mysql, const char *host,
    const char *user, const char *passwd,
    const char *db, unsigned int port,
    const char *unix_socket, unsigned long client_flag)
```

Establishes a connection to the specified MySQL database.

  – **mysql**: pointer to an existing MYSQL object, can be used to set options
  – **host**: the hostname of the server to connect to item user - the username to authenticate with
  – **passwd**: the password for the specified user
  – **db**: the name of the database to use on that MySQL server
  – **port**: set to zero unless you want to use a non-default port
  – **unix_socket**: set this to NULL
  – **client_flag**: set this to zero (can be used for special flags)

Returns a pointer to a MYSQL connection object on success, NULL on failure.

- **Closing a DB Connection**:

```
void mysql_close(MYSQL *mysql)
```

This function closes a previously opened connection. It also deallocates the connection handler pointed to by mysql if the handler was allocated automatically in either mysql_init() or mysql_connect().

  - **mysql**: a pointer to the MYSQL object for the connection to be closed

- **Running SQL Queries**: After you have initialized the API and connected to the server, you can begin to run SQL queries on it. There are a couple of functions that are designed to do this.
  - **mysql_query()**: allows you to send a query as a null-terminated string
  - **mysql_real_query()**: allows you to send a query as a string of a specified length

- **Queries using null-terminated string**:

```
int mysql_query(MYSQL *mysql, const char *stmt_str)
```

This function executes the SQL statement stored in the null-terminated string pointed to by stmt_str. Normally the string must consist of a single SQL statment without a terminating semicolon. If you explicitly enable multiple-statement execution, it can contain several statements separated by semicolons. It is recommended not to do this, normally, because of the potential for SQL injection.

  - **stmt_str**: a pointer to the beginning of a null-terminated string containing the SQL statement(s) to run

Returns zero on success. Non-zero if an error has occurred.

- **Queries using string with length**:

```
int mysql_real_query(MYSQL *mysql,
        const char *stmt_str,
        unsigned long length)
```

This function executes the SQL statement pointed to by stmt_lstr, which is interpreted to be length bytes long. The use of a length as opposed to a terminating null character is what distinguishes this from the mysql_lquery() function from before. This allows binary data, which may contain null characters as valid data, to be sent.

  - **mysql**: pointer to the MySQL connection to run the query through
  - **stmt_lstr**: pointer to the beginning of the string containing the SQL statement(s)
  - **length**: the length, in bytes, of the string, stmt_lstr

It returns zero on success, non-zero if an error has occurred.

- **Get information on a query's results**: After a query is run, there may be a result set, which you would expect from a query with, for example, a SELECT statement. There are a couple of ways to access the results of such a query.

  Other queries might not generate a result set, such as INSERT, UPDATE, or DELETE. There are some things you may want to know about them even though no data is returned

  For queries that don't generate a result set, you may want to know some or all of the following

  – If you want to know how many rows were returned in a result set, you can use mysql_lnum_lrows().
  – If you want to know how many attributes (columns) exist in a result set, you can use mysql_lfield_lcount().
  – If you want to know what value was chosen for an AUTO_lINCREMENT field, you can use mysql_linsert_lid().

- **How many rows were affected?**:

```
my_ulonglong mysql_affected_rows(MYSQL *mysql)
```

  This function can be called immediately after running a query on the server.

  – **mysql**: a pointer to the server connection object you ran the query through

  Greater than zero indicates number of rows affected or retrieved. Zero indicates no rows. Errors are indicated with a -1 return code.

- **How many rows were returned?**:

```
my_ulonglong mysql_num_rows(MYSQL_RES *result)
```

  This function will return the number of rows in the specified result set. If you want to use this before doing things with your data, you will need to use mysql_lstore_lresult() instead of mysql_luse_lresult() to retrieve the result set.

  – **result**: a pointer to the result set object to inquire about

- **How many columns in the result?**:

```
unsigned int mysql_field_count(MYSQL *mysql)
```

  This function will return the number of columns were in the result set of the most recent query on the specified connection.

  – **mysql**: a pointer to the connection object you just queried

- **AUTO_INCREMENT → What is the key of inserted row?**:

```
my_ulonglong mysql_insert_id(MYSQL *mysql)
```

This function returns the value generated for an AUTO_INCREMENT column by the previous INSERT or UPDATE statement.

- **mysql**: a pointer to the connection object we ran the query through

Will return zero unless a value has been stored in an AUTO_INCREMENT field. If multiple rows were affected, only the first one will be returned.

- **Getting info from result sets**: There are two basic functions that you can use to retrieve the data from a result set.
  - The mysql_lstore_lresult() will function similarly to the way PDOStatement::fetchAll() function did, in that it will retrieve all the results at once.
  - The mysql_luse_lresult() works more similarly to the PDOStatement::fetch() function, grabbing the results one row at a time.
  - Neither of these two will directly give you the values in the row; that can be done with the mysql_lfetch_lrow() function.
  - You should make a call to mysql_lfree_lresult() after finishing with the result sets, to free up memory used to store the results.
- **Download all rows of the result set immediately**

```
MYSQL_RES *mysql_store_result(MYSQL *mysql)
```

This function is used to retrieve the result set generated by a query. It will download the whole result set from the server immediately and store it in your program's memory. The values can then be obtained row-by-row with mysql_lfetch_lrow() or you can use mysql_lrow_lseek() to jump to specific rows.

- **mysql**: pointer to the connection object we used to run the query

Returns a pointer to a result set structure if successful, NULL on error.

- **Set up to download one row at a time**:

```
MYSQL_RES *mysql_use_result(MYSQL *mysql)
```

This function sets up a result set that will fetch its rows one at a time from the server. Individual rows are obtained with calls to mysql_lfetch_lrow(). This is more memory efficient than using mysql_lstore_lresult() would be.

- **mysql**: a pointer to the connection object that ran the query

Returns a pointer to the result set structure on success. NULL is returned on failure.

- **Get the row data**:

```
MYSQL_ROW mysql_fetch_row(MYSQL_RES *result)
```

This will fetch the next row in the result set as a MYSQL_lROW structure, if there is one to fetch. It will start at the beginning and advance by one row each time it is called. If you call this after you've fetched the last row, it will return NULL. It will also return NULL if an error has occurred.

– **result**: a pointer to the result set object to fetch rows from

If you have any binary data in any of your fields, you will need to use mysql_lfetch_llengths() to get the lengths, as they may contain the null character as part of their valid data. Otherwise, you can treat a MYSQL_lROW as an array of null-terminated strings. The fields can be addressed as the elements of the array, in the order they appear in the result set, starting from element 0.

- **Get byte-lengths for the fields in a row**:

```
unsigned long *mysql_fetch_lengths(MYSQL_RES *result)
```

This function returns an array of integers that contains the lengths of each of the fields in a row returned by a previous call to mysql_lfetch_lrow(). The integer in a given element of the array returned is the length of the corresponding element in the MYSQL_lROW

– **result**: a pointer to the result set you just got a row from

Returns NULL on error. The most common error will be that you haven't fetched a row, or the last fetch failed.

- **Compiling without Linking**: To compile without linking, you can specify the -c flag to gcc or g++. This will take your source code, program.cc or program.c in this example, and make an object file from the code within it.

```
g++ -c -I/usr/include/mariadb program.cc
gcc -c -I/usr/include/mariadb program.c
```

Either of these statements will yield a new file called program.o, which is the object file compiled from the original source.

- **Compilation Errors**: The compilation stage is concerned with declarations. If you have an error that says something is undeclared, that failure is happening in the compilation stage.

If you have an error during compilation, it is usually one of the following types

- You forgot to include a header file that contains necessary declarations
- You made a syntax error in your code, which the compiler's error message should help you correct
- You made a typo somewhere (misspelled identifiers, etc.)

- **Linking Alone**: After your object files have been created through compilation, you can perform the linking stage with one of the following commands.

```
g++ -o program program.o -lmariadb
gcc -o program program.o -lmariadb
```

  - The -o flag say to name the program whatever the next word is. In this case, your program.o would be linked to make an executable file named program. If your project had multiple source code files, you'd include all of the object files' names, separated by spaces, where program.o is now.
  - the -l flag is used to specify the name of the library to link in.

- **Linking Errors**: The linking stage is concerned with finding the compiled code in object files/libraries. If you see an error about an "undefined reference", then the failure is happening during the linking stage. If an error occurs during the linking phase, it is usually one of the following

  - You failed to list one of the object files that contains the implementations of your functions
  - You failed to tell the linker to include a library that is needed
  - The linker path does not include the directory your library is located in.
  - Your header file has a different declaration than its implementation uses.

- **Compilation and Linking Together**: It is possible to perform compilation and linking with the same command. This may be easier to type, but doing it separately allows you to skip recompiling files that haven't changed. The following commands are an example of how to do both stages together.

```
g++ -o program -I/usr/include/mariadb -lmariadb program.cc
gcc -o program -I/usr/include/mariadb -lmariadb program.c
```

Assembler

4.1 Number systems and computer storage

- **The Decimal number system**: In the decimal system, any natural number m can represented by use of the ten symbols $0, 1, 2, 3, ...9$. These are the decimal digits, m can be represented as

$$d_n d_{n-1} d_{n-2} ... d_1 d_0$$

  Where $m \geqslant 0$. This same number m can be represented as

$$d_n \times 10^n + d_{n-1} \times 10^{n-1} + d_{n-2} \times 10^{n-2} + ... + d_1 \times 10^1 + d_0 \times 10^0$$

  For example, the natural number 123 can be represented as

$$1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

  The decimal system is also called the base ten system since ten digits are utiltized in the number representations.

  There is, however, nothing sacred about the base ten since the notion of a positional number system can be easily generalized to any given base b where b is a natural number greater than or equal to tw

- **Binary**: The binary number system is a base two number system, where any natural number m can be represented with the digits 0 and 1. Observe since

$$123 = 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

  The number 123 would be represented in the binary system as

$$1111011$$

- **Hexadecimal**: The hexadecimal number system is a base 16 number system, which uses $0, 1, 2, ...9, A, B, C, ...F$, where $A, B, C, ..., F$ represent $10, 11, 12, ..., 15$. Observe

$$123 = 7 \times 16^1 + 11 \times 16^0$$

  Thus 123 has the decimal representation $7B$

- **Numbers 1-30 in each system**: The following table gives the representations of the numbers zero through thirty-two in each of these three number systems.

Decimal	Hexadecimal	Binary	Decimal	Hexadecimal	Binary
0	0	0	17	11	10001
1	1	1	18	12	10010
2	2	10	19	13	10011
3	3	11	20	14	10100
4	4	100	21	15	10101
5	5	101	22	16	10110
6	6	110	23	17	10111
7	7	111	24	18	11000
8	8	1000	25	19	11001
9	9	1001	26	1A	11010
10	A	1010	27	1B	11011
11	B	1011	28	1C	11100
12	C	1100	29	1D	11101
13	D	1101	30	1E	11110
14	E	1110	31	1F	11111
15	F	1111	32	20	100000
16	10	10000			

- **Base notation**: For clarity, when it is not clear which system a given number is represented in, a subscript with the base of the system will be used. For example,

$$(123)_{10} = 7B_{16} = 1111011_2$$

- **Binary and hex to decimal**: Let $a_n a_{n-1} ... a_2 a_1 a_0$ be the representation of a number m in base b. Then, the decimal representation of m is given by the sum

$$d_n b_n + d_{n-1} b_{n-1} + ... + d_2 b_2 + d_1 b_1 + d_0 b_0$$

  Where each d_i is the decimal equivalent of the corresponding a_i

  For example, consider the binary number 1011_2, then

$$1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 = 11$$

  Next, consider the hex number $A61$. Observe

$$\begin{aligned} 1 \cdot 16^0 + 6 \cdot 16^1 + A \cdot 16^2 &= 1 \cdot 16^0 + 6 \cdot 16^1 + 10 \cdot 16^2 \\ &= 2657 \end{aligned}$$

- **Decimal to binary or hex**: Obtain the representation of a number n in a given base b from the representation of n in the decimal system by using the following steps

  1. Divide n by b, giving a quotient q and remainder r
  2. Write the representation of r in the base b as the rightmost digit or as the digit to the immediate left of the one last written.
  3. If q is zero, stop. Otherwise set n equal to q and go to step 1

  For example, consider $123_{10} \to h_{16}$, where h is the hexidecimal representation. We follow the above procedure.

$$\begin{aligned} 123/16 &= 7 + 11 \ : \ B_{16} \\ 7/16 &= 0 + 7 \ : \ 7_{16} \end{aligned}$$

  Since we hit a $q = 0$, we stop. The hexidecimal representation is then $7B_{16}$. Next, consider $123 \to b_2$

$$\begin{aligned} 123/2 &= 61 + 1 \ : \ 1_2 \\ 61/2 &= 30 + 1 \ : \ 1_2 \\ 30/2 &= 15 + 0 \ : \ 0_2 \\ 15/2 &= 7 + 1 \ : \ 1_2 \\ 7/2 &= 3 + 1 \ : \ 1_2 \\ 3/2 &= 1 + 1 \ : \ 1_2 \\ 1/2 &= 0 + 1 \ : \ 1_2 \end{aligned}$$

  Thus, $123_{10} = 111011_2$

- **Conversion between binary and hex**: Because 16 digits are required in the hexidecimal system and $2^4 = 16$, a very simple algorithm exists for converting binary representations to hexidecimal representations, and vice versa.

The algorithm may be stated as follows

1. Starting at the right of a binary representation n, separate the digits into groups of four. If there are fewer than four digits in the last (leftmost) group, add as many zeros as may be necessary to the left of the leftmost digit to fill out the group. For example, if $n = 101101$, the digits should be separated into two groups depicted as follows: 10 1101. Since the leftmost group does not contain four digits, two leading zeros are added to give 0010 1101.
2. Convert each group of four binary digits to a hexadecimal digit. The result is the hexadecimal representation of n.

Consider $n = 101101$. Splitting into groups of four, we get the two groups

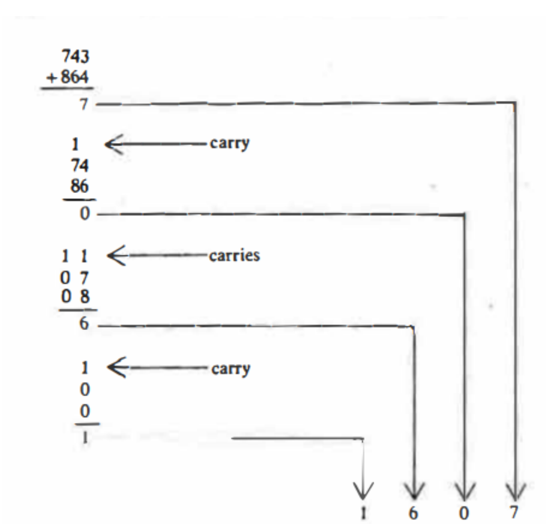
$$0010 \quad 1101$$

Since $0010 = 2_{10}$, and $1101 = 13_{10} = D_{16}$, we get $0010_2 = 2D_{16}$

- **Addition of binary and hexidecimal numbers**: The algorithm for the addition of unsigned integer numbers is as follows

  1. Write the two addends one above the other with the rightmost digits of these numbers aligned.
  2. Add the two rightmost digits; if a 1 appears above these digits, indicating a carry, add 1 to the result. Write the integer portion of this result to the immediate left of the last recorded digit in the sum; If a carry is part of the result, write a 1 above the next higher order pair of digits. (If one or both of the digits do not exist, assume a value of 0 for the missing digits.)
  3. Delete the rightmost digits of the two addends. If the digits of the addends are exhausted, stop; otherwise, go to Step 2.

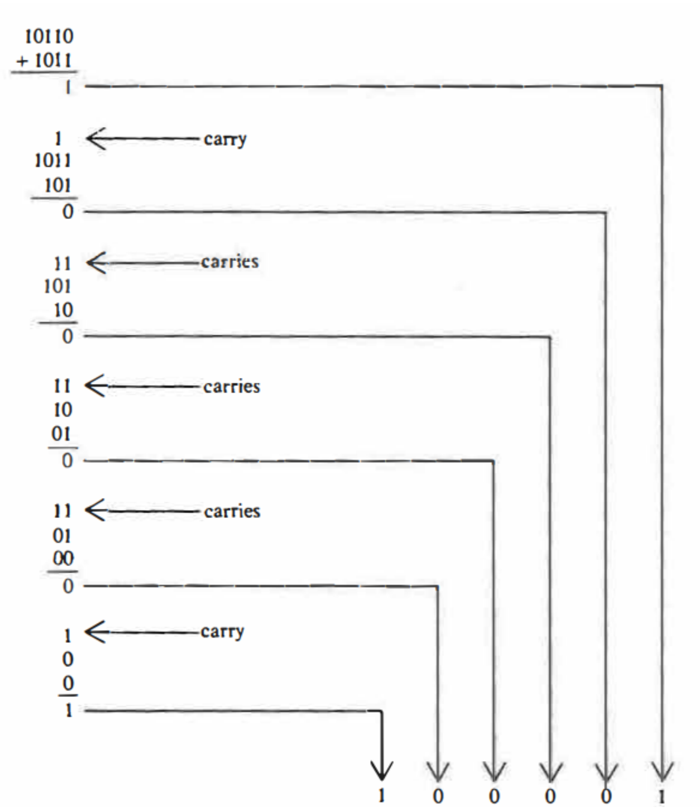
We start we a decimal system example. Suppose 743_{10} and 864_{10} are to be added. Then,



Collecting the results yields 1607_{10}. The carry table for binary is simple, since there are only two digits involved.

+	0	1
0	0	1
1	1	$0 + c$

The result of using this table and the algorithm to find the sum of 10110 and 1011 is shown below

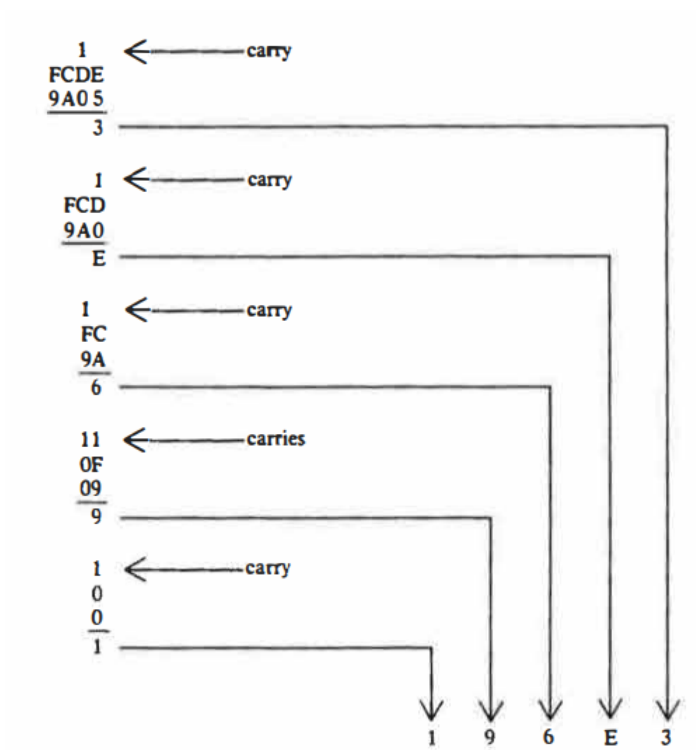


Thus, $10110_2 + 1011_2 = 100001_2$

The carry table for hexidecimal addition is a bit more complex.

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0+c
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0+c	1+c
3	3	4	5	6	7	8	9	A	B	C	D	E	F	0+c	1+c	2+c
4	4	5	6	7	8	9	A	B	C	D	E	F	0+c	1+c	2+c	3+c
5	5	6	7	8	9	A	B	C	D	E	F	0+c	1+c	2+c	3+c	4+c
6	6	7	8	9	A	B	C	D	E	F	0+c	1+c	2+c	3+c	4+c	5+c
7	7	8	9	A	B	C	D	E	F	0+c	1+c	2+c	3+c	4+c	5+c	6+c
8	8	9	A	B	C	D	E	F	0+c	1+c	2+c	3+c	4+c	5+c	6+c	7+c
9	9	A	B	C	D	E	F	0+c	1+c	2+c	3+c	4+c	5+c	6+c	7+c	8+c
A	A	B	C	D	E	F	0+c	1+c	2+c	3+c	4+c	5+c	6+c	7+c	8+c	9+c
B	B	C	D	E	F	0+c	1+c	2+c	3+c	4+c	5+c	6+c	7+c	8+c	9+c	A+c
C	C	D	E	F	0+c	1+c	2+c	3+c	4+c	5+c	6+c	7+c	8+c	9+c	A+c	B+c
D	D	E	F	0+c	1+c	2+c	3+c	4+c	5+c	6+c	7+c	8+c	9+c	A+c	B+c	C+c
E	E	F	0+c	1+c	2+c	3+c	4+c	5+c	6+c	7+c	8+c	9+c	A+c	B+c	C+c	D+c
F	F	0+c	1+c	2+c	3+c	4+c	5+c	6+c	7+c	8+c	9+c	A+c	B+c	C+c	D+c	E+c

Let's add $FCDE$ and $9A05$



Thus, the result is $196E3_{16}$

- **Subtraction of binary and hexidecimal numbers**: In the subtraction algorithm it is assumed that for $a - b$, $a \geqslant b$.

  1. Write the minuend above the subtrahend with the rightmost digits of these numbers aligned.

2. (a) If the rightmost digit in the minuend is greater than or equal to the corresponding digit in the subtrahend, subtract the digit in the subtrahend from the corresponding digit in the minuend and write the result to the immediate left of the last recorded digit in the difference; otherwise
   (b) If the rightmost digit d in the minuend is less than the corresponding digit in the subtrahend, replace d by $d+c$, decrease the next-higher-order nonzero digit in the minuend by 1, replace any intervening zero digits in the minuend by the digit corresponding in value to the base minus 1. Then subtract the rightmost digit in the subtrahend from $d + c$ and write the result to the immediate left of the last recorded digit in the difference.
3. Delete the rightmost digits in the minuend and subtrahend. If the digits of these two numbers are exhausted, stop; otherwise, go to Step 2.