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:**

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

  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.

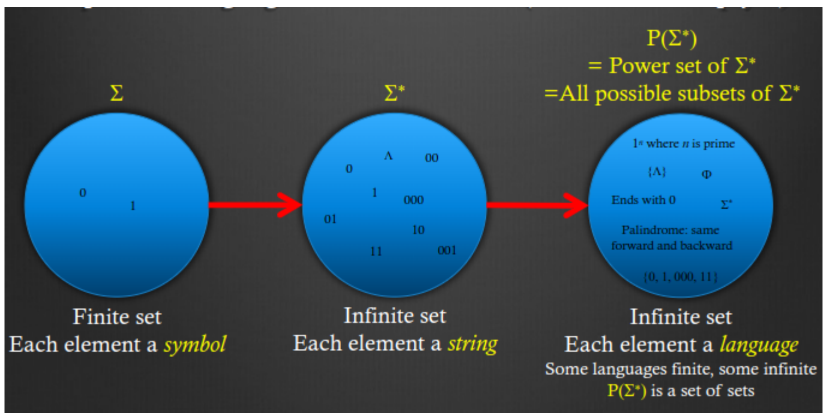
In other words, given a set S, then its power set $\mathcal{P}(S)$ is a set of sets

– **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.

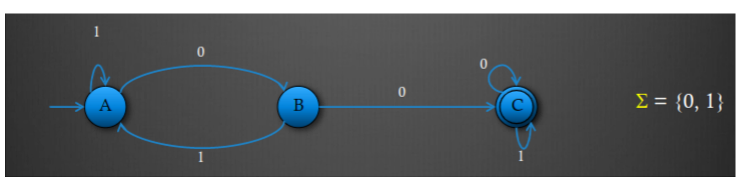
1.2 Regular languages

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.

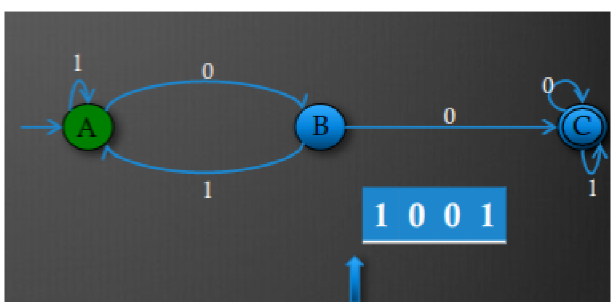
- **How to use an FA**: The purpose of a FA is to define a language over its alphabet Σ. The FA provides the means by which to test a candidate string from Σ and determine whether or not the string is in the language. It does this by "writing" the candidate string on an fictitious input tape and proceeding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

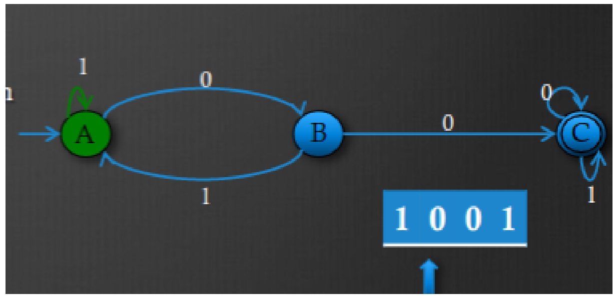
- **Using the previous FA**: Let's now try to use our FA to test whether or not the string 1001 is in the language

  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

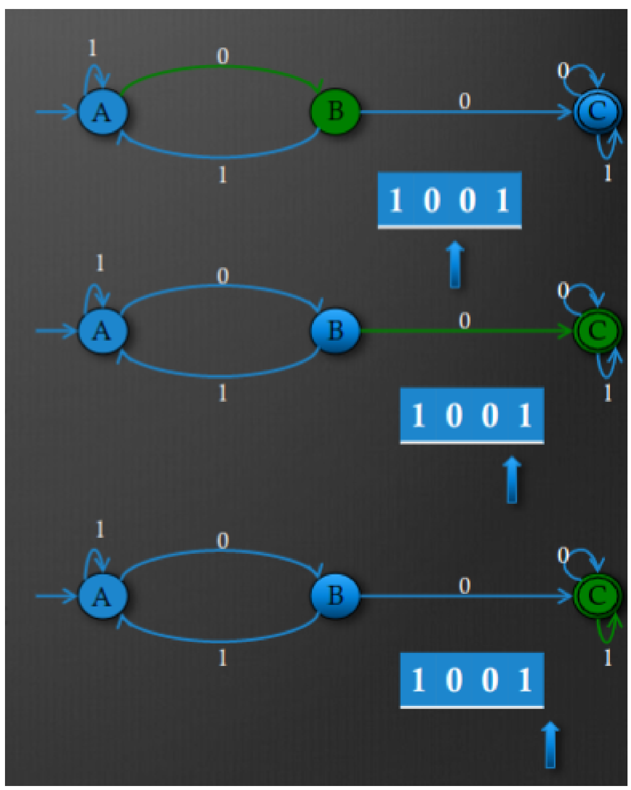


In this case, we started in state A, read symbol 1, and followed the edge labeled 1 from A which brought us back to A

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

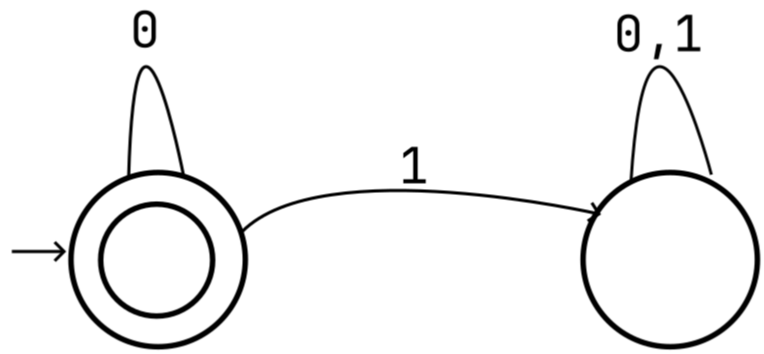
Once the tape head reaches the end of the tape we simply look to see whether or not the FA ended in an accepting state.

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



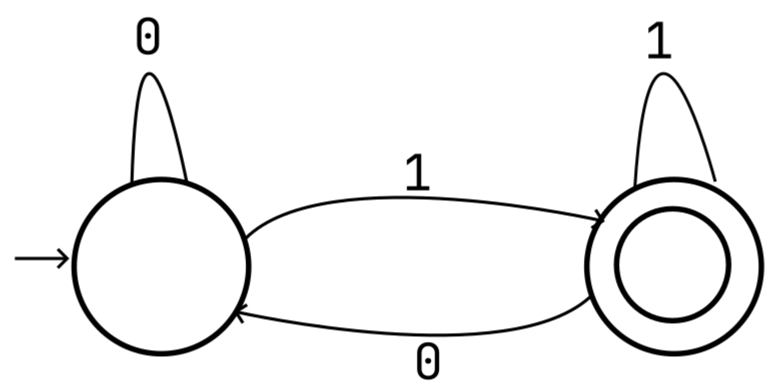
We deduce that the language has only strings with two consecutive zeroes somewhere.

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

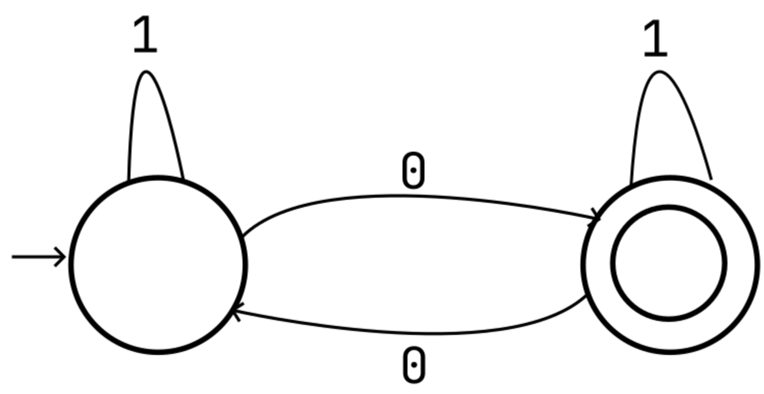


This one is pretty simple. If we have a zero, stay in the accepting state, if we see a one, toss it to the other non-accepting state, its not coming back.

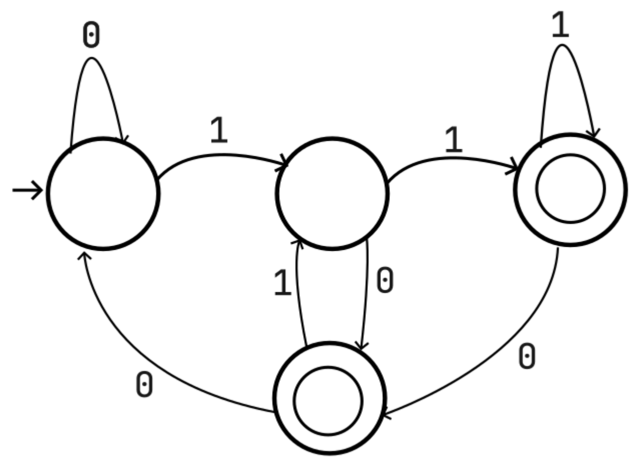
- **FA Example Three: The set of all strings that end in one ($\Sigma = \{0, 1\}$):**



- **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:

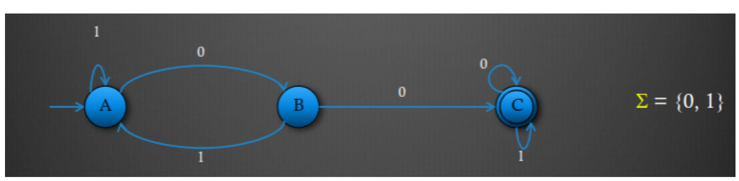
- **Finite State:** A finite automaton has a finite number of states. This means it can only ”remember” a limited amount of information about the input it has processed. Once a finite automaton transitions to a new state, it forgets all previous information except for the current state.
- **No Stack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

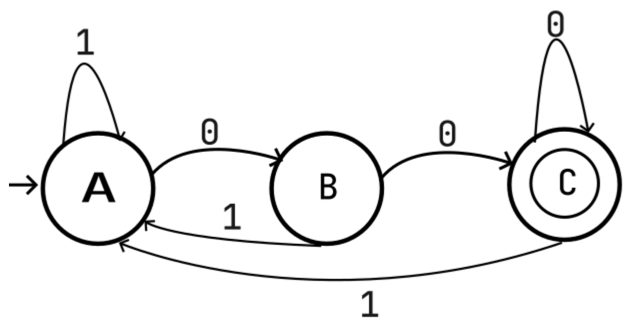
- δ is the *transition function* $\delta : Q \times \Sigma \to Q$

δ	0	1
A	B	C
B	C	A
C	C	C

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

1.2.2 Finite Automata: More examples

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



With:

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

δ	0	1
A	B	A
B	C	A
C	C	A

1.2.3 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^*.

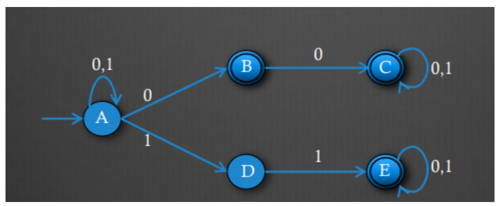
  Recursive step: The second rule allows you to take any string x already in L^* and concatenate it with a string $y \in L$ to produce a new string $xy \in L^*$.

  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.
- **Kleene's theorem**: There is an FA for a language if and only if there is an RE for the language

1.2.4 nondeterministic Finite automata (NFA)

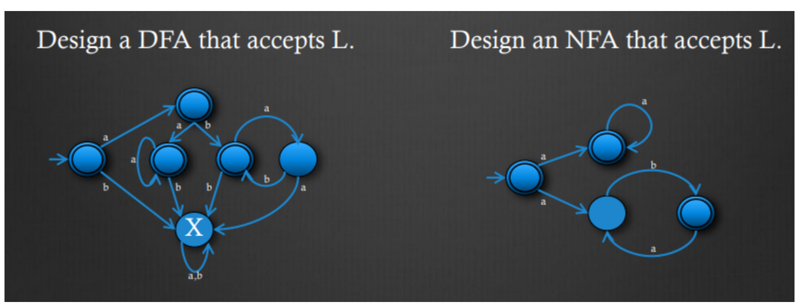
- **NFA definition**:
  - If an automaton gets to a state where there is more than one possible transition corresponding to the symbol read from the tape, the automaton may choose any of those paths. (nondeterminism) We say it **branches**
  - if an automaton gets to a state where there is no transition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



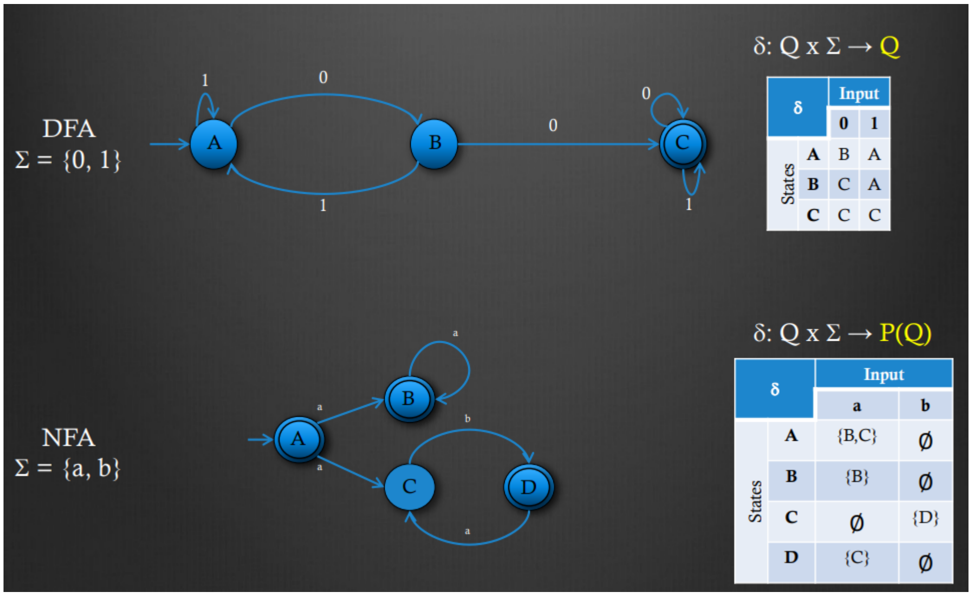
- **DFA or NFA?**: Consider the language L over $\Sigma = \{a, b\}$ which is defined by

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



- **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**:

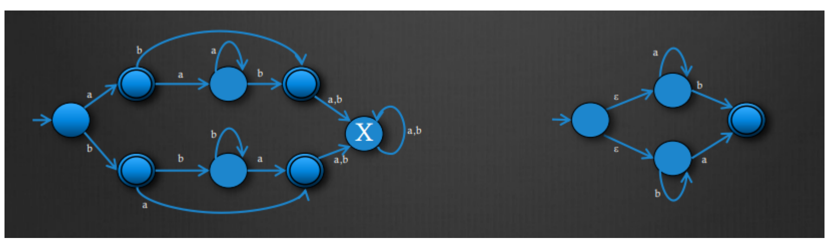


- **NFA with ϵ-transitions**: ϵ-transitions allow the automaton to change state without consuming an input symbol

  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

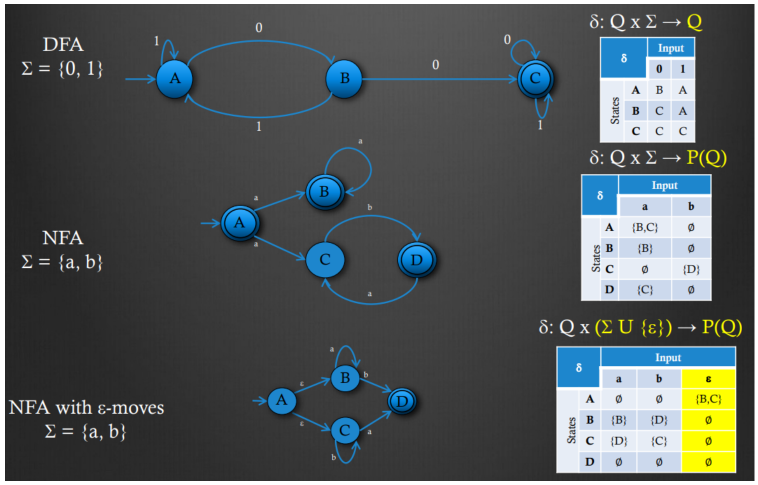
$$L = (b^*a) + (a^*b).$$



- **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)

- **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

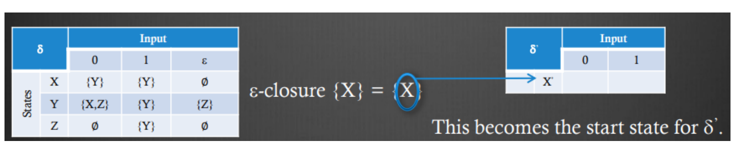
  Things to note about the conversion:

  - 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:

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

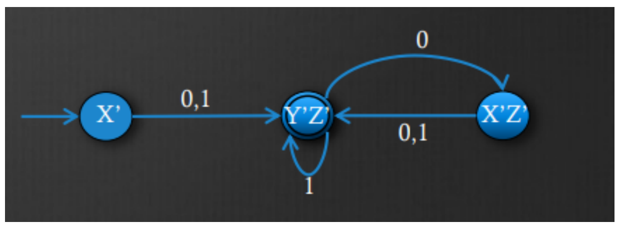
and since $X'Z'$ is a new state, we add it to δ'.

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***

- **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**

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**
- **Finding the intersection of two NFA's**
- **Convert DFA into RE**: To convert a DFA (Deterministic Finite Automaton) into a Regular Expression (RE), you can use the state elimination method or generalized transition automaton method. This process works by gradually reducing the DFA's states and transitions until only a regular expression representing the entire language remains.

  Given a DFA $M(Q, \Sigma, \delta, q_0, F)$, the goal is to find a regular expression that represents the language recognized by this DFA.

  Process:

  1. **Add a new Start and accept state**:
     - Add a new start state q_s with an ϵ-transition (empty string) to the original start state q_0.

       Note: Once you add the new start state q_s with an ϵ-transition to the original start state q_0, q_0 is no longer considered the start state. Instead, q_0 becomes just another intermediate state in the automaton. The new start state is q_s, and it immediately transitions to q_0 without consuming any input (via the ϵ-transition).
     - Add a new accept state q_f and add ϵ-transitions from each of the original accept states to this new accept state q_f.

       Note: Similarly, when you add the new accept state q_f and connect it via ϵ-transitions from the original final states in F, the original final states are no longer considered final states in the sense of marking the end of a string's acceptance. Now, the new final state q_f serves as the sole final state, and the automaton reaches q_f via ϵ-transitions from the original final states.

     These new states simplify the process because now there's exactly one start state and one accept state.
  2. **Eliminate States One by One**:
     - The idea is to progressively eliminate states from the DFA while updating the transitions between the remaining states with regular expressions.
     - Every time you eliminate a state r, you need to update the regular expressions on the transitions between the remaining states to account for the paths that go through r.

     For any three states p, r, and q, if there is a path from p to q that goes through r, the new transition after eliminating r will include the regular expression:

     $$R(p \to q) = R(p \to q) + R(p \to r)R(r \to r)^* R(r \to q)$$

Where:

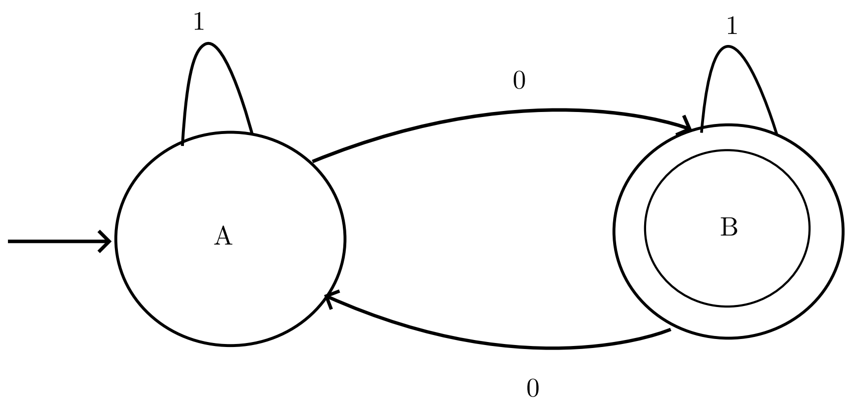
- $R(p \to q)$ is the regular expression for the direct transition from p to q.
- $R(p \to r)$ is the regular expression for the transition from p to r.
- $R(r \to r)$ is the regular expression for the loop on state r.
- $R(r \to q)$ is the regular expression for the transition from r to q.
- $+$ represents union, and $*$ represents the Kleene star (zero or more repetitions).

After updating the transitions, remove the state r.

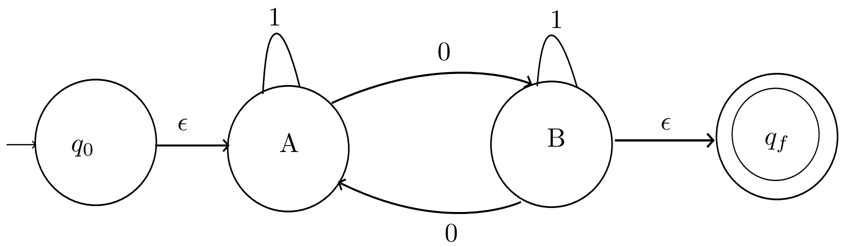
3. **Repeat the Elimination Until Only Two States Remain**: Continue eliminating states and updating the transitions until only two states remain: the start state q_s and the new accept state q_f.

   At this point, the regular expression on the transition from q_s to q_f represents the language of the DFA.

Example: For the alphabet $\Sigma = \{0, 1\}$, let's take the machine that accepts the strings with any number of ones, but the total number of zero's must be odd, and convert it to a RE.

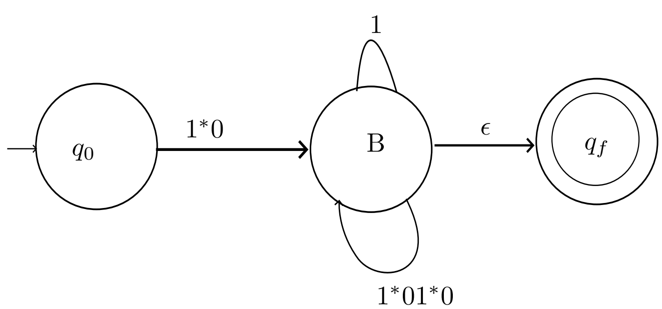


Let's start by making the new start and end states

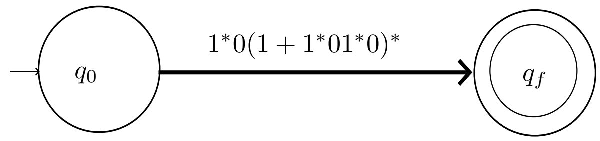


Now we start eliminating states, note that it does not matter in which order we eliminate the states, but for this example we will begin by eliminating state A. To get from the start state q_0 to state B, we need to pass through A, to get from A to B, we can have any number of $1's$ followed by a zero which takes us to be. Thus, the transition from q_0 to B is the regular expression 1^*0

We also have to consider the original transition from B to A, and then back to B, for this we have the RE 1^*01^*0. Thus the machine becomes



To eliminate B, we need to consider the transitions through B ie from q_0 to q_f. We know to get from q_0 to B we have the RE 1^*0, then from B to q_f we have $(1+1^*01^*0)^*$. Thus, the transition for q_0 to q_f is $1^*0(1+1^*01^*0)^*$. And the final machine with only one regular expression is



DSA

2.1 C++ Stuff

2.1.1 Type declarations: Definitions and theorems

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

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

  The Algorithm:

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

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

  Examples:

  – a is an int $\implies$ `int a`
  – a is a pointer to an int $\implies$ `int * a`
  – a is a pointer to a constant int $\implies$ `int const * a` (also `const int * a`)
  – a is a constant pointer to an int $\implies$ `int * const a`

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

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

  Examples:

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

    Note: Notice how we use parenthesis to change direction

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

  Examples:

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

2.1.2 G++

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

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

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

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

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

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

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

    Thus,

    ```
    #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$_64-linux/gnu/libm.so.6islinkedas-lmTheX11graphicslibrary/usr/lib/x86_64$ $linux-gnu/libX11.soislinkedas-lX11$

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

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

2.1.3 Makefiles

Databases

3.1 Introduction to databases (db concepts)

3.1.1 Definitions and theorems

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

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

Note: The storage device is controlled by the DBMS

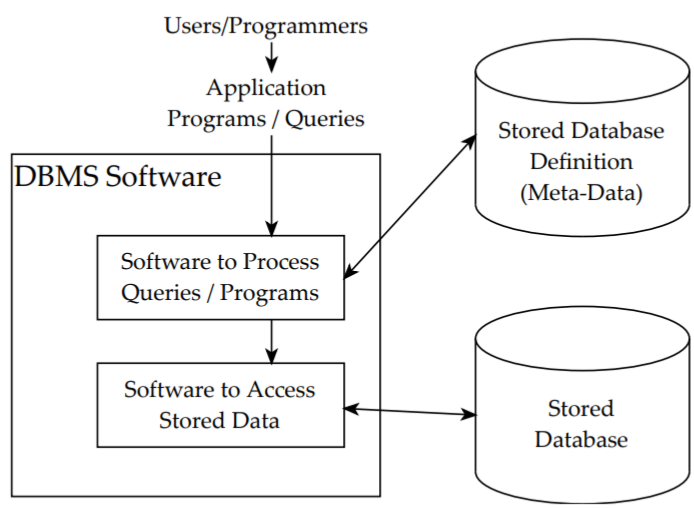
- **Manipulating a Database**
  - retrieve specific information in a query
  - update the database to include changes
  - generate reports from the data

  Most likely already defined by whatever dbms you choose

- **Sharing a Database**: Sharing a database Allows multiple users and programs to access the database at the same time, any conflicts between applications are handled by the DBMS

- **Other Important Functions of a Database**: Other important functions provided by a DBMS include
  - Protection, system protection, security protection
  - Maintenence, allows updates to be performed easily

- **Simplified Database System Environment**:



- **Main characteristics of a database system are:**
  - Self-describing nature of a database system
  - Insulation between programs and data, and data abstraction
  - Support for multiple views of the data
  - Sharing of data and multi-user transaction processing

- **Other Capabilities of DBMS Systems**: Support for at least one data model through which the user can view the data, There is at least one abstract model of data that allows the user to see the "information" in the database, Relational, hierarchical, network, inverted list, or object-oriented

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

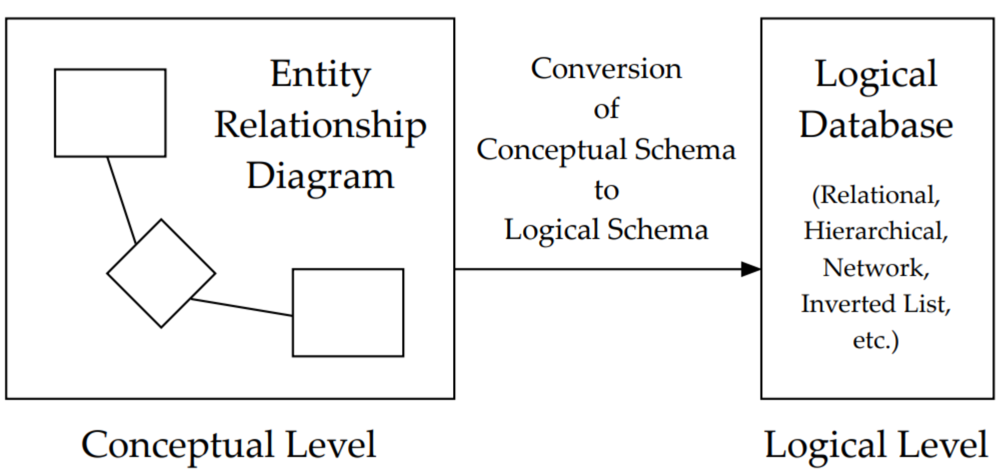
- efficient file access which allows us to "find the boss of Susie Jones"
- allows us to "navigate" within the data
- allows us to combine values in 2 or more databases to obtain "information"

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

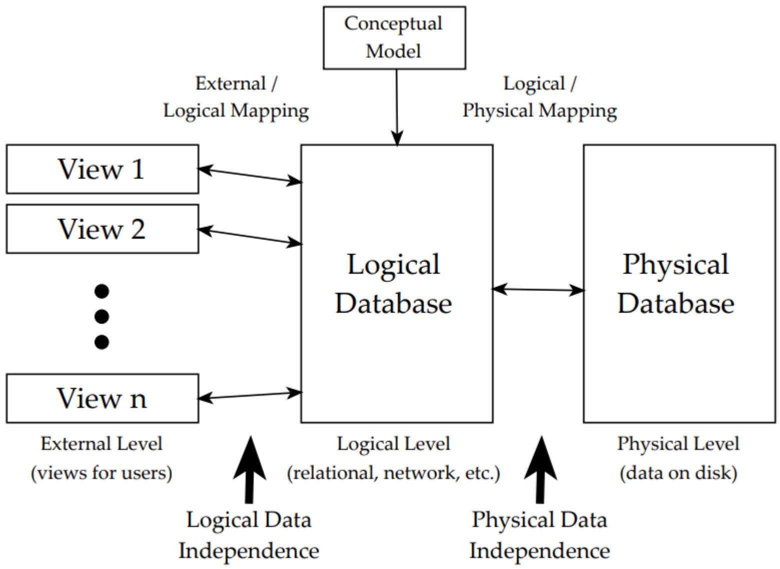
- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Data Control Language (DCL)
- query language access data
- operations such as add, delete, and replace

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

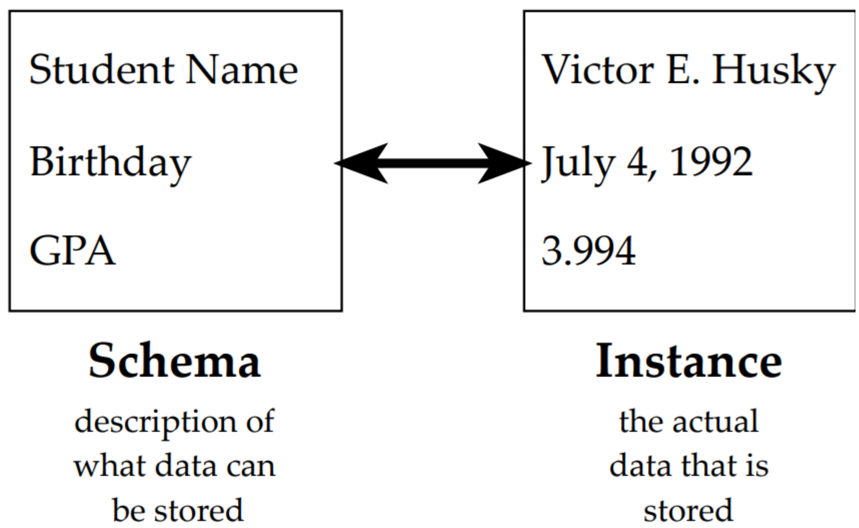
  Note: Obviously, some of these would require more than just software - offsite back-ups, etc
- **Use of Conceptual Modeling**:



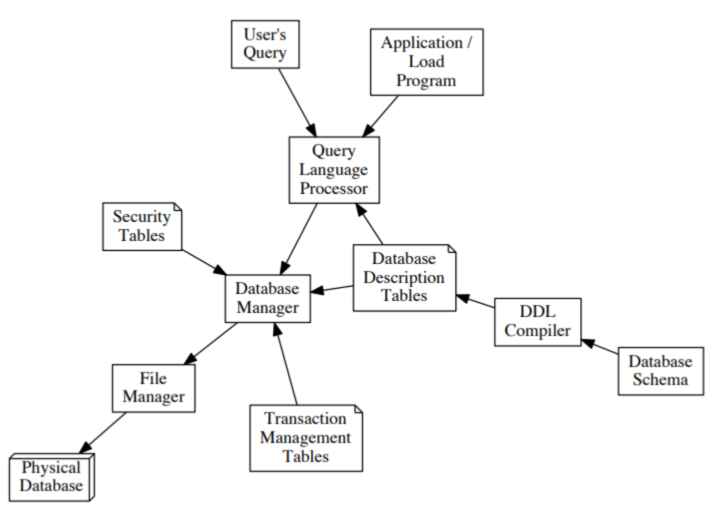
- **Leveled Architecture of a DBMS**:



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



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



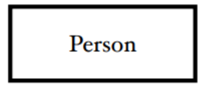
- **Overall DBMS Usage Scenario**: Database Administrator (DBA) define the conceptual, logical, and physical levels using DDL. DBMS software stores instances of these in schemas. User defines views (External Schema) in DDL. User accesses database using DML
- **Advantages of a Database**:
  - Controlled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

3.2 Conceptual Modeling and ER Diagrams

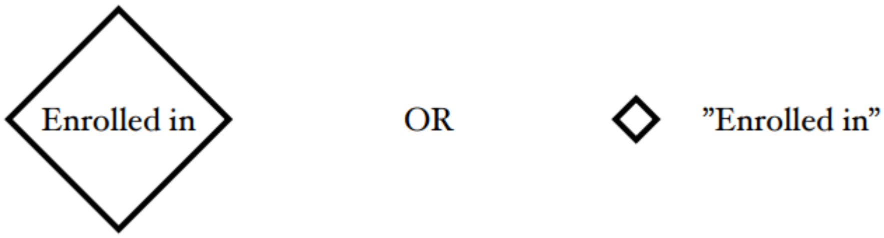
3.2.1 Definitions and theorems

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

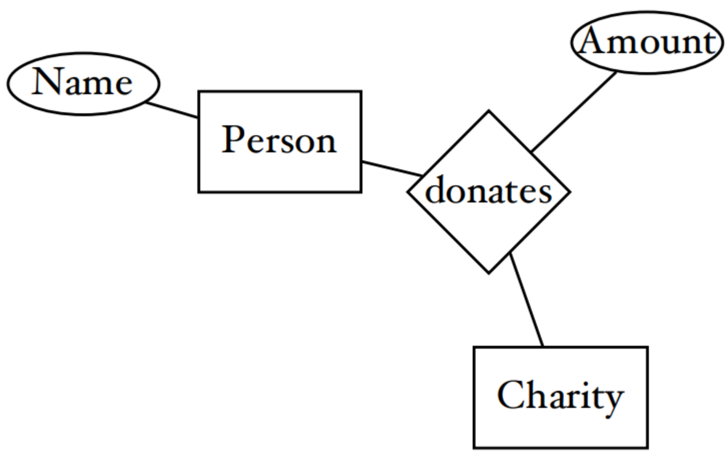
[1] ”Non-atomic data values” refer to data structures or values that are composed of multiple components, as opposed to atomic data values, which are indivisible and represent a single value.



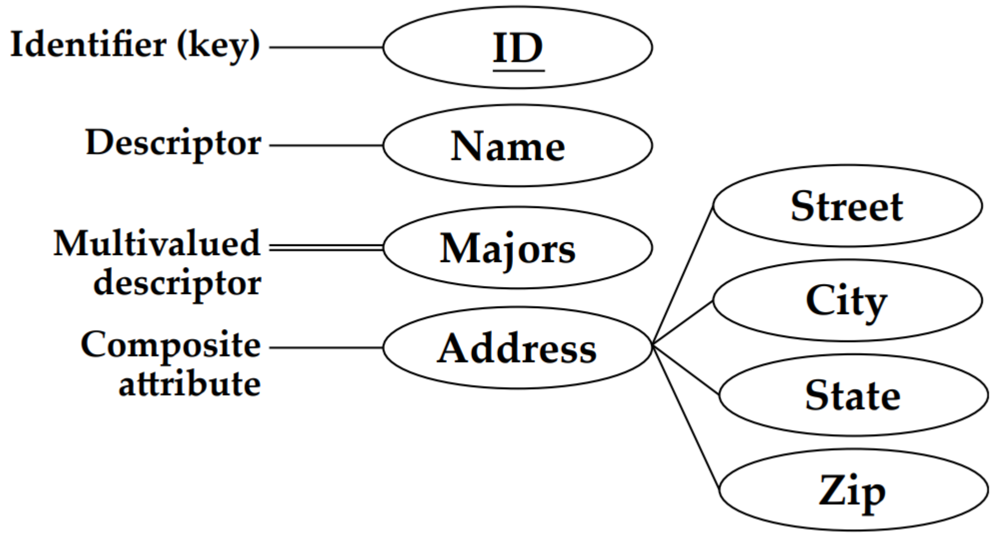
- **Relationships**: Relationships connect one or more entities together to show an association. A relationship *cannot* exist without at least one associated entity. Graphically represented as a diamond with the name of the relationship inside, or just beside it



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

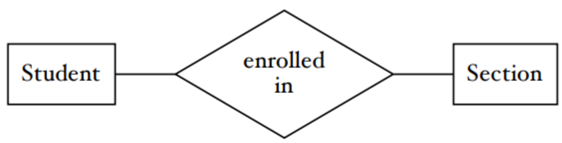


- **Attributes on Entities**: When an attribute is attached to an entity, it is expected to have a value for every instance of that entity, unless it is allowed to be null. For instance, in the diagram above, Name was an attribute of Person. Every person that we store data about will have a value for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

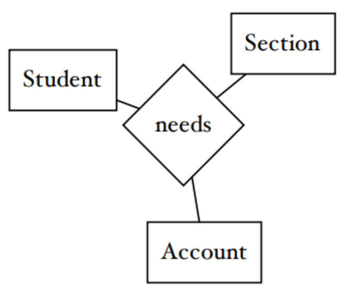


- **Degree of a Relationship**: The degree of a relationship is defined as how many entities it associates. If one entity is associated more than once (such as with a recursive relationship), then the degree counts each time it is referenced.

  - binary



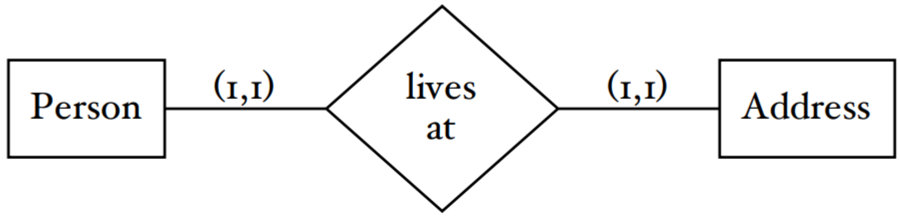
  - ternary



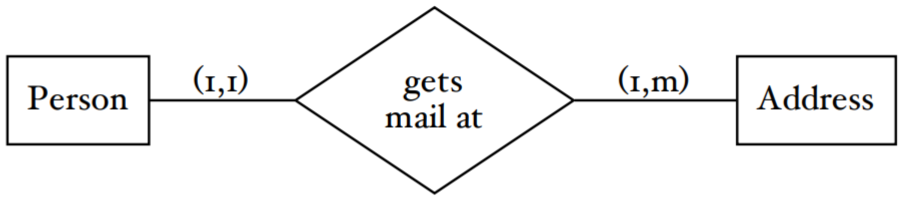
  Note: There is no limit to how many entities there can be in a relationship. After binary, and ternary, we start to call the relationships n-ary, where n is the degree

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

▶ one-to-one

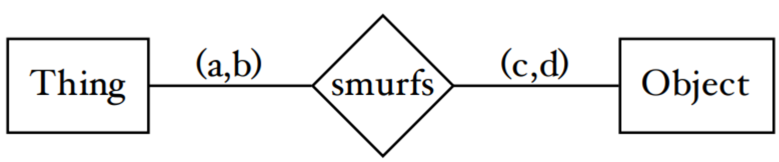


▶ one-to-many



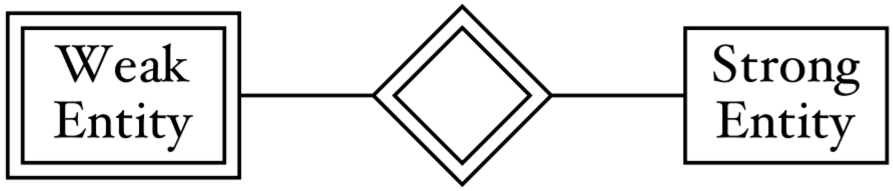
Together (from the image) both sides make up the connectivity, to refer to a single side, we use the term "cardinality", ie the cardinality of a person is (1,1). If we hold Address constant (We know a specific address and are therefore refering to that), how many persons may live at that address, in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

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

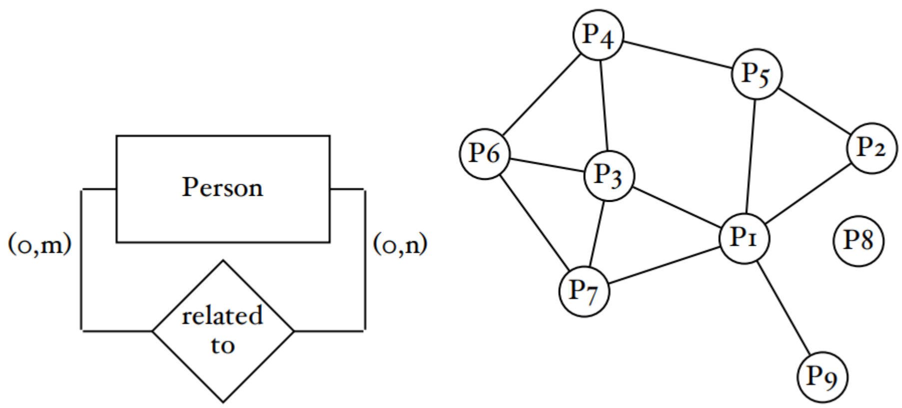


- **Weak Entities**: Sometimes you may run into an entity that depends upon another entity for its existence. The weak entity is a tool you can use to represent this.:w

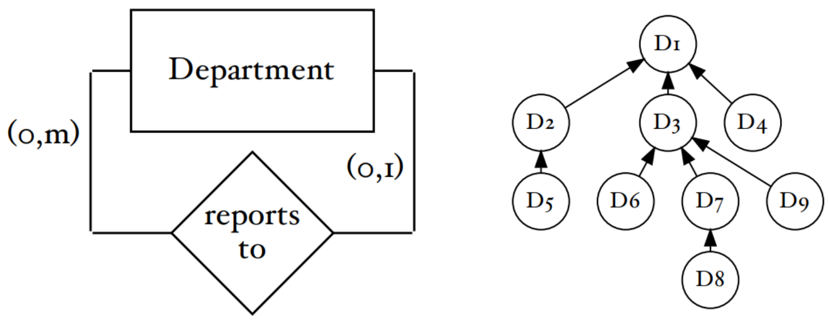
  Weak entities are written like normal entities, except that they have a double rectangle outline. The relationship that connects the weak entity to the strong entity it depends upon will be written with a double diamond. This does not mean that the relationship is weak. It is just to indicate upon which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



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



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

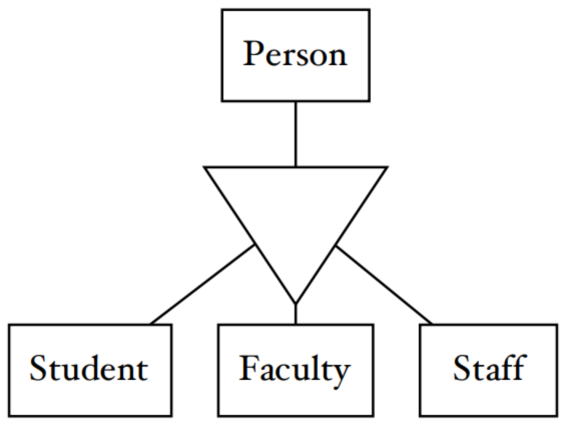


- **Entity or Attribute?**: Sometimes it isn't clear whether something should be an entity or an attribute of some other entity. Usually the decision will come down to how complicated it is to store the data, and how important it is. If it ends up being used in multiple places, it might be a clue that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

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

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

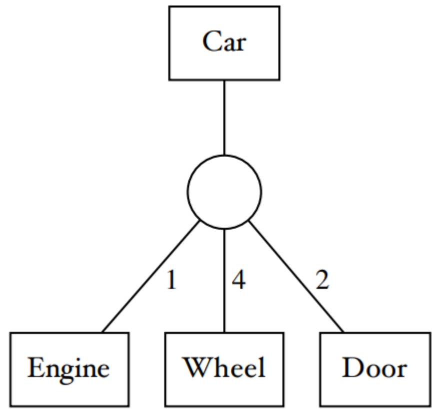
- **IS A Inheritance**: This type of inheritance happens when you have a supertype and one or more subtypes that are members of the supertype. Denoted by an upside-down triangle, with the supertype on top, and the subtypes coming out the bottom.



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

  They are mutually exclusive so you need to pick one of each, ie. GO, GD, SO, SD

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

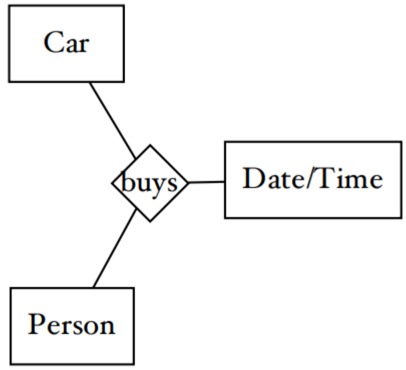


- **Warning about IS-PART-OF**: The IS PART OF inheritance operator does have its uses, but it is not very commonly used, If you see something involving a certain number of things being present, there are several possibilities
  - Sometimes a number is specified that isn't actually important for what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

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

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

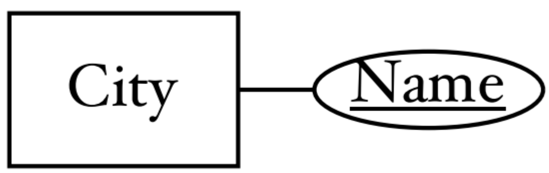
Adding a new entity to the relationship for the date/time of the purchase can fix this problem.



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

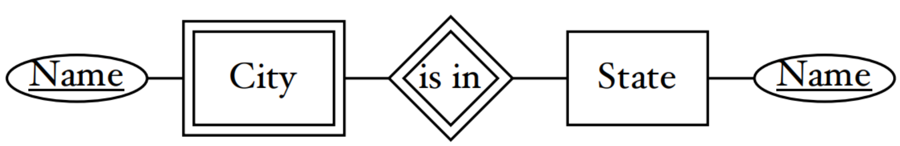
- **Weak Entities - Introduction**: So far, all of the entities we have used have been things that stand on their own. There are some situations where we are modeling an object for which we certainly need to store data, but the items exist only in the context of some other entity. Many of these examples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

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



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

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



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