# **Week 1 Material (Sets, Logic, Integers, Proofs)**

## **Truth Tables**

Negation (NOT): ￢S = Opposite of S

Conjunction (AND): A∧B = True only when A, B are both true

Disjunction (OR): A∨B = False only when A, B are false

Conditional (IF-THEN): A⟶B = False only when A is true, B is false

## **Logical Equivalences**

DeMorgan’s Laws:

* ￢(A∧B) is equivalent to (￢A)∨(￢B)
* ￢(A∨B) is equivalent to (￢A)∧(￢B)

Others:

* A∧(B∨C) is equivalent to (A∧B)∨(A∧C)
* A∨(B∧C) is equivalent to (A∨B)∧(A∨C)
* A⟶B is equivalent to (￢A)∨B
* ￢(A⟶B) is equivalent to A∧(￢B)

## **Proofs and Definitions**

d | n = For some integer *k*, *n* = *d\*k* where *d* ≠ 0

(d | a)∧(a | b) ⟶ (d | b), (d | (ax + by) *for any integers x and y*

Integers (Z) = …, -2, -1, 0, 1, 2, …

Natural numbers (N) = 0, 1, 2, 3, ...

Prime number = An integer *p* > 1 if 1 and *p* are its only positive divisors

Proof by exhaustive checking: Requires a finite # of things to check

Disproving by exhaustive checking: Finding a counter-example

(Conditional) Direct approach: Assume A is true, prove B is true

(Conditional) Contrapositive: Assume B is false, prove A is false

If and only if (iff) proofs: Prove both A⟶B and B⟶A

Proof by contradiction: Assume S is false, demonstrate contradiction, assumption must be false, S must be true

* Statement:
* Let’s assume the statement is false, that is:
* Then it logically follows that:
* Contradiction:
* Thus, we have proven by contradiction that:
* QED

## **Sets**

A simple set looks like this: {a, b, c, d}.

* Same as {c, a, b, d}, {a, b, a, d, c}
* Order doesn't matter
* Must have curly brackets
* Can have duplicates

The union of two sets is putting the elements of 2 sets together.

(Ex. {a, b, c, d}∪{c, d, e, f} = {a, b, c, d, e, f})

The intersection of two sets is putting the elements of 2 sets that coincide together.

(Ex. {a, b, c, d}∩{c, d, e, f} = {c, d}; {∅}∩{a, b, {{}}, c, {d}} doesn’t contain ∅)

* The intersection of an empty set ∅ and a set is always just the empty set ∅

The empty set has nothing in it, represented by ∅ and { }.

A is a subset (⊆) of B if *everything inside A* (curly brackets) is an element of B.

(Ex. {a, b} is a subset {a, b, c, {d}} since a and b are both in the second set.)

* ∅ is always a subset of any set.
* Both A and B MUST be sets in order to compare them.
* A is a subset of B iff A∩B = A.
* A is a subset of B iff A∪B = B.

A is an element (∈) of B if *all of A* is inside set B (A doesn’t have to be a set).

(Ex. a is an element of {a, b, c, {d}} since a is within the second set.)

# **Week 2 Material (Graphs, Strings, Languages)**

## **Graphs**

Graph = a set of vertices (nodes) connected by edges

two graphs equivalent if vertices adjacent to each other are the same

Adjacent = any pair of vertices connected by an edge

Planar = a graph that can be drawn on a plane so that no edges intersect

Directed graph = a graph where each edge points in one direction

Degree of a vertex = number of edges it touches (add 2 if it has a loop)

Subgraph = a graph (V’, E’) is a subgraph of (V, E) if V’⊆V and E’⊆E

Path = sequence of vertices you travel through in a tuple (order matters)

Cycle = a path whose beginning/ending vertices equal, no edge occurs > 1 time

Acyclic = a graph with no cycles

Length of a path = # of edges

### **Regular Graphs**

Vertices are a set. (Ex. {1, 2, 3, 4, 5, 6}.)

Edges are a set of sets. (Ex. {{1, 2}, {1, 5}, {2, 3}}.)

(V, E) is a pair. (Ex. ({1, 2, 3, 4, 5, 6}, {{1, 2}, {1, 5}, {2, 3}}).)

### **Directed Graphs**

Vertices are a set. (Ex. {1, 2, 3, 4, 5, 6}.)

Edges are a set of pairs (order matters; (start, end)). (Ex. {(1, 2), (1, 5), (2, 3)}.)

(V, E) is an ordered pair. (Ex. ({1, 2, 3, 4, 5, 6}, {(1, 2), (1, 5), (2, 3)}).)

Weighted graphs have edges represented as 3-tuple: (source, definition, weight)

## **Strings / Languages**

String = finite ordered sequence of 0 or more symbols

String length = # of elements in that string (notation: | |)

Λ (lambda) = empty string, has 0 length

Alphabet = a finite set of symbols, can be empty

A\* = the set of all strings over alphabet A

Concatenate two strings = placing them next to each other to make a new string

Language = a set of strings, can be empty/infinite; ∅ a language over any alphabet

Product of 2 languages = LM: take 1 string from L, 1 string from M, concatenate

∅ ≠ {Λ}

s*n* = *n* copies of symbol s

L*n* = L multiplied by itself *n* times; L0 = {Λ}

L\* = {L0∪L1∪L2∪…}, set of all possible concatenations of strings from L

L+ = {L1∪L2∪L3∪…}, same as L\* but doesn’t include L0 (not valid in reg. expres.)

The concatenation of a language and an empty set = {∅}.

Stuff never used as symbols: Λ, ∅, groups of English letters, set/tuple symbols

# **Week 3 Material (Regular Langs / Expressions)**

Regular languages = if language can be expressed as regular expression or recognized by a DFA or NFA

Regular expressions = are all regular languages

Base case rules (for regular langs/expressions):

1. ∅ is a regular language/expression over alphabet A
2. {Λ} (the set containing lambda) is a regular language/expression over A
3. {a} is a regular language/expression over A for all a∈A

Inductive step (for regular langs): If L & M are regular languages over A, then

1. L∪M is a regular language over A
2. ML is a regular language over A
3. L\* is a regular language over A

Inductive step (for regular express.): If R & S are regular expressions over A, then

1. (R) is a regular expression over A
2. R + S is a regular expression over A
3. RS is a regular expression over A
4. R\* is a regular expression over A

“The language of regular expression Q is R. L(Q) = R”

L(a + b + c) = {a, b, c}

L((a + b)(b + c) = {a, b}{b, c}

L(∅) = ∅ = {}

# **Week 4 Material (Reference for Properties)**

1. (+ properties)

R + T = T + R **plus 1.1**

R + ∅ = ∅ + R = R **plus 1.2**

R + R = R **plus 1.3**

(R + S) + T = R + (S + T) **plus 1.4**

2. (• properties)

R∅ = ∅R = ∅ **dot 2.1**

RΛ = ΛR = R **dot 2.2**

(RS)T = R(ST) **dot 2.3**

3. (Distributive properties)

R(S + T) = RS + RT **distrib 3.1**

(S + T)R = SR + TR **distrib 3.2**

(Closure properties)

4. ∅\* = Λ\* = Λ **closure 4.1**

5. R\* = R\*R\* = (R\*)\* = R + R\* **closure 5.1**

R\* = Λ + R\* = (Λ + R)\* = (Λ + R)R\* = Λ + RR\* **closure 5.2**

R\* = (R + … + R*k*)\* for any *k* ≥ 1 **closure 5.3**

R\* = Λ + R + … + R*k* - 1 + R*k*R\* for any *k* ≥ 1 **closure 5.4**

6. R\*R = RR\* **closure 6**

7. (R + S)\* = (R\* + S\*)\* = (R\*S\*)\* = (R\*S)\*R\* = R\*(SR\*)\* **closure 7**

8. R(SR)\* = (RS)\*R **closure 8**

9. (R\*S)\* = Λ + (R + S)\*S **closure 9.1**

(RS\*)\* = Λ + R (R + S)\* **closure 9.2**

Remember to:

* Number the steps
* Along with the property, list the variables used
* Start with the left side of the equation first
* Put equals signs in each line
* End with QED

# **Week 5 Material (DFAs & NFAs)**

DFA is a finite directed graph and stands for deterministic finite automata.

* Each state emits one labelled out transition for each symbol in alphabet A
* Has one special “start state”
* Has a set of “final states” (can have 0 to *n* nodes, not infinite), double circles
* Final state means that if you land there, the string is in the regular language
* If no final states, only language it accepts is the empty set ∅
* If all final states, the language it accepts is A\*
* “Trash state” is state you’ll never leave once you get there

Columns of DFA transition table are: State, all symbols in alphabet

* Values are like normal

NFA are like DFAs, but stand for nondeterministic finite automata, and:

* Can have 0, 1, or more out transitions for every symbol in the alphabet
* Can have lambda transitions that you can take whenever you want
* Easier to construct than DFA
* Accepts a string if there’s any path that processes it + ends up at final state

Columns of DFA transition table are: State, all symbols in alphabet, lambda

* Alphabet in tuple does NOT contain lambda
* All items in transition tables are sets
* Table can have empty sets if no states have those symbol transitions

Formal description for both DFAs and NFAs:

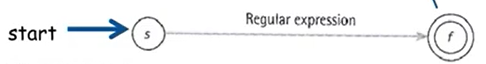
(states, alphabet, transition fn, start, final)

Ex. ({0, 1, 2}, {a, b}, T, 0, {1})

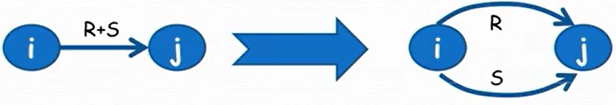
# **Week 7 Material (Mealy Moore & more)**

To turn a regular expression into an NFA:

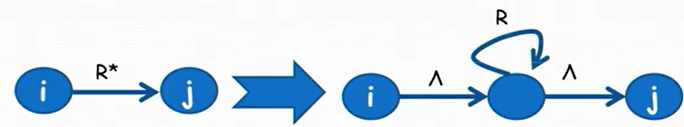
* Step 1: Write the regular expression a new kind of machine



* Step 2: Keep applying the following 3 rules until all edges have letter or ^

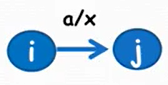






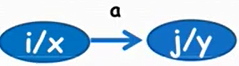
To turn a DFA or NFA into a regular expression:

* Step 1: Create a new start state *s* and draw a new edge labelled with ^ from *s* to the original start state
* Step 2: Create a final state *f* and draw new edge(s) labelled with ^ from all the accept states to *f*. Eliminate old final states, and replace all double edges with a single one.
* Step 3: A node *k* is “on route between” nodes *i* and *j* if:
  + *i* ≠ *k*
  + *j* ≠ *k*
  + old(*i*, *k*) ≠ ∅
  + old(*k*, *j*) ≠ ∅
  + *Note:* It’s OK if *i* == *j*

Call the node you want to get rid of *k*. For every pair *k* is “on route between”, new(*i*, *j*) = old(*i*, *j*) + old(*i*, *k*)old(*k*, *k*)\*old(*k*, *j*)

How to interpret a Mealy Machine: If you start at state *i* AND you see the letter *a*, then go to the state *j* AND print an *x*.

* It has no final states
* Its ***transitions*** produce output
* It doesn’t accept or reject input; instead, it generates output from input
* It cannot have non-deterministic states (multiple transitions w/ the same input)



How to interpret a Moore Machine: When you enter the *i* state, you print the letter *x*. If you are at state *i* AND you see the letter *a* in the string, you go to state *j* AND print a *y*.

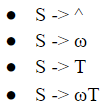
* It has no final states
* Its ***states*** produce output
* It doesn’t accept or reject input; instead, it generates output from input
* It cannot have non-deterministic states (multiple transitions w/ the same input)

# **Week 8 Material (Grammars)**

A grammar consists of:

* A finite set *N* of grammar symbols called non-terminals (LHS of rules)
* A finite set *T* of symbols called terminals, where *N* ∩ T = Ø
* A non-terminal, not necessarily always called *S*, called the start symbol
* A finite set of production rules of the form where they are both strings over the alphabet *N* ∪ *T* with the restrictions that
  + is not the empty string
  + There is at least one production with S alone on the left hand side
  + Each non-terminal must appear on the LHS of some production

If ^ is part of a grammar, then it is one of the symbols in the set of terminals.



A grammar is a regular grammar if each production takes one of the following forms where the uppercase letters are non-terminals and ω is a non-empty string of terminals, shown right:

* Only one non-terminal on RHS
* It must appear on the far right side of the RHS
* Grammars are NOT unique (there can be 2 different grammars for the same language)
* The same can be said for regular expressions and NFAs

A grammar is called a context-free grammar if each production takes the following form

where S is non-terminal and is a non-empty string of terminals and non-terminals. They’re less stricter. However, something important to note:

* Every regular grammar is also context-free
* Some context-free grammars are NOT regular grammars

A sentential form is a string made up of terminals and non-terminals.

* If *x* and *y* are sentential forms and is a production rule, then the replacement of by in *x* *y* is called a derivation and you denote it by writing
* More notation:
  + derives in one step
  + derives in 1 or more steps
  + derives in 0 or more steps

A leftmost derivation is a derivation that at each step of the leftmost non-terminal of the sentential form is reduced by some production (i.e. always replace the leftmost non-terminal first). There’s also the rightmost derivation.

Theorem: If a language is infinite, there must be a way to loop in your production rules.

* Think in the back of your head the Pigeonhole principle, where you cannot have 28 pigeon holes and 29 pigeons, because there is guaranteed to have 2 in one.
* A production is recursive if its left side appears on the right side. Ex.
* A production is indirectly recursive if A derives a sentential form that contains A. Ex. .
* A grammar is recursive if it contains a production that is recursive or indirectly recursive.
* Are all grammars for infinite languages recursive? YES

(*Week 9 is skipped because the material will not be on the test.*)

# **Week 10 Material (CFGs and PDAs)**

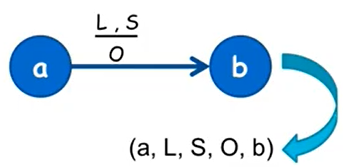
All regular grammars are a subset of context-free grammars, so all regular grammars are context-free grammars, but NOT vice versa.

A language is a context-free language (CFL) if it’s generated by a context-free grammar. To combine CFLs,

* M ∪ N is a context-free language. Its CFG starts:
* MN is a context-free language. Its CFG starts:
* M\* is a context-free language. Its CFG starts:

Anyways, PDA is an NFA with a stack, where the stack is a data structure that has three different operations:

* Push = value will be added to the top of the stack
* Nop = do nothing
* Pop = the top value would be returned as the value from the method



PDA only accepts when at an accept state and the string is processed. For notation, give a series of 3 entries in a tuple in the following format: (state, string remaining, stack), or represent each edge as a 5 tuple: (a, L, S, O, b).

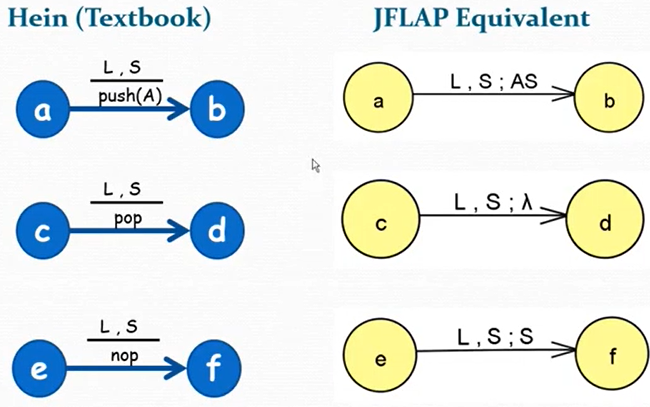
PDAs that accept regular expressions don’t need to use the stack. The stack in the textbook starts with X.

A PDA is deterministic if there is at least one move possible from each state. A PDA is non-deterministic if:

* Two 5-tuples have the same first 3 components
* Two 5-tuples have the same 1st and 3rd component and one of the 5-tuples has a lambda as the second component
* Example: Choices at state 0 have X at the top of the stack and the string can either be an a or lambda

PDAs in JFLAP: How to do stack operations (the stack starts with Z in JFLAP)

* Don’t do anything else (“pop”)
* PUT THAT BACK! (“nop”)
* Put that back AND add something on top too (“push”)



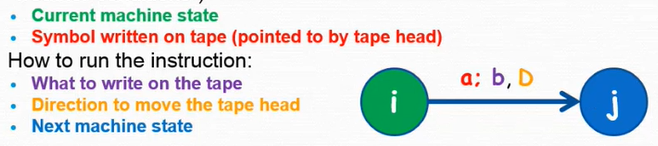
# **Week 11 Material (TMs & Halting Problem)**

The Turing Machine is an equivalent to the PC on the desk and is meant for everything else that DFAs, NFAs, and PDAs can’t cover.

* The input string is written on the tape and all other tape cells contain ^
* The tape head starts at the leftmost symbol of the input string
* There is one start state and one halt state
* The TM stop if there is no valid move or if it arrives at the halt state
* A string is accepted by the TM if the machine enters the halt state

JFLAP uses ☐ while the textbook uses ^ for special symbol blank.

The format:



If a Turing Machine has at least 2 instructions with the same state and input letter, then it is non-deterministic; otherwise, it is deterministic.

The TM we use is equivalent to the TM with a tape that is only infinite in one direction, the TM with multiple heads, and the TM with multiple heads and multiple tapes.

**The Halting Problem Conclusion:** There exists at least one program that cannot be written using any kind of Turing Machine.

# **Week 12 Material (Logic Basics & Equivalence Proofs)**

We represent propositions by formulas called well-formed formulas (WFFs). The WFF alphabet:

* Truth symbols: T (True) & F (False)
* Propositional variables: uppercase letters (P, Q, R, etc.)
* Connectives (operators):

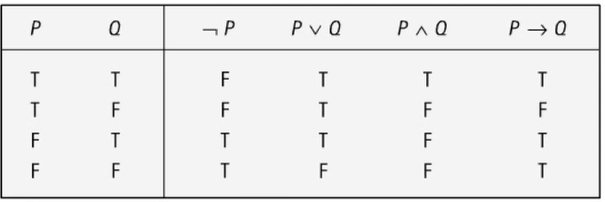
￢ (not, negation) (1 WFF)

∧ (and, conjunction) (2 WFFs)

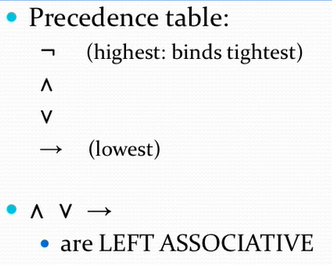
∨ (or, disjunction) (2 WFFs)

⟶ (conditional, implication) (1 WFF from another)

* Parentheses symbols: ( and ) (WFF surrounded by parens)



The truth table for WFF operators:



The precedence table and associatives:

* A WFF is a tautology if its truth table values are all true.
* A WFF is a contradiction if its truth table values are all false.
* A WFF is a contingency if it has both true and false values.

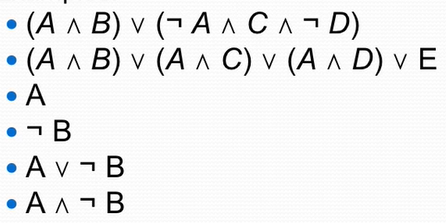
Every WFF has a unique syntax tree and truth table due to the precedence table.

Two WFFs, A and B, are logically equivalent (or equivalent) (A ≡ B) if they have the same truth value for each assignment of truth values to the set of all propositional variables occurring in the WFFs.

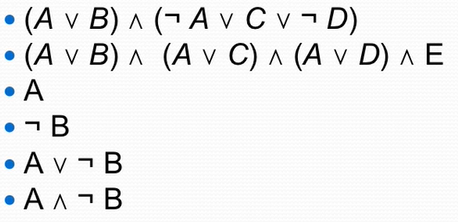
Another way to state equivalence is if and only if (iff) you can prove A ⟶ B and B ⟶ A. Remember that ≡ is an equivalence relation:

* Reflexive: R ≡ R
* Symmetric: If R ≡ S, then S ≡ R
* Transitive: If R ≡ S and S ≡ T, then R ≡ T

A literal is either a propositional variable or its negation. Examples include A, ¬A



Disjunctive normal form (DNF) is a WFF of the form C1∨ or ∨ C*n*, where each C*i* is a conjunction of literals. Examples include the ones shown to the right.



Conjunctive normal form (CNF) is a WFF of the form C1∧ or ∧C*n*, where each C*i* is a disjunction of literals. Examples include the ones shown on the bottom right.

Any WFF has:

* An equivalent WFF in disjunctive normal form
* An equivalent WFF in conjunctive normal form

How to turn any WFF into DNF or CNF:

1. Get rid of the ⟶

* Use the equivalence: A ⟶ B ≡ (¬A) ∨ B (conv 1)

1. Use DeMorgan’s Laws to “push” ¬ into parentheses

* ¬(A ∧ B) ≡ ¬A ∨ ¬B (DeM 1)
* ¬(A ∨ B) ≡ ¬A ∧ ¬B (DeM 2)

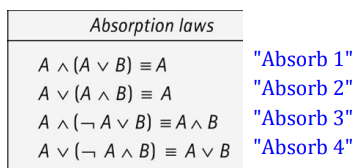
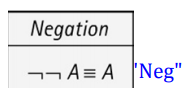
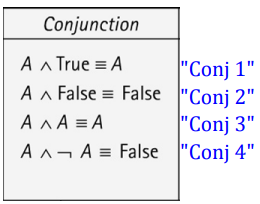
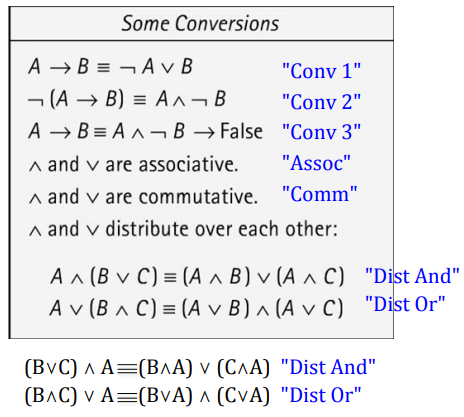
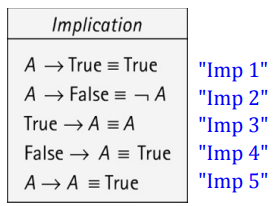
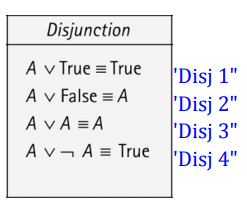
1. Get rid of double negation

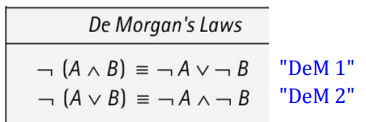
* ¬ ¬A ≡ A (neg)

1. Use distributive and associative equivalences

* A ∧ (B ∨ C) ≡ (A ∧ B) ∨ (A ∧ C) (dist and)
* A ∨ (B ∧ C) ≡ (A ∨ B) ∧ (A ∨ C) (dist or)
* A ∨ (B ∨ C) ≡ (A ∨ B) ∨ C (assoc)
* A ∧ (B ∧ C) ≡ (A ∧ B) ∧ C (assoc)

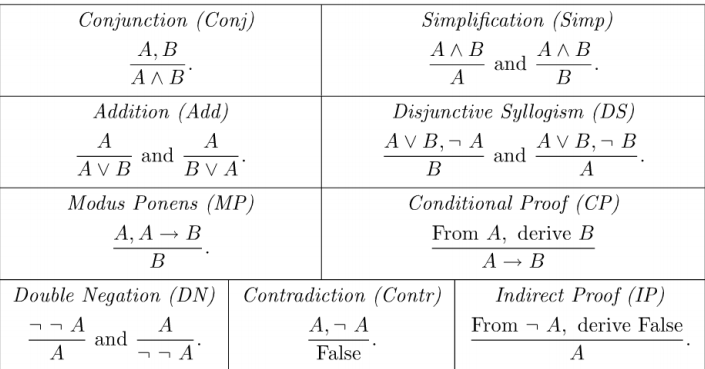
**Logic: Basic Equivalences (Reasons for Proof)**

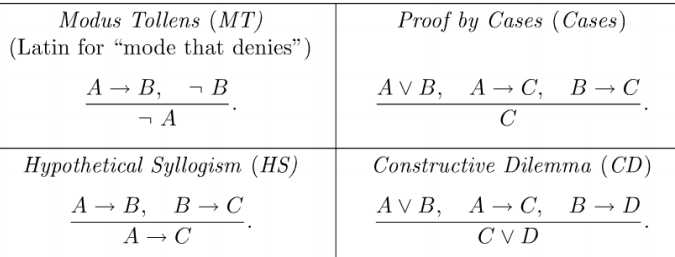




# **Week 13 Material (More Prop Logic & Proofs)**

**Proof Rules**





How to read the rules: If you know the things on the top, then you can conclude that the bottom of the line is true.

An axiom is something you know to be true.

A proof is a finite sequence WFFs such that each WFF is either an axiom or can be inferred from the previous WFFs in the sequence.

The last WFF in a proof is called a theorem. As for proof notation,

* Put each WFF on a numbered line along with a reason.
* Use a letter P for a premise.
* Follow the proof with QED.

If you use CP as the last line of your proof, you just indent and write QED, not the long implication. You also need to cite ALL the lines since the premise. You also can’t use CP or IP anywhere else above (for now).

Proving that WFF is a tautology using IP:

* Start with ¬ WFF as a premise
* Prove False without using CP or IP
* Now, use IP to conclude WFF

A subproof is a proof that is a part of another proof. It starts with a premise and ends by applying CP or IP to the result. Only the very line after the subproof can cite lines within the subproof (think of it as the scope of a local variable).

* You only have a subproof if you’re planning on exiting it with CP or IP
* When you start a subproof, you can have just ONE premise, and you need to write what your CP or IP “exit goal” is (ex. [For B]).

CP and IP can only be used as the last line of a proof or a line right after a subproof. If you use them to exit a subproof, then everything in the subproof is out of scope for the rest of your proof.

# **Week 14 Material (Predicate Logic)**

A predicate is a function or method that “returns” true or false (boolean). For example, “*Let g(x) mean ‘x is green’.*” By itself, it doesn’t have a truth value, so you can create a proposition by using a constant (specific things in the world).

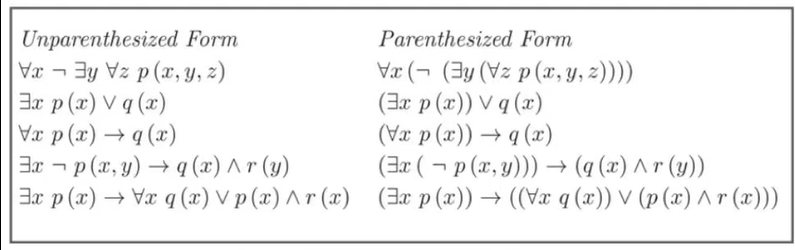
A domain is like the universe of the problem. For example,

* Domain: Rowan students
* Predicate: b(x) means x has a banner ID
* Statement: For all x in our domain, b(x) [aka, all Rowan students have a banner ID]

You can make similar statements for “it is not the case” and “there exists at least one” x.

“For all” = ∀ “It is not the case” = ¬ ∀ “There exists” = ∃ “Not the same” = n “x==y” = eq(x,y)

**Rules of Precedence**

**

Same as propositional logic, plus if any quantifiers or the negation symbol appear next to each other, then the rightmost symbol is grouped with the smallest WFF to its right.

The **scope** of ∃x in (∃x W) is W. Similarly, the scope of (∀x W) is W. In the absence of parentheses, the scope of a quantifier is the smallest WFF immediately to its right.

Examples:

