Set Theory

Part 1

1

What is a Set?

Organizing Information

2

What is a Set?

- A *set* is an unordered collection of “objects”
- The collection objects are also called “members” or "elements"
- One of the most fundamental structures in mathematics

3

Set Notation

- We typically denote a set name using capital letter
- Members are separated with commas and encapsulated within curly brackets

4

Standard Sets

Letter	Name	Members
Z	Integers	…, -2, -1, 0, 1, 2, 3, …
N	Natural Numbers	1, 2, 3, 4, …
Q	Rational Numbers	a / b where both a and b are integers and b is not 0

5

Standard Sets

Letter	Name	Members
R	Real Numbers	All non-imaginary numbers. e.g. 1, 2.5, 3.1415….
U	Universal Set	All values of potential interest (U depends on context)

6

Set Notation: Membership

- Set notation uses a special symbol to denote if an object is a member of a set
- Below, the set V contains vegetables

```
potato ∈ V
```


7

Set Notation: Membership

- This is read as "potato is an element of V"
- …or "potato is a member of V"

```
potato ∈ V
```


8

Set Notation: Not a Member

- There is another special symbol that denotes an object is not a member of a set
- In the example below, the set F contains fluffy animals

```
lizard ∉ F
```


9

Multiple members can be listed

- Multiple elements can be listed
- The expression below states that both *potato* and *carrot* are in the set V.

```
potato, carrot ∈ V
```


10

{ }

Defining Sets

How to specify items

11

Defining Sets

- Sets can be defined a number of different ways
- Each competing notation has advantages & disadvantages – depending on what you are defining

{ }

12

Set Notation: Explicit

- We can *explicitly* define this by listing each element
- For example, we can define a set *S* for members of the Three Stooges

```
S = {moe, larry, curly, shemp}
```


13

Set Notation: Pattern

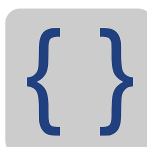
- We can also specify a set by using a *pattern*.
- In the example below we are define a set of integers between 0 and 9.

```
A = {0, 1, 2, ... 9}
```


14

Set Builder Notation

- A set can also be defined using *set builder notation*
- Consists of a variable name, a pipe symbol, and an true/false expression



15

By Characteristic

- The most basic form consists of a variable and an true/false statement
- In this example, everything that satisfies "x is an even integer" will be the set

```
{x | x is a even integer}
```


16

By Characteristic Examples

Expression	Result
{ x | x is an integer }	{ …, -1, 0, 1, 2, 3, … }
{ x | x is an even integer }	{ …, -2, 0, 2, 4, 6, … }
{ x | x is odd natural number}	{ 1, 3, 5, 7, 9, … }

17

Shorthand Notation

- Definitions can also be restricted by another set
- There are two different notations that *mean the same thing*

```
{x ∈ S | true/false expression on x}

{x | x ∈ S and true/false expression on x}
```


18

Characteristic Example

- Remember, Z is the set of all integers
- It reads: *"All x where x is in Z and x is even"*

```
A = {x | x ∈ Z and x is even}
```


19

By Characteristic Examples

Expression	Result
{ x ∈ Z | 0 < x < 5 }	{1, 2, 3, 4}
{ x | x ∈ N and x < 7 }	{1, 2, 3, 4, 5, 6}

20

Characteristic with Structure

- The left-hand-side (before the pipe) doesn't have to be a simple variable name
- It can also be any mathematical expression

```
{f(x) | true/false expression using x}

{y | y = f(x) and true/false using x}
```


21

Let's Try One…

- The second part of the notation must always be a true/false expression
- So, how do we create a set that contains:

{2, 4, 6, 8, 10, …}

22

Let's Try One…

```
First approach:
A = {x | x ∈ N and x is even}

Second approach:
A = {2x | x ∈ N}
```


23

How Does It Evaluate?

- Basically, when you look at something like: { 2x | x ∈ N }, you should do the following
- Steps:
  1. Identify which variables make the right-hand-side true
  2. Plug them into the left-hand-side. These are the values in the set.

24

More Examples

Expression	Result
$\{ 2x + 1 \mid x \in Z \}$	$\{\ldots, -3, -1, 1, 3, 5, \ldots\}$
$\{ x \in Z \mid \text{sqrt}(x) \in Z \}$	$\{0, 1, 4, 9, 16, 25, \ldots\}$

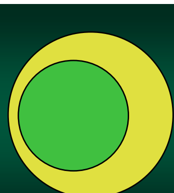
25

Empty Set

- An *empty set* contains no elements
- Can be represented with two curly-brackets (nothing in between)
- There is also a special symbol for empty sets

```
A = { }
A = ∅
```


26



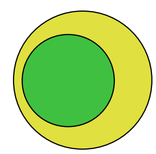
Subsets

Set Your Sets to Fun

27

Subsets

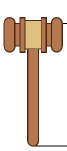
- Commonly, sets are compared to one another using set relationship operators
- Basically, set are defined on elements which they may have in common



28

Subsets

- Set A is considered a subset of set B if all the members of A are also members of B
- The subset operator is similar looking to the member operator



$A \subseteq B$ if and only if:
for all $x \in A$ there is $x \in B$

29

Subsets

- In the example below, the set {1, 4} is a subset of the set {1, 3, 4, 5}
- Note that the reverse is not true.

```
{ 1, 4 } ⊆ { 1, 3, 4, 5 }
```


30

Subsets

- To denote a set is not a subset, we use the subset operator and add a slash
- Below, the set {3, 5} is not a subset of {3, 7} because {3, 7} does not contain 5.

$$\{ 3, 5 \} \not\subseteq \{ 3, 7 \}$$

31

Null is Always a Subset

- A null set contains no elements
- Hence, the null set is always a subset

$$\emptyset \subseteq \{ 2, 3 \}$$
$$\{\} \subseteq \{ 2, 3 \}$$

32

Proper Subsets

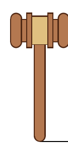
- Set A is a *proper subset* of B if A is a subset of B, but **not** equal to B
- Note: the notation lacks the underline – it is consistent with other operators like < and ≤

$$\{ 3, 5 \} \subset \{ 3, 5, 7 \}$$
$$\{ 1, 2 \} \not\subset \{ 1, 2 \}$$

33

Equality

- Sets A and B are considered equal if-and-only-if… each contain the same elements
- … remember, duplicates don't count



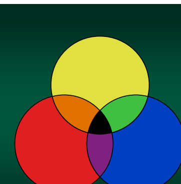
A = B if and only if:
all $x \in A$ there is $x \in B$ and
all $y \in B$ there is $y \in A$

34

Equality

- So, are { 1, 2, 3 } and { 2, 1, 3 } equal?
- How about { 1, 1, 2, 3, 3 } and { 3, 2, 1 }
- Answer is **yes!**
  - order does not matter in a set
  - multiple occurrences does not change if an element is a member

35



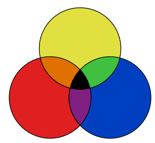
Venn Diagrams

Graphically Representing Sets

36

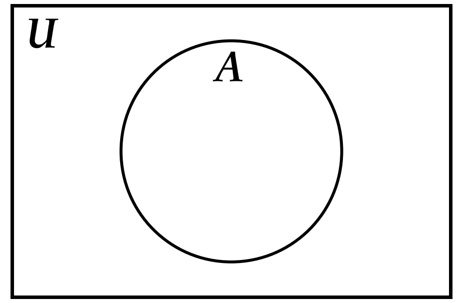
Venn Diagrams

- Sets can also be abstractly representing graphically using Venn Diagrams
- Each set is represented by circle
- Overlaps between each set can show logical relations with set members



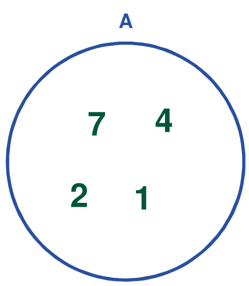
37

Basic Venn Diagram



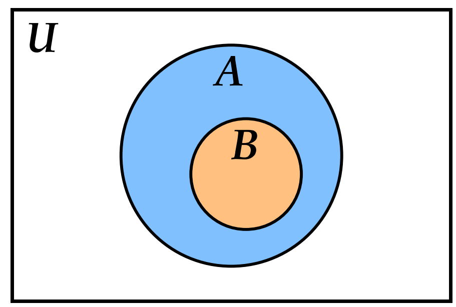
38

Example: A = {2, 7, 1, 4}



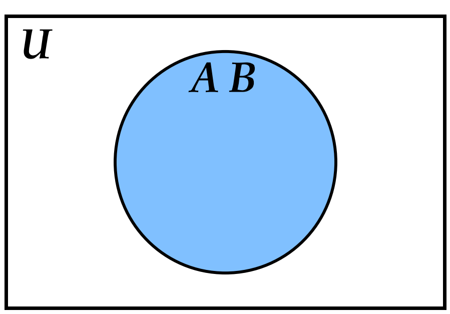
39

Subset Venn Diagram



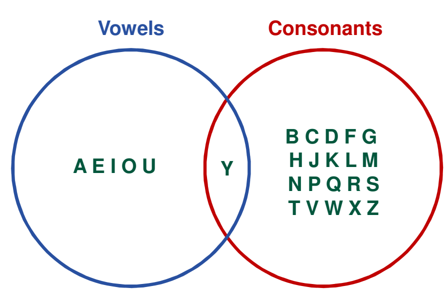
40

Equality

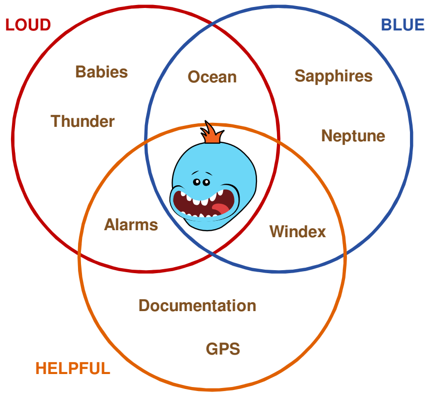


41

Example: Letters in English



42



43

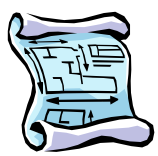
Tuples

Order is Important

44

Tuples & Sets

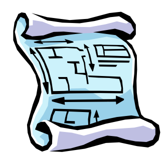
- To denote sets, we delimit the list of members with curly brackets
- For example, the prime numbers between 1 and 10 is {2, 3, 5, 7}
- Order does not matter, so {2, 3, 5, 7} = {7, 5, 3, 2}



45

Tuples

- However, in many cases the *order is important*
- These are called *n-tuples* where "n" is the number of elements
- 2-tuples are also called *ordered pairs*



46

Tuple Notation

- To denote a tuple – we delimit the elements by either parenthesis, angle brackets or square brackets
- Curly-brackets are never used to avoid obvious confusion

```
( 1, 2, 3 )
< 1, 2, 3 >
[ 1, 2, 3 ]
```


47

Tuple Examples

- Order is important, so any element out of position will cause inequality

```
( 1, 2, 3 ) ≠ ( 3, 2, 1 )
```

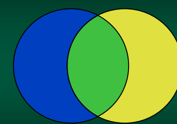

48

Tuple Examples

- Logic generally applies to algorithms since, in procedural programming, order is important
- The following is a tuple of events in California History

```
( Sutter's Fort Built, Bear Flag Revolt,
  Gold Rush, California Joins Union )
```


49



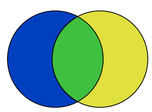
Set Operators

Defining Sets Using Sets

50

Operations on Sets

- New sets can be made from old sets using set operators.
- Just like new numbers can be created from old numbers: 1 + 2 = 3
- So, for the rest of this section, let **U** be the universe, and let A and B be sets



51

Union

- A union of two sets combines all members of each set into a new one
- So, the result is two merged sets

$$A \cup B = \{ x \mid x \in A \text{ or } x \in B \}$$

52

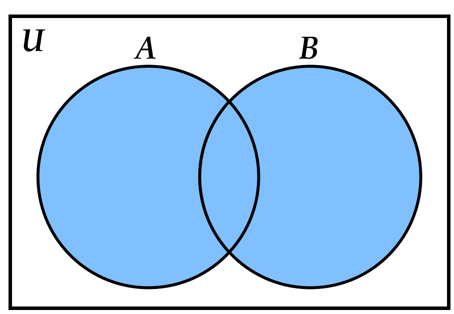
Union

- The symbol ∪ looks like **U**
  - which is also used for the "universe set"
  - be careful not the confuse the two

$$A \cup B = \{ x \mid x \in A \text{ or } x \in B \}$$

53

A ∪ B



54

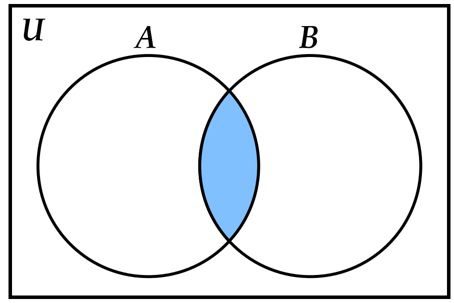
Intersection

- The intersection of two sets contains only those elements that are found in both sets
- So, the result is where the two sets overlap

$A \cap B = \{ x \mid x \in A \text{ and } x \in B \}$

55

A ∩ B



56

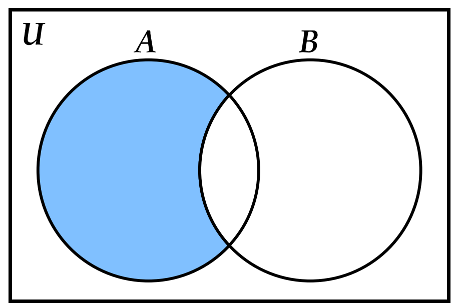
Difference

- *Difference* excludes all items found in one set from another
- Also known as the *relative complement*

$A \setminus B = \{ x \mid x \in A \text{ and } x \notin B \}$

57

A \ B



58

Difference – So Many Notations

- Difference can be written A \ B or A – B (*even though it is not the same as subtraction)*
- Both notations are valid, but some mathematians prefer one over another

```
A - B
A \ B
```


59

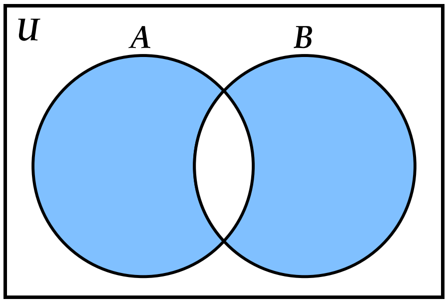
Symmetric Difference

- The *Symmetric Difference* is all the items that are in either of two sets, but not both
- It can be defined two different ways

$$A \oplus B = (A \cup B) \setminus (A \cap B) = (A \setminus B) \cup (B \setminus A)$$

60

A ⊕ B



61

Let's Draw Some…

Let's try it

- So, A ⊕ B has two definitions
- Let's test if both definitions create the same result

$(A \cup B) \setminus (A \cap B)$

$(A \setminus B) \cup (B \setminus A)$

62

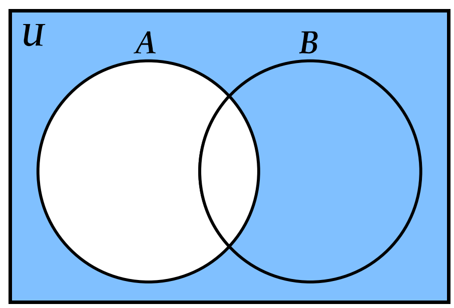
Complement

- The *complement* of a set A, is all elements in the Universe, not in A
- Remember: what elements are in the Universe depends on the sets

$A' = \{ x \mid x \notin A \}$

63

A'



64

Different Notations Used

- Single postfix apostrophe
- An "over bar" (which is underlining on top)
- Superscript "c" for complement

$A' \equiv \overline{A} \equiv A^c$

65

Complement Example

- If set A is a subset of a set B, then the complement of A is all elements not in A but still in B
- Look at the following:

```
A ⊂ B
A = {1, 2, 3}
B = {1, 2, … 10}
```


66

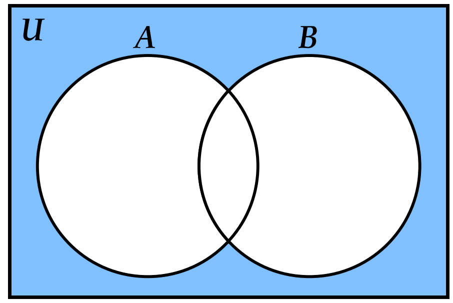
Complement Example

- Set A is a subset of a set B
- Therefore its “universe” is defined as the set of B

```
Therefore…

A' = {4, 5, 6, 7, 8, 9, 10}
```

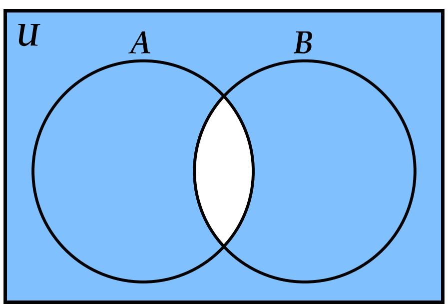

67

(A ∪ B)'



68

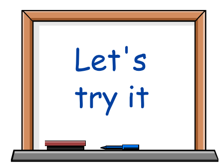
(A ∩ B)'



69

Let’s Draw Some…

- Let’s draw some Venn Diagrams using a several sets
- Using set operators we can highlight any area we want



70

Let’s Draw Some

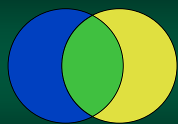

```
U = {a, b, c, d, e, f}

A = {a, b, c}
B = {b, c, d, e}
```


71

Let’s Draw Some

```
Two sets:

In A but not in B: {a}
In B but not in A: {d, e}
In both A and B: {b, c}
In neither A nor B: {f}
```


72



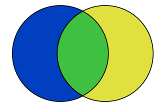
Set Algebra

Just a preview…

73

Set Algebra

- Sets share the same principles as basic math
- You can visually treat the union as an * and the intersection as a +
- You can then factor out sets



74

Communitive Law

- Both ∩ and ∪ are communitive
- This means the left-hand and right-hand operands can be switched

$$A \cap B \equiv B \cap A$$
$$A \cup B \equiv B \cup A$$

75

Idempotent Law

- When a set is combined with itself, it is equivalent to just the statement (no duplicate)
- This applies to both ∩ and ∪

$$A \cap A \equiv A$$
$$A \cup A \equiv A$$

76

Involution Law

- One of the most basic equivalences in logic is the *double negation*
- It is fairly obvious, so not more needs to be said

$$(A')' \equiv A$$

77

Complement Law

- When a set is used with its complement it will result in either the universe or the empty set

$$A \cap A' \equiv \varnothing$$
$$A \cup A' \equiv U$$

78

Complement Law

- Complement Law also can be applied to the Universal Set and Empty Set
- The results should be fairly obvious

$\varnothing' \equiv U$
$U' \equiv \varnothing$

79

Identity Law

- Some operands will have no effect on the truth table of a statement
- In this case, the statement can be simplified

$A \cap U \equiv A$
$A \cup \varnothing \equiv A$

80

Domination Law

- This might look similar to the identity law, but look very careful at which operator is being used
- These will result in either the universe or the empty set

$A \cup U \equiv U$
$A \cap \varnothing \equiv \varnothing$

81

Associative Law

- Some operators in math are *associative*
- For example: (a + b) + c = a + (b + c)
- Same applies to ∩ and ∪

$A \cap (B \cap C) \equiv (A \cap B) \cap C$
$A \cup (B \cup C) \equiv (A \cup B) \cup C$

82

Distributive Law

- Math has operators that are *distributive*
- For example: a * (b + c) = (a * b) + (a * c)
- Works for both ∩ and ∪

$A \cap (B \cup C) \equiv (A \cap B) \cup (A \cap C)$
$A \cup (B \cap C) \equiv (A \cup B) \cap (A \cup C)$

83

Another Look

(A ∪ B) ∩ (A ∪ C)

→ *(A * B) + (A * C) =*
*A * (B + C)*

A ∪ (B ∩ C)

84

DeMorgan's Law

- *DeMorgan's Law* states important rule for logical equivalency
- These are used to convert ∩ to ∪ and vice-versa

85

DeMorgan's Law

- So, it states you can change the operator from ∩ to ∪ or vice-versa
- If you negate both operands

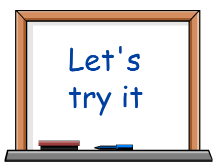
$(A \cap B)' \equiv A' \cup B'$

$(A \cup B)' \equiv A' \cap B'$

86

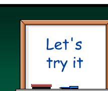
Analyzing DeMorgan's Law

- Let's draw some Venn Diagrams and analyze if DeMorgan's Law works
- First, let's look the logic of what an expression says



87

Let's Draw These…



Let's see if the following are the same:

$A \cup (B \cap C)$

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

88

Algebra: Example 1

(A ∪ B)' ∩ B

A' ∩ B' ∩ B — After using DeMorgan's Law

A' ∩ ∅ = — After using Complement Law

∅ — After using Domination Law

89

Algebra: Example 2

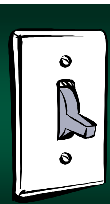
(A ∩ B) ∪ (A' ∩ B)

(A ∪ A') ∩ B — After using Distributive Law

U ∩ B — After using Complement Law

B — After using Identify Law

90

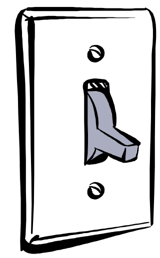


Duals

Don't "Flip" Out

Duels

- The dual of a theorem is created by:
  - flip all **U** and ∅
  - flip all ∪ and ∩
- Important property: *if an expression is an identity then its duel is an identity*



Example Duels

- The two expressions (identities) below are duels of each other
- Both expressions are true, but let's test

```
1. A = (U ∩ A) ∪ (A ∩ B)
2. A = (∅ ∪ A) ∩ (A ∪ B)
```

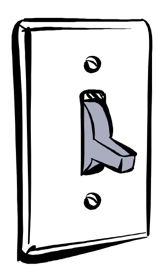

Example: Expression 1

```
A = (U ∩ A) ∪ (A ∩ B)
  = A ∩ (U ∪ B)       Distributive
  = A ∩ U             Domination
  = A                 Identity
```


Example: Expression 2

```
A = (∅ ∪ A) ∩ (A ∪ B)
  = A ∪ (∅ ∩ B)       Distributive
  = A ∪ ∅             Domination
  = A                 Identity
```


Duels



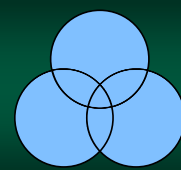
- Also note that the proof (i.e. the reduction) of the expressions was identical
- The number of steps (as well as each law) was the same

Set Attributes

Part 2

1

Fundamental Products

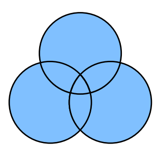


How Many Subsets Are There?

2

Fundamental Products

- *Fundamental Product* is an intersection of each set *(or it's complement)*
- They reveal all the base subsets of interest
- …since, each fundamental product is unique



3

Fundamental Products

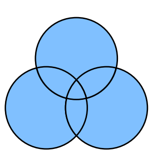
For each set $S_{1..n}$ in the universe, each product, P, is defined:

$$P = A_1 \cap A_2 \cap A_3 \cap A_3 \cap \ldots \cap A_n$$

where A_i is the set S_i or S'_i

4

Some Attributes

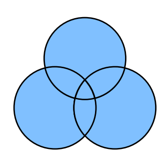


- There are few properties that can be observed from fundamental products
- These will be important in other areas of discrete mathematics

5

Three Major Attributes

1. There are $m = 2^n$ such fundamental products
2. Any two such fundamental products are disjoint
3. The universal set U is the union of all fundamental products



6

#1. Number of Products

- Number of fundamental products *m* grows exponentially in relation to the number of sets *n*
- Observe: this is beginning to look "binary"

$$m = 2^n$$

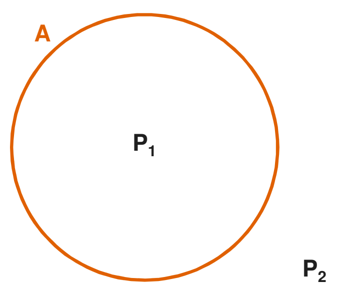
7

With 1 Set

- For a Universe with a single set, A, it results in 2^1 products
- Namely A and A'

P_1 = A
P_2 = A'

8



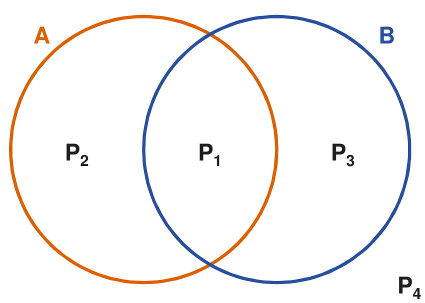
9

With 2 Sets

- With two sets, A and B, there are a total of $2^2 = 4$ products

P_1 = A ∩ B
P_2 = A ∩ B'
P_3 = A' ∩ B
P_4 = A' ∩ B'

10

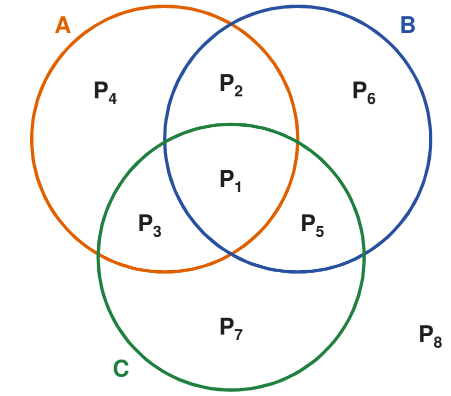


11

With 3 Sets: $2^3 = 8$

P_1 = A ∩ B ∩ C
P_2 = A ∩ B ∩ C'
P_3 = A ∩ B' ∩ C
P_4 = A ∩ B' ∩ C'
P_5 = A' ∩ B ∩ C
P_6 = A' ∩ B ∩ C'
P_7 = A' ∩ B' ∩ C
P_8 = A' ∩ B' ∩ C'

12



13

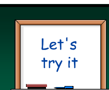
#2. All Products are Disjoint

- Any two different fundamental products are disjoint
- Which means, they have no elements in common

$$P_i \cap P_j = \varnothing \quad \textbf{when } i \neq j$$

14

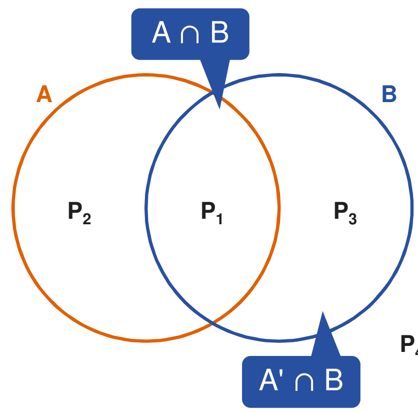
#2. All Products are Disjoint



- We can use set algebra to show *fundamental products* can be unioned into the original sets

$$P_i \cap P_j = \varnothing \quad \textbf{when } i \neq j$$

15



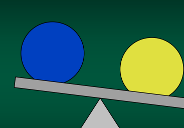
16

#3. Union of all products is U

- The union of all fundamental products is the universe set U
- This should be fairly obvious from what we observed

$$U = P_1 \cup P_2 \cup P_3 \cup P_3 \cup \ldots P_n$$

17



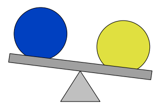
Cardinality

Counting Sets

18

Cardinality of a Set

- The *cardinality* of a set is the number of *distinct* elements
- This information is used in counting – the classification of the set's contents



19

Different Notations Used

- There are two different notations used
- The most common is the | pipe delimiters
- Alternatively, the "n" function is used

$$|A| \equiv n(A)$$

20

Examples

```
A = {1, 3, 5, 7}

|A| = 4
```


21

Examples

Duplicates don't count

```
B = {1, 2, 3, 3, 3, 4}

|B| = 4
```


22

Counting

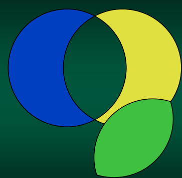
- If the set contains a finite number of elements, it is said to be *countable* – i.e. the cardinality is knowable
- If the set is infinitely large, but the elements can be uniquely identified, then it is *countably infinite*
- Otherwise it is said to be *uncountable*

23

Countable Examples

Set	Result
$\{ x \mid x \in N \text{ and } x \leq 100\}$	Countable
$\{ 2x \mid x \in N \}$	Countably Infinite
$\{ x \mid x \in R \text{ and } 0 < x < 1\}$	Uncountable

24



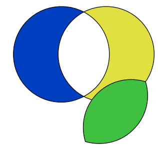
Inclusion-Exclusion

Counting by Subtracting!

25

Inclusion-Exclusion

- Sets can overlap – and can contain the same elements
- So, when counting items in sets, you must be careful not to count an item twice
- *Inclusion-exclusion* principle, can get the correct count



26

Reasoning with Venn Diagrams

Say 100 people receive a questionnaire with two questions:

1. Do you watch The Orville
2. Do you watch Rick & Morty?

60 said 'yes' to The Orville.
35 said 'yes' to **both**.
How many people watch Rick & Morty?

27

Disjoint Set Cardinality

- If sets A and B are disjoint then they have no elements in common
- Cardinality of the union is the sum of the cardinality of both A and B

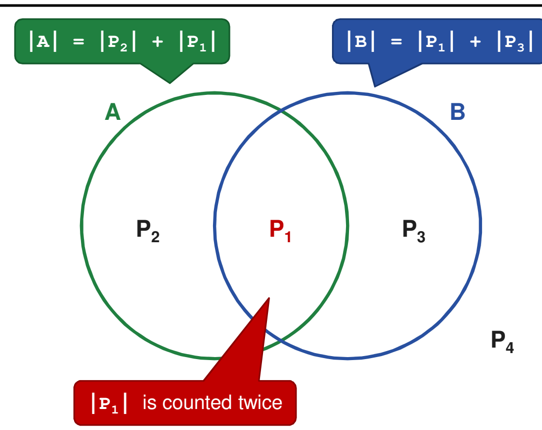
$$|A \cup B| = |A| + |B|$$

28

Set Exclusion

- If sets A and B overlap they have elements in common
- We cannot simply add $|A| + |B|$
- Why? $|A| + |B|$ counts the intersection twice!

29



30

Set Exclusion

- So, we need to remove the duplicate count
- The cardinality of the union is the sum of A and B excluding the intersection

$$|A \cup B| = |A| + |B| - |A \cap B|$$

31

Set Exclusion

- Note: this is the same equation for disjoint sets
- If disjoint, the intersection is ∅
- So, this formula works in all cases

$$|A \cup B| = |A| + |B| - |A \cap B|$$

32

Revisit that Question

Say 100 people receive a questionnaire with two questions:

1. Do you watch The Orville
2. Do you watch Rick & Morty?

60 said 'yes' to The Orville.
35 said 'yes' to both.
How many people watch Rick & Morty?

33

Using the Formula…

- Union of The Orville (T) and Rick & Morty (R) contains 100
- The Orville set contains 60
- The intersection contains 35

$$|T \cup R| = |T| + |R| - |T \cap R|$$

34

Using the Formula…

- Union of The Orville (T) and Rick & Morty (R) contains 100
- The Orville set contains 60
- The intersection contains 35

$$100 = 60 + |R| - 35$$

35

Using the Formula…

- Union of The Orville (T) and Rick & Morty (R) contains 100
- The Orville set contains 60
- The intersection contains 35

$$\begin{aligned} |R| &= 100 - 60 + 35 \\ &= 75 \end{aligned}$$

36

Power Series

All the combinations

37

Power Series

- A *power set* of a set S is a set of all the subsets of S
- This also, obviously, contains the null set
- The notation for the power set S is *P*(S)

38

Power Set Example

```
G = {a, b}

P(G) = { ø, {a}, {b}, {a,b} }
```


39

Power Set Example 2

```
H = {a, b, c}

P(H) = { ø, {a}, {b}, {c},
         {a,b}, {a,c}, {b,c}
         {a,b,c} }
```


40

Power Set Example 3

```
I = {a, b, c, d}

P(I) = {  ø,
         {a}, {b}, {c}, {d}
         {a,b}, {a,c}, {a,d},
         {b,c}, {b,d}, {c,d},
         {a,b,c}, {a,b,d}, {b,c,d}, {a,c,d}
         {a,b,c,d} }
```


41

Cardinality of Power Sets

- What is the cardinality of power sets?
- Is it possible to determine | P(S) | if we know | S |
- This will be important later…

42

Let's Look at the Examples

```
G = {a, b}

P(G) = { ø, {a}, {b}, {a,b} }

|G|    = 2
|P(G)| = 4
```


43

Power Set Example 2

```
H = {a, b, c}

P(H) = {  ø, {a}, {b}, {c},
          {a,b}, {a,c}, {b,c}
          {a,b,c} }

|H|    = 3
|P(H)| = 8
```


44

Power Set Example 3

```
I = {a, b, c, d}

P(I) = {  ø, {a}, {b}, {c}, {d}
          {a,b}, {a,c}, {a,d},
          {b,c}, {b,d}, {c,d},
          {a,b,c}, {a,b,d}, {b,c,d}, {a,c,d}
          {a,b,c,d} }

|I|    = 4
|P(I)| = 16
```


45

Cardinality of Power Set

- The cardinality of a power set is 2^n where n is the cardinality of the original set
- This is used in statistics… covered later

$$|P(S)| = 2^{|S|}$$

46

Partitions

Cutting a Set Into Pieces

47

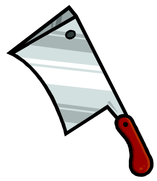
Partitions

- A *partition* of a set A is a collection of non-empty disjoint sets whose union is A
- So, it is like the set A was "chopped", cleanly, into subsets

48

Requirements

- Each subset must be mutually exclusive
- … unless they are identical *(because duplicates don't count in sets)*



49

Partition Example

- The following is a valid partition of the set {1, 2, 3, … 9}

```
{ {1}, {2,3,5,7}, {4,6}, {8,9} }
```


50

Partition Example

- The following is a partition of N.

```
N = { {1}, {2,3}, {4,5,6}, ... }
```


51

Partition Examples

For the set {1, 2, 3, 4}…

Set	Partition?
`{ {1}, {2}, {3}, {4} }`	**Yes**
`{ {1,2}, {1,2}, {3,4} }`	**Yes.** {1,2} is duplicate
`{ {1,2,3}, {2,4} }`	**No.**

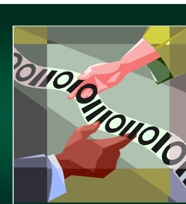
52

Set Theory in Computer Science

Part 3

1

Binary Numbers



Bit of This and a Bit of That

2

What is a Number?

- We use the Hindu-Arabic Number System
  - positional grouping system
  - each position is a power of 10
- Binary numbers
  - based on the same system
  - powers of **2** rather than 10
  - each digit is in the set { 0, 1 }

3

Base 10 Number

The number 1783 is ...

10^4	10^3	10^2	10^1	10^0
10000	1000	100	10	1
0	**1**	**7**	**8**	**3**

1000 + 700 + 80 + 3 = 1783

4

Binary Number Example

The number 0100 1010 is ...

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	**1**	**0**	**0**	**1**	**0**	**1**	**0**

64 + 8 + 2 = 74

5

Binary Number Example

The number 1101 1011 is ...

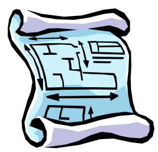
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
1	**1**	**0**	**1**	**1**	**0**	**1**	**1**

128 + 64 + 16 + 8 + 2 + 1 = 219

6

Numbers are Tuples

- In Hindu-Arabic system, the order of the symbols is important – so they are tuples
- e.g. 123 ≠ 321
- Other number styles use sets – i.e. the ancient Egyptian system



7

Looking at Numbers

- Numbers are tuples 1947 ≠ 1974
- Members of the decimals number are also members of the set {0, 1, 2, … 9}

```
1947 → (1,9,4,7)
```


8

Looking at Binary Numbers

- Binary numbers are tuples 10010100 ≠ 11100000
- Members of the binary number are also members of the set {0, 1}

```
10100111 → (1,0,1,0,0,1,1,1)
```


9

Looking at Binary Numbers

- So, for a binary number B, all $x \in B$ holds the following: $x \in \{0, 1\}$

```
10100111 → (1,0,1,0,0,1,1,1)
```


10

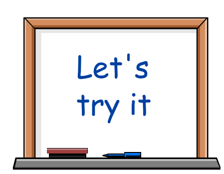
So….

```
{1776, 1846, 1947} →

{ (1,7,7,6), (1,8,4,6)
  (1,9,4,7) }
```

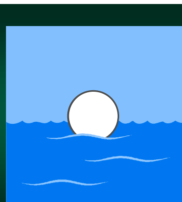

11

Let's Make a Set-Based System

- We are mostly used to tuple-based number systems
- But, for most of history, people used sets
- Let's create one



12



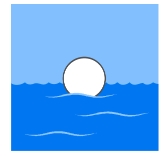
Floating Point Numbers

Real numbers are *real* complex

13

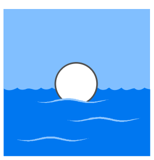
Floating Point Numbers

- Often, programs need to perform mathematics on *real* numbers
- *Floating point numbers* are used to represent quantities that cannot be represented by integers



14

Why use them?



- Regular binary numbers can only store whole positive and negative values
- Many numbers outside the range representable within the system's bit width (too large/small)

15

IEEE 754

- Practically modern computers use the *IEEE 754 Standard* to store floating-point numbers
- Represent by a mantissa and an exponent
  - similar to scientific notation
  - the value of a number is: $mantissa \times 2^{exponent}$
  - uses signed magnitude

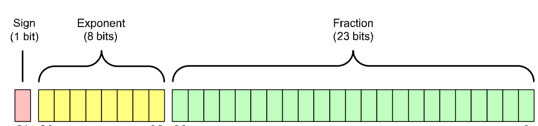
16

IEEE 754

- Comes in three forms:
  - single-precision: 32-bit
  - double-precision: 64-bit
  - quad-precision: 128-bit
- Also supports special values:
  - negative and positive *infinity*
  - and "not a number" for errors (e.g. 1/0)

17

IEEE 754 Single Precision (32 bit)



18

Fractional Field

- The fraction field number that represents part of the mantissa
- If a number is in proper scientific notation…
  - it always has a single digit before the decimal place
  - for decimal numbers, this is 1..9 (never zero)
  - for base-2 numbers, it is always 1

19

Fractional Field

- So, do we need to store the leading 1? It will always be a 1
- The faction field, therefore…
  - only represents the fractional portion of a binary number
  - the integer portion is assumed to be 1
  - this increases the number of significant digits that can be represented (by not wasting a bit)

20

Exponent Field

- The exponent field supports negative and positive values but does not use sign-magnitude or 2's complement
- Uses a "biased" integer representation
  - fixed value is added to the exponent before storing it
  - when interpreting the stored data, this fixed value is then subtracted

21

Exponent Field

- Bias is different depending on precision
  - single precision: 127
  - double precision: 1023
  - quad precision: 16383
- For example, for single precision…
  - exponent of 12 stored as (+12 + 127) → 139
  - exponent of -56 stored as (-56 + 127) → 71

22

Interpretation: Normal Case

- Exponent Field: not all 0's or all 1's
- Fraction Field: Any

$$\pm\ (1.\mathit{fraction}) \times 2^{(\text{exponent} - \text{bias})}$$

23

Interpretation: Zero

- Exponent Field: all 0's
- Fraction Field: all 0's

0

24

Interpretation: Tiny Numbers

- Exponent Field: all 0's
- Fraction Field: Any

$$\pm\ (0.\mathit{fraction}) \times 2^{(1 - \text{bias})}$$

25

Interpretation: Infinity

- Exponent Field: All 1's
- Fraction Field: 0

± ***infinity***

26

Interpretation: Invalid Numbers

- Exponent Field: All 1's
- Fraction Field: Not 0

Not a number (NaN)

27

Interpretation: Invalid Numbers

NaN → **1 / 0**

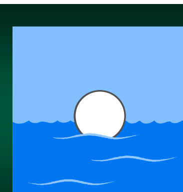
Naan →

28

More Single-Precision Examples

```
Zero:
0 00000000 00000000000000000000000

Positive Infinity:
0 11111111 00000000000000000000000

Negative Infinity:
1 11111111 00000000000000000000000
```


29



Let's Encode Some Numbers!

This is actually fun!

30

Something Else About Numbers…

The number 36.74 is ...

10^2	10^1	10^0	10^{-1}	10^{-2}
100	10	1	1/10	1/100
0	3	6	7	4

= (3×10) + (6×1) + 7/10 + 4/100

31

Binary Fractions!

The number **101.011** is ...

2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
16	8	4	2	1	1/2	1/4	1/8
0	0	1	0	1	0	1	1

= 4 + 1 + 1/4 + 1/8 = 5.375

32

Let's Encode 14.25 (32 bit)

- First, we need to convert 14.25 to its binary equivalent
- So, we need to calculate the integer part and fraction part of the number
- Everything after the decimal is a fraction
  - 0.25 is actually 25 / 100
  - we need to find the base 2 equivalent (1/4)

33

Step 1: Convert to binary

```
14 → 1110

0.25 → 1/4 → 0.01

Hence:
14.25 → 1110.01
```

binary 01 / 100

34

Step 2: Scientific Notation

- IEEE stores the data in scientific notation
- So we move the "binary point" over

1110.01 → 1.11001 × 2^3

35

Step 2: Scientific Notation

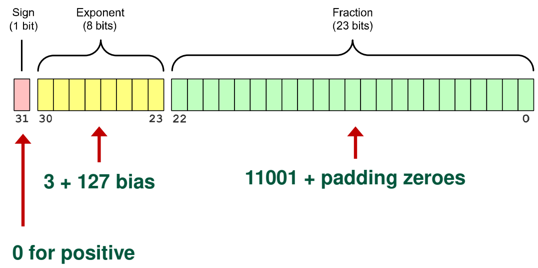
- In binary scientific notation, the leading digit is always going to be 1
- Why store it? IEEE doesn't.
- Only data after the point is encoded

Fraction

1.11001 × 2^3 → (1 + .11001) × 2^3

36

Step 3: Encode



Result: 14.25 (32 bit)

- The following is the encoded version of 14.25
- The rules are similar for double-precision

```
0 10000010 11001000000000000000000
```


Result: 14.25 (32 bit)

- We can also convert this into bytes
- Note: the floating point fields don't really "fit" cleanly into actual bytes

```
01000001 01100100 00000000 00000000
   41       64       00       00
```


Example 2: Encode 13.75 (32 bit)

- First, we need to convert 13.75 to its binary equivalent
- So, we need to calculate the integer part and fraction part of the number
- Everything after the decimal is a fraction
  - 0.75 is actually 75 / 100
  - we need to find the base 2 equivalent (3/4)

Step 1: Convert to binary

```
13 → 1101

0.75 → 3/4 → 0.11

Hence:
13.75 → 1101.11
```

binary 11 / 100

Step 2: Scientific Notation

- IEEE stores the data in scientific notation
- So we move the "binary point" over

$$1101.11 \rightarrow 1.10111 \times 2^3$$

Step 2: Scientific Notation

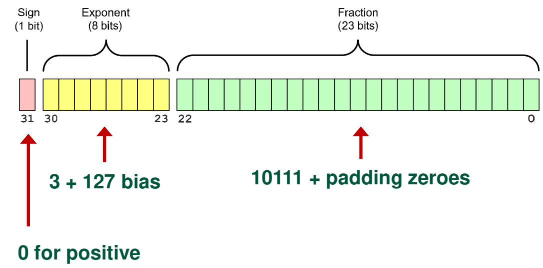
- In binary scientific notation, the leading digit is always going to be 1
- Why store it? IEEE doesn't.
- Only data after the point is encoded

Fraction

```
1.10111 × 2³ → (1 + .10111) × 2³
```


43

Step 3: Encode



44

Result: 13.75 (32 bit)

- The following is the encoded version of 13.75
- The rules are similar for double-precision

```
0 10000010 10111000000000000000000
```


45

Result: 13.75 (32 bit)

- We can also convert this into bytes
- Note: the floating point fields don't really "fit" cleanly into actual bytes

```
01000001 01011100 00000000 00000000
   41       5C       00       00
```


46

Tuples and Floats

Its all set theory, folks!

47

Floats Are Tuples

- Just like regular binary numbers, floating-point numbers of tuples
- They consist of three fields making them 3-tuples

```
(sign, exponent, fraction)
```


48

Encoding of 14.25

- Sign is a 1-tuple
- Exponent is a 8-tuple
- Fraction is a 23-tuple

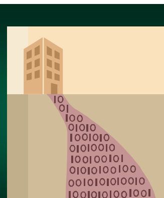
```
0 10000010 11001000000000000000000
```


49

Set Notation of 14.25

```
(  (0),
   (1,0,0,0,0,0,1,0),
   (1,1,0,0,1,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0)   )
```


50



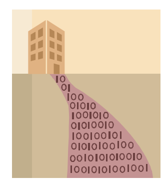
Bit Vectors

Sets and Bits

51

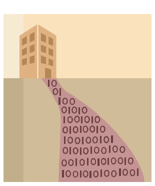
Bit Vectors

- A *bit vector* is a way to store countable sets using bits
- Also known as a *bit array*, *bit set*, and *bit map*
- Compact format that can perform a set operations with a single operation *(fast!)*



52

Bit Vectors



- Each object in the universe is given a single bit in the bit array
- If the $x \in A$, then the bit is 1, otherwise 0
- Order is important, so this is a tuple approach

53

Example 1

- U = { fry, bender, farnsworth, leela, zoidberg}
- A = { fry, bender, leela }

```
U = 11111
A = 11010
```

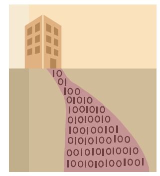

54

Example 2

- U = { 2, 3, 5, 7, 11, 13, 17, 19}
- A = { 3, 5, 11, 19 }

```
U = 11111111
A = 01101001
```


Why this is useful



- Computers can easily perform and & or operations on bytes (or multiple bytes)
- This means set operations can be performed amazingly fast

Let's look at the definitions again…

- The definitions of union and intersection are nearly identical
- The relationship between the elements is defined using an *and* or *or*

$A \cup B = \{ x \mid x \in A \text{ or } x \in B \}$
$A \cap B = \{ x \mid x \in A \text{ and } x \in B \}$

Let's look at the definitions again…

- We can apply a *bit-wise-and* & a *bit-wise-or* to our bit array
- It will apply the operation to each of the bits in matching columns

$A \cup B = \{ x \mid x \in A \text{ or } x \in B \}$
$A \cap B = \{ x \mid x \in A \text{ and } x \in B \}$

Let's look at the definitions again…

- So, each bit in A will be compared to its matching bit in B
- Bit match can do sets!

$A \cup B = \{ x \mid x \in A \text{ or } x \in B \}$
$A \cap B = \{ x \mid x \in A \text{ and } x \in B \}$

Example: Union (using or)

```
U = {a,b,c,d,e,f,g}
A = {b,c,d} = 0111000
B = {d,e,f} = 0001110

    0111000
 or 0001110
    0111110 = {b,c,d,e,f}
```

Example: Intersection (using and)

```
U = {a,b,c,d,e,f,g}
A = {b,c,d} = 0111000
B = {d,e,f} = 0001110

    0111000
and 0001110
    0001000 = {d}
```


61

Complement

- How do we do a complement of a set A?
- We must flip all the bits from 1 to 0, and 0 to 1
- We can use a *binary-not* or the *XOR* operation

$$A' = \{ x \mid x \notin A \}$$

62

Java/C Code

&& and || are Boolean.
These are bit-wise.

```
Intersection :  a & b
Union        :  a | b
Complement   :  ~a
```

The tilde ~ is a bitwise not.

63

Exclusion

- Finally, how do we do set difference?
- The "subtract" operator will not work
- Let's look at the definition a bit more closely

$$A \setminus B = \{ x \mid x \in A \text{ and } x \notin B \}$$

64

Exclusion

- It's essentially the definition of *intersection*
- Except, the second operand is the definition of *complement*.

$$A \setminus B = \{ x \mid x \in A \text{ and } x \notin B \}$$

65

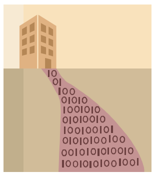
Java/C Code

Just complement the second operand

```
Exclusion :  a & ~b
```


66

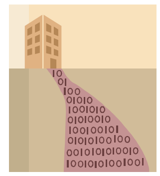
Limits of Bit Vectors



- Bit vectors, while useful, do have some notable limitations
- They only work on finite, countable sets
- For all other cases, you will have to work use a more advanced ADT

67

CSC 130 is waiting for you!



- For most cases, a very sophisticated list or tree can be used
- You will need to know:
  - lists / trees
  - sorting
  - binary-searches
  - Big-O

68

Relations

Part 4

1

Cross Products

Databases use this…

2

Cross Products

- Sets can be multiplied, which will result in a set of tuples
- Well, a set of ordered pairs, to be more specific
- Cross products are important in databases and counting *(to name a few)*

3

Products

- A *cross product* is a set of ordered pairs
- Note: Unlike multiplication, the order of the operands is important

$$A \times B = \{ (x, y) \mid x \in A \text{ and } y \in B \}$$

4

Example Product

```
Given:
  A = {small, big}
  B = {cat, dog}

A × B = { (small, cat), (small, dog),
          (big, cat), (big, dog) }
```


5

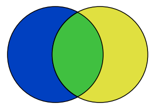
Binary Relations

How Stuff Compares to Stuff

6

Relations

- A *binary relation* is a stated fact between on two objects
- A "fact" is called a *predicate*
- Evaluates to true or false
- These are the foundation of most programming tasks



7

Example Relations

- "x is bigger than y"
- "x lives less than 50 miles from y"
- "x ≤ y"
- "x and y are siblings"
- "x has a y"

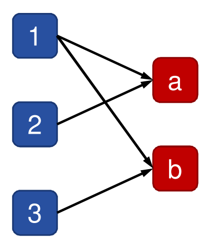
8

Relations

- A *binary relation* from A to B is a subset of the cross-product $A \times B$
- A relation from A to B is a set of ordered pairs (a, b) where $a \in A$ and $b \in B$
- We can use the shorthand notation of: a R b to denote that $(a, b) \in R$

9

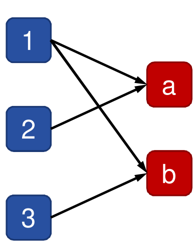
Relationship Chart



	a	b
1	✓	✓
2	✓	
3		✓

10

Relationship Chart




```
{ (1, a), (1, b),
  (2, a), (3, b) }
```


11

Example: Capitols

- A is a set of all cities in the World
- B is a set of all states in the World
- The relation *a* R *b* specifies that *a* is the capitol of *b*

12

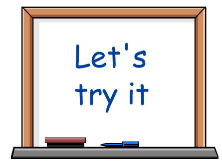
Example: Capitol Members

- (London, Britain)
- (Sacramento, California)
- (Madrid, Spain)
- (Tokyo, Japan)
- (New Delhi, India)
- (Albany, New York)

13

Let's Draw One!

- Let's draw a relationship graph
- It will be between students and shows/movies they enjoy



14

Relation Domain

- The *domain* of a relation is a set of all the first elements of each tuple
- So, it is the elements that the make up the left-hand side of *a* R *b*

$\{ a \mid (a, b) \in R \text{ for some } b \}$

15

Relation Range

- The *range* of a relation is a set of all second elements from each tuple
- So, it is the elements that the make up the right-hand side of *a* R *b*

$\{ b \mid (a, b) \in R \text{ for some } a \}$

16

Example

```
R = { (1,1), (1,2), (1,3),
      (1,4), (2,4), (4,4) }

Domain of R = { 1, 2, 4 }

Range of R = { 1, 2, 3, 4 }
```


17

Inverse Relation

- The *inverse* of a relation swaps first and last element for each tuple
- The number of elements are the same, but the range and domain are reversed

$R^{-1} = \{ (b, a) \mid (a, b) \in R \}$

18

Inverse Example

```
R = { (1,1), (1,2), (1,3),
      (1,4), (2,4), (4,4) }

R-1 = { (1,1), (2,1), (3,1),
        (4,1), (4,2), (4,4) }
```


19

Relations can be infinitely large

- On a finite set, relations are quite simple…
- For a set with n elements, the maximum number of relations is simply $n \times n = n^2$
- However, many relations are defined over an infinite set – e.g. all integers, reals, etc…

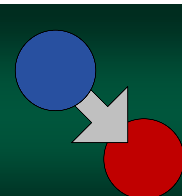
20

Representing Relations

- We can represent a relation using set notation
- However, some are required to be denoted using set builder notation

```
R1 = { (a, b) | a is bigger than b }
R2 = { (a, b) | a ≤ b }
```


21



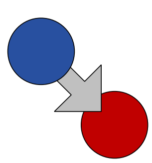
Types of Relations

How a Set Sees Itself

22

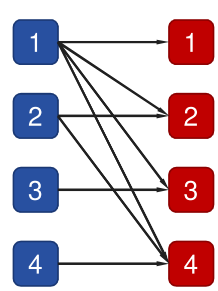
"Relation On"

- Some relations of a set A are upon itself
- In other words, each object in the related to the same "type" of object
- This is called a *relation on* A
- …and it is a important to examine its properties



23

Example Relationship Chart



	1	2	3	4
1	✓	✓	✓	✓
2		✓		✓
3			✓	
4				✓

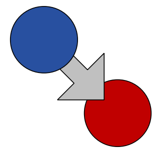
24

Example Relation Chart

- The previous chart represents when *a* divides *b*
- In other words, *a* times some integer equals the value *b*
- So, R = { (1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 4), (3, 3), (4, 4) }

25

Reflexive Relations

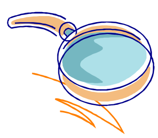


- A *reflexive* relationship means that there is *a*R*a* for every *a*
- Basically, everything has to be related to itself
- *Every element, in the domain, must be related to itself*

26

To Determine Reflexive…

- Look for some $a \in A$ where there isn't a *a*R*a*
- If found, not reflexive
- Otherwise reflexive



27

Reflexive Example

```
Relation on set {1, 2, 3, 4}

R = { (1,1), (1,4), (2,2),
      (2,3), (3,3), (4,4) }
```

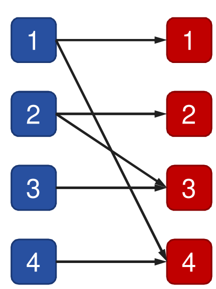

28

Reflexive Example

```
Relation on set {1, 2, 3, 4}

R = { (1,1), (1,4), (2,2),
      (2,3), (3,3), (4,4) }
```


29

Reflexive Example



	1	2	3	4
1	✓			✓
2		✓	✓	
3			✓	
4				✓

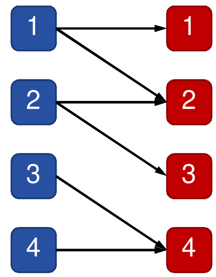
30

Nonreflexive Example

```
Relation on set {1, 2, 3, 4}

R = { (1,1), (1,4), (2,2),
      (2,3), (3,1), (4,4) }
```


31

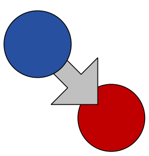
Nonreflexive Example



	1	2	3	4
1	✓	✓		
2		✓	✓	
3				✓
4				✓

32

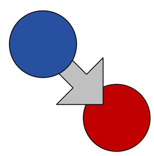
Symmetric Relations



- A *symmetric* relationship means that for every *a*R*b* there is a *b*R*a*
- So, if (a, b) exists in the relation, so must (b, a)

33

Symmetric Relations

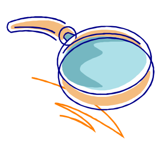


- Note: Unlike the definition for reflexive, not every element in the domain needs to exist
- If a relation is not symmetric, it is *antisymmetric*

34

To Determine Symmetric…

- Look for an *a* and *b* where there is a *a*R*b* but no *b*R*a*
- If found, nonsymmetric
- Otherwise symmetric



35

Symmetric Example

```
Relation on set {1, 2, 3, 4}

R = { (1,1), (1,2), (2,1)
      (2,4), (3,3), (4,2) }
```

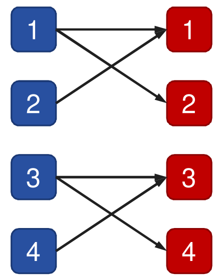

36

Symmetric Example

```
Relation on set {1, 2, 3, 4}

R = { (1,1), (1,2), (2,1)
      (2,4), (3,3), (4,2) }
```


37

Symmetric Example



	1	2	3	4
1	✓	✓		
2	✓			✓
3			✓	
4		✓		

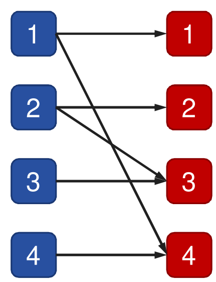
38

Nonsymmetric Example

```
Relation on set {1, 2, 3, 4}

R = { (1,1), (1,4), (2,2),
      (2,3), (3,3), (4,4) }
```


39

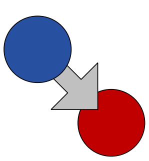
Nonsymmetric Example



	1	2	3	4
1	✓			✓
2		✓	✓	
3			✓	
4				✓

40

Transitive Relations

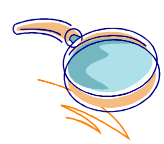


- A *transitive* relationship means that for every *a*R*b* and *b*R*c* that also *a*R*c*
- So, if (a, b) and (b, c) exists in the relation, so must (a, c)

41

To Determine Transitive…

- Look for an *a*, *b*, *c* where there is a *a*R*b* and *b*R*c* but no *a*R*c*
- If found, non transitive
- Otherwise transitive



42

Transitive Example

```
Relation on set {1, 2, 3, 4}

R = { (2,1), (3,1), (3,2),
      (4,1), (4,2), (4,3) }
```


43

Transitive Example

```
Relation on set {1, 2, 3, 4}

R = { (2,1), (3,1), (3,2),
      (4,1), (4,2), (4,3) }
```

Starting with (4,3)

44

Transitive Example

```
Relation on set {1, 2, 3, 4}

R = { (2,1), (3,1), (3,2),
      (4,1), (4,2), (4,3) }
```

Starting with (3,2)

45

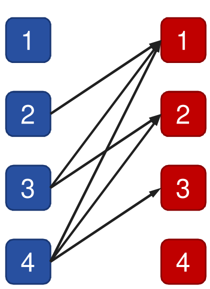
Transitive Example

```
Relation on set {1, 2, 3, 4}

R = { (2,1), (3,1), (3,2),
      (4,1), (4,2), (4,3) }
```

Starting (again) with (4,3)

46

Transitive Example

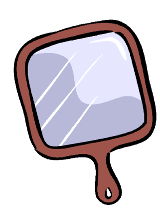


	1	2	3	4
1				
2	✓			
3	✓	✓		
4	✓	✓	✓	

47

Equivalence Relations

- If a relation has all three properties:
  - reflexive
  - symmetric
  - transitive
- Then, and only then, it is an *equivalence*



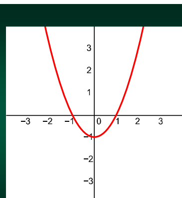
48

Example Table

Relation	Reflexive	Symmetric	Transitive	Equivalent
≤				
⊂				
Perpendicular lines				
Parallel lines				

Example Table

Relation	Reflexive	Symmetric	Transitive	Equivalent
≤	✓	✗	✓	✗
⊂	✗	✗	✓	✗
Perpendicular lines	✗	✓	✗	✗
Parallel lines	✓	✓	✓	✓

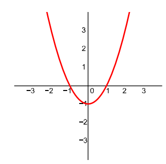


Functions

Math Friendly Relations

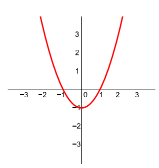
Functions

- We have all seen functions – which take inputs and produce output
- Example: $f(x) = x^2$
  - $f(1) = 1$
  - $f(2) = 4$
  - $f(3) = 9$ ...

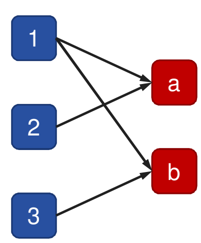


Functions

- Sets give a way to document "types" in mathematical functions
- A *function* from set X to set Y is a mapping from <u>each</u> element in set X to elements in set Y



Relationship Review




```
{ (1, a), (1, b),
  (2, a), (3, b) }
```

Function Attributes

- Function Rules:
  - must be defined for *every element in domain*
  - each value in domain *maps to one element*
- Notice that a function defines a set of ordered pairs: e.g. (1,1) (2,4) (3,9) ...
- We can therefore think of a function as a special kind of relation.

Relations vs. Functions

- Each domain element, in a relation, can specify *many* relationships
- While, each element in a function domain only specifies *one* relationship
- So….
  - every function is a relation
  - but not every relation is a function

Definition of a Function

Let f be a relation from A → B

f is a function if and only if:

each a ∈ A appears exactly once in an ordered pair (a, b) ∈ f for some b

Function Signature

- We will restrict a functions inputs and outputs by giving a "signature" for it
- f is the function name

```
f: N → N
```


Function Signature

- The first N is the function domain
- The second N is the function range (codomain)

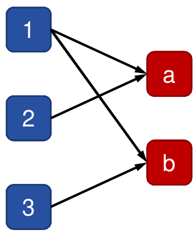
```
f: N → N
```


Domain and Range Definitions

- The domain and range of f is defined exactly as we saw for relations
- Which is not surprising given what a function really is

domain(f) = { x | (x, y) ∈ f for some y }
range(f) = { y | (x, y) ∈ f for some x }

Let's Look At This Again….




```
{ (1, a), (1, b),
  (2, a), (3, b) }
```

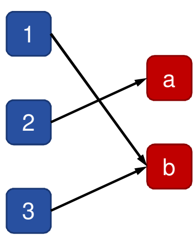

61

Relations vs. Functions

- Not that in the example (with 1,2,3 and a, b) that some elements in A had multiple values in B
- In a function, each member in A maps to exactly one value in B
- So, that relation was not a function!

62

Function Example




```
{ (1, b), (2, a),
  (3, b) }
```


63

Examples

- For the following examples, let each example be defined as a relation from A to B
- Domain and range (codomain) are defined as:

```
A = {1, 2, 3}
B = {x, y, z}
```


64

Is This a Function?

```
Let f = { (1,x), (2,y) }

No, the domain value 3 is missing
as a first ordered-pair element
```


65

Is This a Function?

```
Let g = { (1,x), (2,y),
          (3,z), (1,y) }

No, the domain element 1 is listed
twice.
```

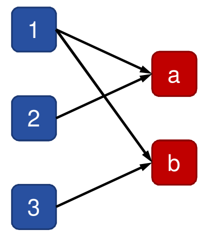

66

Is This a Function?

```
Let h = { (1,x), (2,y), (3,x) }

Yes, each domain element of A is
the first element once.

(There is no restriction on
 the second element)
```


67

Is This a Function?

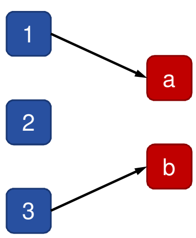


No!

(1,a) and (1,b) both exist

68

Is This a Function?

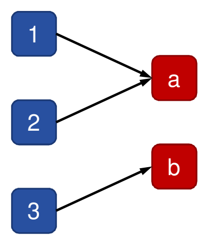


No!

The domain element 2 is not mapped to a or b

69

Is This a Function?



Yes!

70

Function Definitions

The Mapping of Sets

71

Function Definitions

- Functions are usually defined using a formula
- You should be able to tell that these match a Java method definition – header and body

```
f: Z → Z
f(x) = x * x
```


72

Function Definitions

- First part tells us that f maps every integer to an integer
- Second part tells us f(x) and x^2 are the same thing

```
f: Z → Z
f(x) = x * x
```


73

Example

- In the following, is *g* a valid function?
- R is a set of reals
- √ is the square root function

```
g: R → R
g(x) = √x
```


74

Example

- **No.**
- Not every element of R maps to something in R
- For example, g(-1) $\notin$ R

```
g: R → R
g(x) = √x
```


75

Manipulating Relations

How a Set Sees Itself

76

Manipulating Relations

- Because relations are representable as sets, we can use set notation to define them
- We can also use set notation to manipulate them

77

Example

```
A = { (1,1), (2,2), (3,3) }
B = { (1,1), (2,4), (3,9) }
```


78

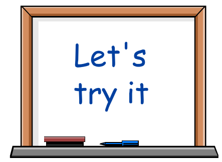
Example

```
A ∪ B =   { (1,1), (2,2), (2,4),
            (3,3), (3,9) }

A ∩ B =   { (1,1) }

A \ B =   { (2,2), (3,3) }

B \ A =   { (2,4), (3,9) }
```


79

Let's Examine Some…

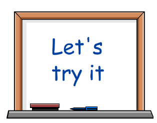
- Let's use students to create two relations
- Classes you plan to take
- Classes you might/did enjoy



80

Let's Examine Some…

- Let's examine two relations over a simple set {1, 2, 3} using set operators
- We can check if it is:
  - reflexive
  - symmetric
  - transitive



81

Let's Examine Some…

```
A = {1,2,3}: R, S relations.

R = { (1,1), (1,2), (2,2),
      (2,3), (3,1), (3,3)}

S = { (1,1), (1,2), (1,3),
      (2,1), (2,3), (3,2) }
```


82

Closures

Making a relation "complete"

83

Closure

- *Closure* of relation R is the smallest set (when unioned) gives R the desired property
- So, the closure of R is R ∪ C, where C is the smallest set giving R ∪ C the desired property

84

Some Examples

- For the following examples, the relation is over the set {1, 2, 3, 4}
- The slides will show how to make the closures for reflexive, symmetric, and (the hard one) transitive

85

Example Reflexive Closure

```
R = { (1,2), (2,3), (3,4) }

C = { (1,1), (2,2), (3,3), (4,4) }
```

Missing (1,1) (2,2), (3,3) and (4,4)

86

Example Reflexive Closure

```
R ∪ C = { (1,2), (2,3), (3,4),
          (1,1), (2,2), (3,3),
          (4,4) }
```


87

Example Symmetric Closure

```
R = { (1,2), (2,3), (3,4) }

C = { (2,1), (3,2), (4,3) }
```


88

Example Symmetric Closure

```
R ∪ C = { (1,2), (2,3), (3,4),
          (2,1), (3,2), (4,3) }
```


89

Example Transitive Closure (1 of 3)

```
R = { (1,2), (2,3), (3,4) }

C = { (1,3)                    }
```

Added due to (1,2) and (2,3)

90

Example Transitive Closure (2 of 3)

```
R = { (1,2), (2,3), (3,4) }

C = { (1,3), (2,4)        }
```

Added due to (2,3) and (3,4)

Example Transitive Closure (3 of 3)

```
R = { (1,2), (2,3), (3,4) }

C = { (1,3), (2,4), (1,4) }
```

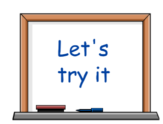
Had to add after we added (2,4) since R contains (1,2)

Example Transitive Closure

```
R ∪ C = { (1,2), (2,3), (3,4),
          (1,3), (2,4), (1,4) }
```


Let's Try Some

- Set is over {1, 2, 3, 4, 5}
- R = { (1,2), (2,3), (3,5), (4,1) }
- What are the closures for this relation?



Composition

The Inception of Functions

Composition

- *Composition* of two functions means the output of one function is used as the input as another
- This is very common in programming – you use the result of one expression as input to another

Notation

- Notation for composition is straight forward – it simply consists of a empty circle operator
- Sometimes the (x) is put in front of the first function, but this is not always the case

```
f ∘ g(x)   ≡   f(g(x))
```


97

Composition Example

```
f(x)  = x + 4
g(x)  = x²

f ∘ g(z)  = f(g(z))
          = f(z²)
          = z² + 4
```


98

Composition Example 2

```
f(x)  = x + 4
g(x)  = x²

g ∘ f(z)  = g(f(z))
          = g(z + 4)
          = z² + 8z + 16
```


99

Composite Example

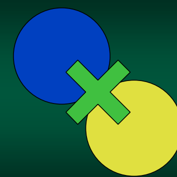
```
R = { (1,2), (3,1), (5,3) }
S = { (2,3), (2,6), (3,9) }

R ∘ S = { (1,3), (1,6), (5,9) }
```


100

Relations in Computer Science

Part 5

1



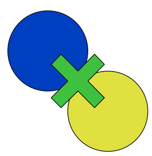
Cross Products & Databases

SQL <u>is</u> set notation

2

Cross Products & Databases

- We are in the "Information Age" where knowledge is now computerized
- Information is stored in databases
- These systems are based on tuples and sets



3

Fields

- *Fields* contain the smallest unit of data
  - e.g. Number, Text
  - So, each can be seen as a tuple (it can be a set, but rarely so)
- Each field has a unique field name
  - Name
  - Age
  - etc....

4

Records

- A *record* is a set of data fields
  - represents a logical group of data
  - these include related numbers, text, images, etc...
- Examples
  - Course: department, number, section
  - Student: name, age, class
  - Computer: brand, speed, cost, etc...

5

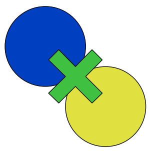
Database Example

First	Last	Major	Greek
Peter	Griffen	Und	Tappa Kegga Bru
Joe	Gunchy	CSc	Cuppa Kappa Chino
Rick	Sanchez	Sci	elta Phart
Eric	Cartman	Bus	Eta Lotta Pi

Record

6

Relationships & Cardinality



- *Relational Databases* allow the user to query multiple related tables
- Related tables are *joined* which performs a cross product on two tables
- Restrictions are used to eliminate unneeded records

7

Locating Specific Data

- A *query language* is used to:
  - locate information
  - sort records
  - change data in records
- Examples:
  - SQL (Structured Query Language)
  - Natural language queries – not used often

8

SQL Inner Join

```
SELECT Student.name,
       Course.grade
FROM Student
INNER JOIN Course
ON Student.sid = Course.sid
WHERE Course.department = "csc"
```


9

SQL Inner Join - Sets (simplified)

$$\{\ (s_{name},\ c_{grade})\ |\ s \in \text{Student and } c \in \text{Course and } s_{sid} = c_{sid} \text{ and } c_{department} = \text{"csc"}\ \}$$

10

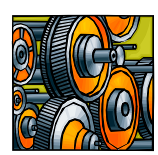
Abstract Data Types

What *int* really means

11

Application of Sets

- An *abstract data type (ADT)* defines:
  - a set of possible values and
  - operations (functions) that an be performed on those values
- These are the basis for all classes and data structures in programming languages



12

Integer Example

- In the code below, *int* is an ADT (found in most programming languages)
- It declares a variable *n* of type *int*
- n represents a value from int's set of values

```
int n;
```

Spring 2020 Sacramento State - Cook - CSc 28 13

13

Domain of 'int'

- int is defined (normally) as 32-bit
- So, the set is { -2^{31}, …, (2^{31} – 1) }
- Also note: int $\subset$ Z
  - this means that it cannot store any integer
  - … just within a small range subset of Z

Spring 2020 Sacramento State - Cook - CSc 28 14

14

What Java's notation means

- When you declare a variable, you are stating that it is a member of a set
- In the example below, the two statements mean same thing: set notation vs. Java notation

```
n ∈ int
int n;
```

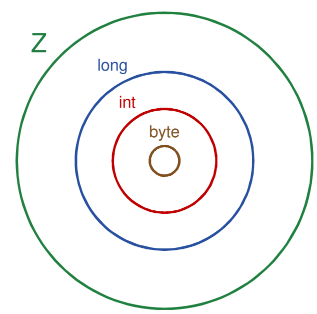
Set notation

Java notation

Spring 2020 Sacramento State - Cook - CSc 28 15

15

Java Integer Sets



2/18/2020 Sacramento State - Cook - CSc 28 - Spring 2018 16

16

Possible Set Violation

```
void test(int x)
{
    ...
}

int main()
{
    long n;
    test(n);
}
```

long is a superset of int

int may not able be able to store n

2/18/2020 Sacramento State - Cook - CSc 28 - Spring 2018 17

17

No Set Violation

```
void test(long x)
{
    ...
}

int main()
{
    int n;
    test(n);
}
```

2/18/2020 Sacramento State - Cook - CSc 28 - Spring 2018 18

18

Operations

- ADT also defines that n can be manipulated by via functions +, -, ×, ÷
- Sometimes, programming languages are different (division for example)

```
÷ : Z, Z → Z  in Java, C++, C#
÷ : Z, Z → R  in Visual Basic
```


19

Functions

Notation varies, but logic the same

20

Functions

- Many programming languages support customized functions
- The format often mirrors the notation used in discrete mathematics

21

C Family

- C programming language family includes: C, C++, Java, and C#
- The notation is very terse, but includes all the same information

```
name : int → int
int name(int n)
```

Discrete math

22

Visual Basic

- Visual Basic evolved from the original BASIC programming language
- The notation is far more verbose

```
name : Integer → Integer
Function name(n As Integer) As Integer
```


23

Pascal

- Pascal was very popular in the 1980's and 90's
- Created many concepts that were integrated into other languages

```
name : integer → integer
function name(n : integer) : integer
```


24

Fortran

- Fortran was the first third-generation programming language
- It has evolved over 50 years

```
name : integer → integer

function name(n) result(x)
    integer, intent(in) :: n
    integer :: x
```


25

Swift

- Swift was created by Apple to replace older Objective-C
- Influenced by multiple languages

```
name : Int → Int

func name(n : Int) -> Int
```


26

Swift

- Swift was created by Apple to replace older Objective-C
- Influenced by multiple languages

```
name : Int → Int

func name(n : Int) -> Int
```

An arrow!

OMG! An arrow!

Wow! Correct notation!

27

Boolean Logic

Part 6

1

Logic Statements

Make Mr. Spock Proud

2

Logic Statements

- Logic is used to construct all proofs and computer systems
- A *statement* is any declarative sentence that results in either true or false

3

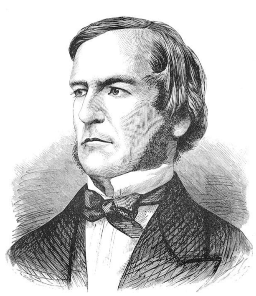
Examples of Statements

- *There are exactly 35 people in this room*
- *Sacramento State is located next to a river*
- *10 + 2 = 11*
- *We have great choices for the 2020 Election*

4

Boolean Logic

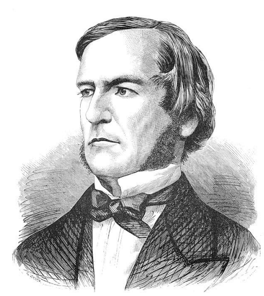
- Discovered by George Boole
- First published in *The Mathematical Analysis of Logic (1847)*



5

Boolean Logic

- Revolutionized logic & proofs and is part of framework of modern of computer science



6

Boolean Operators

- Statements can be combined in *compound statements* using Boolean operators
- After statements are combined, they are still statements
- For example: "p and q"
  - given that p and q are both statements
  - then "p and q" is also a statement

7

Let's Review Boolean Operators

Operator	Name
`p and q`	True only if both p and q are true
`p or q`	True if either p or q true
`not p`	True if p false
`p xor q`	True if p and q are different
`p implies q`	True unless p is true and q is false

8

Logic Notation of Operators

Logic	C Family	Visual Basic
$p \wedge q$	`p && q`	`p and q`
$p \vee q$	`p || q`	`p or q`
$\neg p$	`!p`	`not p`
$p \oplus q$	*none*	`p xor q`

9

Truth Tables

- Truth tables are useful tools for analyzing a large Boolean expression
- The table includes all the possible combinations of True and False for each input into the equation
- This results in 2^n rows where n is the number of inputs

10

Truth Table

p	q	$\neg p$	$p \vee q$	$p \wedge q$	$p \rightarrow q$
T	T	F	T	T	T
T	F	F	T	F	F
F	T	T	T	F	T
F	F	T	F	F	T

11

AND Example

True True

$1 < 2 \wedge 10 < 40$

Result: True

12

AND Example 2

True — False

$1 < 2 \land 12 < 10$

Result: False

13

OR Example

True — True

$1 < 2 \lor 10 < 40$

Result: True

14

OR Example 2

True — False

$5 > 3 \lor 44 < 8$

Result: True

15

OR Example 3

False — False

$49 < 47 \lor 99 < 10$

Result: False

16

NOT Example

True

$\neg(1 < 2)$

Result: False

17

NOT Example

False

$\neg(1 = 2)$

Result: True

18

Examples

$1 < 3 \land 10 < 40$	True
$1 = 3 \land 10 < 40$	False
$\neg (12 \neq 12)$	True
$1 > 3 \lor 30 < 20$	False

19

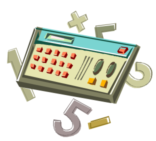
Logical Operator Precedence

Yup, we have that here too!

20

Logical Operator Precedence

- In *propositional logic*, statements can be combined with other statements using logical operators
- So, they can be chained together to form complex logic



21

Algebra: Order of Operations

- Some mathematical operators have a high "precedence" than others
- They are computed first

```
3 + 6 / 3 * 2
```


22

Algebra: Order of Operations

- Knowing the correct order is vital
- For example, what is the result of the expression below?

```
3 + 6 / 3 * 2
```


23

Algebra: Order of Operations

- It is 7
- Divide and multiply are equal (and then done left to right), addition is done last

```
3 + 6 / 3 * 2 = 7
```


24

Many try, many fail

- Many students, using "PEMDAS" , think multiply is done before divide (M is before D)
- … or they just go left to right with no regard to precedence

```
3 + 6 / 3 * 2  =  4    WRONG! F-!
```


25

Standard Precedence Levels

1	¬
2	∧
3	∨ ⊕

Highest Level

Lowest Level

26

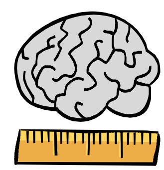
Defining Boolean Logic

"Want to define it?"
"True."

27

Defining Boolean Logic

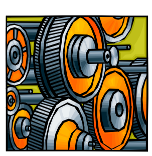
- Let's look at what exactly Boolean logic is in context of data types and functions
- Once we define the Boolean Data type, we can apply it to other systems



28

Functions

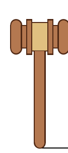
- Also recall functions from earlier
- An *abstract data type* is a set of values and functions on those values
- So, we can define the data type for Boolean values



29

Extending Boolean to Other Types

- The abstract data type for a Boolean Data Type can be written as a 6-tuple



$$B = (S, \vee, \wedge, \neg, \bot, \top)$$

30

Extending Boolean to Other Types

- The first property is a set S contains two elements
- These are: $\bot$ (smaller) and $\top$ (bigger)

$$B = (S, \vee, \wedge, \neg, \bot, \top)$$

31

Extending Boolean to Other Types

- 3 operations on the set S: $\vee$, $\wedge$, $\neg$
- Must follow the 4 primary axioms: Identity, Communitive, Distributive, Complement

$$B = (S, \vee, \wedge, \neg, \bot, \top)$$

32

Defining Boolean Logic

- Boolean Logic is closely related to Set Theory
- So much, in fact, that Boolean Logic can be considered as a special case of sets

{ }

33

Defining Boolean Logic

- We can show that the behavior of Boolean Logic can be created in sets
- It's not surprised that many of the laws for sets work for Boolean Logic

{ }

34

Set Theory & Boolean Logic

- First, we can define True as the Universe
- Remember that in binary logic, it simply *is* or it *isn't*
- So, the Universe means 1 or true

$$T = \mathbf{U}$$
$$F = \varnothing$$

35

Set Theory & Boolean Logic

- The complement of $\mathbf{U}$ is $\varnothing$
- So, naturally, False is represented with $\varnothing$

$$T = \mathbf{U}$$
$$F = \varnothing$$

36

Boolean Operators with Sets

- The Union operator is analogous to the And operator
- Likewise, Intersection is analogous to Or.

$$a \wedge b = A \cap B$$
$$a \vee b = A \cup B$$

37

And-Intersection Comparison

Boolean Logic	Set Theory
T ∧ T = T	U ∩ U = U
T ∧ F = F	U ∩ ∅ = ∅
F ∧ T = F	∅ ∩ U = ∅
F ∧ F = F	∅ ∩ ∅ = ∅

38

Or-Intersection Comparison

Boolean Logic	Set Theory
T ∨ T = T	U ∪ U = U
T ∨ F = T	U ∪ ∅ = U
F ∨ T = T	∅ ∪ U = U
F ∨ F = F	∅ ∪ ∅ = ∅

39

Boolean Not with Complement

- The Boolean Not operator can also be implemented using set theory
- In this case, complement

$$\neg a = A'$$

40

Or-Intersection Comparison

Boolean Logic	Set Theory
¬T = F	U' = ∅
¬F = T	∅' = U

41

The Axioms

- The Axioms are:
  - Identity
  - Communitive
  - Distributive
  - Complement
- The axioms from Set Theory apply to Boolean Logic

42

Tautology & Contradiction

When the logic is quite elementary

43

Tautology & Contradictions

- Some statements are always true or false regardless of the variables used
- If the statement is always true, it is called *tautology*
- If the statement is always false, it is called a *contradiction*

44

Example Tautologies

- The following are examples of tautologies
- The result will always be **true**

$p \lor \neg p$
$p \rightarrow p$
$p = p$

45

Example Contradictions

- The following are examples of contradictions
- The result will always be **false**

$p \land \neg p$
$p \oplus p$

46

Example

- So, what is the truth table for the example below?
- Let's create a truth table

$(\neg p \land q) \land (p \lor \neg q)$

47

Logic Equivalence

Same meaning, different form

48

Logical Equivalence

- *Logical equivalence* is when two different statements are the same
- The truth tables for both statements are identical

49

First Four Axioms

- The first four fundamental axioms were developed by *Edward Huntington* in 1904
- Other rules are derived from these

50

Communitive Law

- Both $\wedge$ and $\vee$ are communitive
- This means the left-hand and right-hand operands can be switched (symmetric relation)

$$a \wedge b \equiv b \wedge a$$
$$a \vee b \equiv b \vee a$$

51

Identity Law

- Some operands will have no effect on the truth table of a statement
- In this case, the statement can be simplified

$$a \wedge \text{true} \equiv a$$
$$a \vee \text{false} \equiv a$$

52

Complement Law

- When a statement is used with its complement (itself negated), it will result in either true or false
- So, it is always as tautology or contradiction

$$a \wedge \neg a \equiv \text{false}$$
$$a \vee \neg a \equiv \text{true}$$

53

Distributive Law

- Math has operators that are *distributive*
- For example: a * (b + c) = (a * b) + (a * c)
- Works for both $\wedge$ and $\vee$

$$a \wedge (b \vee c) \equiv (a \wedge b) \vee (a \wedge c)$$
$$a \vee (b \wedge c) \equiv (a \vee b) \wedge (a \vee c)$$

54

Logical Equivalence

- A number of useful laws can be derived from the first four
- It is vital to remember all of these when solving complex Boolean equations

55

Associative Law

- Some operators in math are *associative*
- For example: (a + b) + c = a + (b + c)
- Same applies to ∧ and ∨

$$a \wedge (b \wedge c) \equiv (a \wedge b) \wedge c$$
$$a \vee (b \vee c) \equiv (a \vee b) \vee c$$

56

Absorption Law

- There is a special case of the Distributive Law where one variable is *absorbed* (i.e. eliminated)
- Applies to both ∧ and ∨

$$a \wedge (a \vee b) \equiv a$$
$$a \vee (a \wedge b) \equiv a$$

57

Idempotent Law

- When a statement is combined with itself, it is equivalent to just the statement (no duplicate)
- This applies to both ∧ and ∨

$$a \wedge a \equiv a$$
$$a \vee a \equiv a$$

58

Involution Law

- One of the most basic equivalences in logic is the *double negation*
- It is fairly obvious, so not more needs to be said

$$\neg\neg a \equiv a$$

59

Domination Law

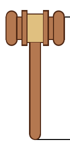
- This might look similar to the identity law, but look very careful at which operator is being used
- These will result in either true or false.

$$a \vee \text{true} \equiv \text{true}$$
$$a \wedge \text{false} \equiv \text{false}$$

60

DeMorgan's Law

- So, it states you can change the operator from $\wedge$ to $\vee$ or vice-versa
- If you negate both operands



$\neg (a \wedge b) \equiv \neg a \vee \neg b$

$\neg (a \vee b) \equiv \neg a \wedge \neg b$

61

Truth Table – Testing Not-Or

a	b	¬ a	¬ b	¬ a ∧ ¬ b	¬ (a ∨ b)
T	T	F	F	F	F
T	F	F	T	F	F
F	T	T	F	F	F
F	F	T	T	T	T

62

Truth Table – Testing Not-And

a	b	¬ a	¬ b	¬ a ∨ ¬ b	¬ (a ∧ b)
T	T	F	F	F	F
T	F	F	T	T	T
F	T	T	F	T	T
F	F	T	T	T	T

63

Example

- So, can we simplify the expression below?
- Let's create a truth table

$(a \wedge b) \vee (\neg a \wedge b)$

64

Boolean Algebra

- Truth tables become unwieldy as the number of variables increase
- Logical algebra is another way to evaluate equivalence
- Equivalences can be used to generate one expression from another

65

Example Simplification

$(a \wedge b) \vee (\neg a \wedge b) =$

$(a \vee \neg a) \wedge b =$ After using Distributive Law

$\text{true} \wedge b =$ After using Complement Law

b After using Identify Law

66

Implication

The operator of science

67

Implication

- The only Boolean operator that causes confusion is implication
- However, its usage is ***vital*** to understand – since it is used in your programs (even if you might not see it)

68

Implication

- For "p implies q"…
- p is called the *antecedent* (or hypothesis or assumption)
- q is called the *consequent* (or conclusion)

69

Implication

- "p implies q" is contradicted (false) **only** when…

  p is true and **q is false**

- In all other cases, it is true

70

Standard Precedence Levels

1	¬
2	∧
3	∨ ⊕
4	→

Highest Level

Lowest Level

71

Implies Example

True True

1 < 2 → 10 < 40

Result: True

72

Implies Example 2

True

False

5 > 3 → 44 < 8

Result: False

73

Implies Example 3

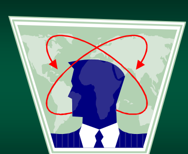
False

False

10 < 2 → 12 < 10

Result: True

74



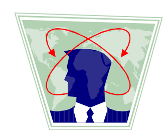
Analyzing Implication

Understanding is Power

75

Analyzing Implication

- Implication is both simple and complex
- It is used in all aspects of logical proof and the basis of all programs
- Understanding is complexity is essential to understanding logic (and discrete math)



76

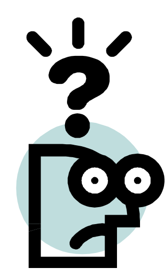
Implication Examples

- *If the Moon is made of cheese then the Moon is a tasty snack.*
- *If the flag has a bear then it's the Flag of California.*
- *If it is a fish then it is lives in water.*
- *If the university is Sacramento State then the mascot is a hornet.*

77

Many Ways to Say "Implies"

- *A implies B*
- *A → B*
- *B if A*
- *If A then B*
- *B given A*



78

Example From History

- Let's look at a implication from California history
- During the Gold Rush, people were inspired by a simple idea…
- *"If I pan for gold then I'll get rich!"*

79

"If I pan for gold then I'll get rich"

- Let's look at this statement closer
- It can be rewritten: *"Pan of Gold→ Get Rich"* or, very tersely, *"P → R"*

P → R

80

"If I pan for gold then I'll get rich"

- There four combinations of the truth table
- Which of these combinations would invalidate the statement

P → R

81

True → True

- If P is true, and R is true…
- *"We panned for gold and got rich"*
- Statement is true
  - we asserted that if P is true then R is true
  - since both are true, the statement is affirmed
  - true → true = true

82

False → True

- What if P is false and R is true
- *"We didn't pan for gold and got rich"*
- Statement is true
  - the fact we got rich (without panning for gold), doesn't mean that the statement is false
  - it has not contradicted the statement
  - false → true = true

83

False → False

- What if P is false and R is false?
- *"We didn't pan for gold and didn't get rich"*
- Statement is true
  - the fact that both are false, still does not contraction our original statement
  - it stated "IF we pan for gold **then** we get rich"
  - false → false = true

84

True → False

- Finally what if P is true and R is false?
- *We panned for gold, but didn't get rich*
- Statement is false
  - we asserted if P is true then R must be true
  - however, since this contradicts the assertion, the result of the implication is false
  - true → false = false

85

Implication Hiding in Plain Sight

- Consider the expression below
- The word "then" is alternative way of saying "implies"
- So, Is it True? False?

if x > 2 then x^2 > 4

86

Implication Hiding in Plain Sight

- There are different values of x that will make the antecedent and consequent both true and false
- If both are true, then the statement is correct

if x > 2 then x^2 > 4

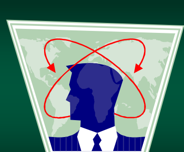
87

Implication Hiding in Plain Sight

- If **x > 2** is false, then we don't care about the consequent

if x > 2 then x^2 > 4

88



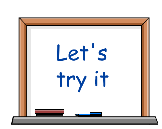
Deconstructing Implication

Other operators; same logic

89

Deconstructing Implication

- The implication logic can be broken down into the forms that are easier to remember
- This is actually quite important when we cover a few logical tricks later one
- So, let's look at the truth table for other operators



90

Analyzing Implication

- So, can implication be written using just logical "and", "or", or "not"?
- Yes, we can!

$p \rightarrow q$

91

Truth Table – One way to do it

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

True whenever q is true

True whenever p is false

92

Analyzing Implication

- So, we can definite $p \rightarrow q$ as "not p or q"
- Like before, let's prove in our truth table

$p \rightarrow q \equiv \neg p \lor q$

93

Truth Table – Or Logic

p	q	$\neg p$	$\neg p \lor q$	$p \rightarrow q$
T	T	F	T	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

94

Truth Table – One way to look at it

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Only false for:

$p \land \neg q$

We can negate this.

95

Analyzing Implication

- So, we can definite $p \rightarrow q$ as "not (p and not q)"
- It doesn't look quite right, let's test it out

$p \rightarrow q \equiv \neg (p \land \neg q)$

96

Truth Table – And Logic

p	q	¬ q	p ∧ ¬ q	¬ (p ∧ ¬ q)	p → q
T	T	F	F	T	T
T	F	T	T	F	F
F	T	F	F	T	T
F	F	T	F	T	T

Analyzing Implication

$$p \rightarrow q \equiv \neg(p \land \neg q)$$
$$\equiv \neg p \lor q$$

Arguments

Part 7

1

Arguments

Proving a Point

2

Arguments

- A combination of true statements can be used to claim another as true
- An *argument* is a collection of statements (called *premises*), which, when all are true, imply a consequence

3

Example Argument

```
raining → wet outside
not wet outside
_____________________
?
```

Our conclusion goes here

4

Example Argument 2

```
raining → wet outside
not raining
_____________________
?
```

What conclusion goes here?
Does it work?

5

Example Argument 3

```
x is duck or x is swan
x isn't a swan
_____________________
?
```

Obvious! But why?

6

When an Argument is Valid

- When all the premises are true then the consequence must be true
- If all the premises are true, but the conclusion can be false, the argument is disproven

7

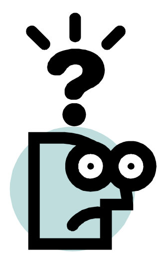
When an Argument is Valid

- However, if any premise is false, then the argument is not disproven – *it is still valid*
- We can often prove arguments by building truth tables

8

Argument Notation

- Arguments can be written out several ways
- The most common approach is to write each premise on a different line
- The consequence is written below the premises separated with horizontal line



9

Common Notation

$p \rightarrow q$

$q \rightarrow r$ — Premises

$p \rightarrow r$ — Consequence

10

Another Argument Notation

- Arguments can be written on a single line
- Premises are separated with commas
- Consequence can use the symbol ⊦ or ∴

"Therefore"

$p \rightarrow q,\ q \rightarrow r\ \vdash\ p \rightarrow r$

11

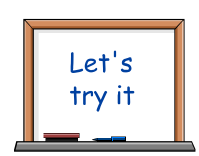
Let's Try This

$p \rightarrow q$

p

---

q — Valid!



12

Arguments and Implication

- Arguments are actually implications with each premise connected with $\wedge$
- So, if you have premises A, premise B, and conclusion C, then it has the following form

$$A \wedge B \rightarrow C$$

13

Valid Arguments

Proving a Point

14

Valid Arguments

- *Rules of Inference* are valid arguments that are commonly used in proofs
- Most of these are obvious to you… it is natural logical thought

15

Rules of Inference

- *Modus Pones*
  (aka Law of Detachment)
- *Modus Tollens*
- *Disjunctive Syllogism*
- *Hypothetical Syllogism*

16

Modus Ponens

- *Modus Ponens* is the most basic Rule of Inference
- Based on the logic that if:
  - an implication is true
  - implication's hypothesis is true
  - then the implication's conclusion must be true

17

Modus Ponens Example

If it is a fish, then it lives in water.

It is a fish.

Therefore, it lives in water!

18

Modus Ponens

Rules of Inference

$p \rightarrow q$
p

---

q

19

Modus Ponens

p	q	p → q
T	T	T
T	F	F
F	T	T
F	F	T

20

Modus Tollens

- *Modus Tollens* is closely relate to modus ponens
- Based on the logic that if:
  - an implication is true
  - implication's conclusion is false
  - then the implication's hypothesis must be false

21

Modus Tollens Example

If it is a fish, then it lives in water.

It doesn't live in water.

Therefore, it is <u>not</u> a fish!

22

Modus Tollens

Rules of Inference

$p \rightarrow q$
$\neg q$

$\neg p$

23

Modus Tollens

p	q	p → q	¬ p	¬ q
T	T	T	F	F
T	F	F	F	T
F	T	T	T	F
F	F	T	T	T

24

Disjunctive Syllogism

- *Disjunctive Syllogism* is based on the ∨ operator that
- Based on the logic that if:
  - an or-statement is true
  - one of the operands is false
  - then, the other operand must be true

Disjunctive Syllogism Example

It breathes water or air.

It doesn't breath water.

Therefore, it breathes air.

Disjunctive Syllogism

$p \lor q$
$\neg p$

---

q

Rules of Inference

Hypothetical Syllogism

- *Hypothetical Syllogism* is based on an implication chain
- Gives a logical "chain" of events
- So, if a → b and b → c then a → c

Hypothetical Syllogism Example

If is a trout, then it is a fish

If it is a fish, then it lives in water.

Therefore, a trout lives in water!

Hypothetical Syllogism

$p \rightarrow q$
$q \rightarrow r$

$p \rightarrow r$

Rules of Inference

Let's Apply The Logic

- Let's these rules on the argument below
- We can use either a truth table or logical deduction

```
If I study then I will an A
If I don't watch Netflix then I will study
I didn't get an A
________________________
I watched Netflix
```


31

Simply Logic: letters

- First, let's simply the structure of the argument so we can see the logic
- We will assign each part a letter

```
s = studied
a = got an A
n = watched Netflix
```


32

Simply Logic: New Form

```
1.  s → a
2.  ¬ n → s
3.  ¬ a
    ___________
    n
```


33

Modus Tollens – study, no A

```
1.  s → a
2.  ¬ n → s
3.  ¬ a
    ___________
    n
```

1 and 3:
Modus Tollens

34

Modus Tollens – study, no A

```
1.  ¬ s
2.  ¬ n → s
3.  ¬ a
    ___________
    n
```

1 and 3:
Modus Tollens

"Did not study"

35

Modus Tollens – study, Netflix

```
1.  ¬ s
2.  ¬ n → s
3.  ¬ a
    ___________
    n
```

1 and 2:
Modus Tollens

36

Modus Tollens – study, Netflix

1. ¬ s
2. ¬ ¬ n
3. ¬ a

---

n

1 and 2:
Modus Tollens

"Did not *not* watch Netflix"

Spring 2020 Sacramento State - Cook - CSc 28 37

37

Double Negation

1. ¬ s
2. ¬ ¬ n
3. ¬ a

---

n

Double negation

Spring 2020 Sacramento State - Cook - CSc 28 38

38

Double Negation

1. ¬ s
2. n
3. ¬ a

---

n

2:
Double Negation

"Watched Netflix"

Spring 2020 Sacramento State - Cook - CSc 28 39

39

Logical Fallacies

"This is most illogical" – Mr. Spock

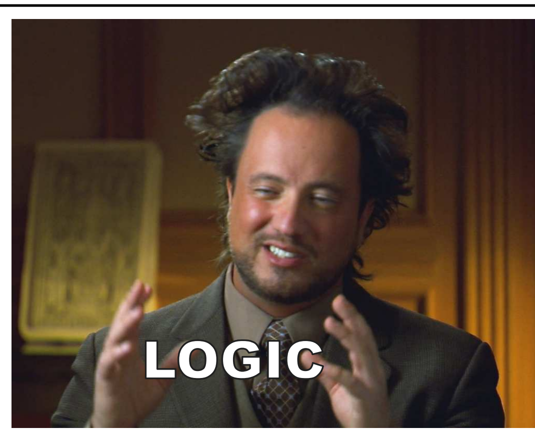
40

Logical Fallacies

- There are a number of fallacious arguments that, while they might look logical, are wrong
- The following slides contain some of them
- For fun, apply them to current political discourse or *History Channel 2*

Spring 2020 Sacramento State - Cook - CSc 28 41

41



42

Fallacy of the Converse

- *Fallacy of the Converse* is based on assumption that if the conclusion is true then the hypothesis is true
- Also called:
  - *affirming the consequent*
  - *converse error*

43

Fallacy of the Converse Example

If it is a fish, then it lives in water.

It lives in water.

Therefore, it is a fish!

44

Fallacy of the Converse

$p \rightarrow q$
q

---

p

p can still be false

45

Fallacy of the Converse

p	q	p → q
T	T	T
T	F	F
F	T	T
F	F	T

46

Fallacy of the Inverse

- *Fallacy of the Inverse* is based on assumption that if the hypothesis is false, then the conclusion is also false
- Also called:
  - denying the antecedent
  - inverse error

47

Fallacy of the Inverse Example

If it is a cat, then it is furry.

It is not a cat.

Therefore, it is not furry!

48

Fallacy of the Inverse

$p \rightarrow q$

$\neg p$

$\neg q$

q can be either true or false

Fallacy of the Converse

p	q	p → q	¬ p	¬ q
T	T	T	F	F
T	F	F	F	T
F	T	T	T	F
F	F	T	T	T

Fallacy of Affirming a Disjunct

- *Fallacy of Affirming a Disjunct* is based on assumption that if there are two attributes and one is true, the other must be false
- Other names:
  - *fallacy of the alternative*
  - *false exclusionary disjunct*

Affirming a Disjunct Example

Suspect is either a politician or a lawyer.

Suspect is a politician.

Therefore, the suspect isn't a lawyer.

Fallacy of Affirming a Disjunct

$p \vee q$

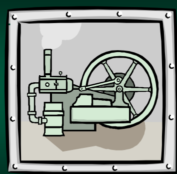
p

$\neg q$

Just because p is true, doesn't mean q has to be false.

Fallacy of the Converse

p	q	p ∨ q	¬ q
T	T	T	F
T	F	T	T
F	T	T	F
F	F	F	T



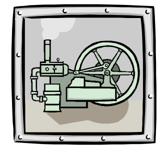
Boolean Algebra & Proofs

Proofs and Logic are one

55

Boolean Algebra

- Remember Boolean algebra laws: Associative, Commutative, etc…
- These can be used to expand an expression… and then simplify it in a different form



56

Example

if "a or b" is true and

"a and b" is false

then

a = not b

57

Example (with a rewrite)

We can rewrite it as an argument:

```
a ∨ b = true
a ∧ b = false
_____________
a = ¬b
```


58

The Strategy

- We could use a Truth Table to prove this
- Let's use Boolean Algebra to prove if this is correct

```
a ∨ b = true
a ∧ b = false
_____________
a = ¬b
```


59

The Approach

- Start with a
- Try to change it into ¬ b
- Use the premises to replace values

```
a ∨ b = true
a ∧ b = false
_____________
a = ¬b
```


60

$a \lor b = \text{true}$
$a \land b = \text{false}$

$a =$	a	
$=$	$a \land \text{true}$	Identity
$=$	$a \land (b \lor \neg b)$	Complement
$=$	$a \land b \lor a \land \neg b$	Distributive
$=$	$\text{false} \lor a \land \neg b$	Premise
$=$	$b \land \neg b \lor a \land \neg b$	Complement
$=$	$\neg b \land (b \lor a)$	Distributive
$=$	$\neg b \land \text{true}$	Premise
$=$	$\neg b$	Identity

61

Please Wait

CSC 28
will begin shortly

(open the chat window)

1

Basic Proof

Part 8

2

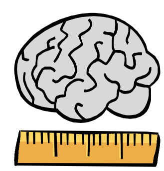
Theorems

The Big Bang Theory

3

Theorems

- A *theorem* is a statement we intend to prove using existing known facts (called *axioms* or *lemmas*)
- Used extensively in all mathematical proofs – which should be obvious



4

Example

- Most theorems are of the form: If A, then B
- The theorem below is very easy to interpret

If a and b are even integers
then a × b is an even integer

5

Example

- Theorems are arguments
- They can be structured as such

```
a is even
b is even
_____________
a × b is even
```


6

Example

- Sometimes it is hard to see
- Below, the same theorem is written using different language

x and y are even integers and the product is even.

The product xy is even when x and y are both even.

7

Some Basic Definitions

Abstract? Not really.

8

Definition: x is even

a is even if and only if…

a = 2n for some integer n

9

Definition: x is odd

a is odd if and only if…

a = 2n + 1 for some integer n

10

Definition: x | y (x divides y)

a | b (a divides b) iff…

b = k × a for some integer k

11

Definition: x ∈ Q (x is a rational)

a is a rational number iff…

a = b / c for some integers b and c, and c ≠ 0

12

Proving A → B

Modus Ponens

13

Proving A → B

- The Boolean operator A → B is true except when A is true and B is false
- So, we can logically prove that A → B by showing that *whenever A is true, B **must** also be true*
- *… so T → F isn't possible*

14

Proving A → B

- This is essentially a *Modus Ponens* proof
- You are showing that if A is true, and A → B is true, then B **must** be true
- Also note that "A" and "B" can be compound statements

15

Modus Ponens

A → B — Prove this is always true

A — If this is true

B

16

The Steps

- There are basically just two steps to follow:
  1. Assume A is true
  2. Show that B must be true
- This shows that B is true whenever A is true

17

Example 1

- Let's prove the following theorem from before
- This is actually quite easy

If a and b are even integers
then a × b is an even integer

18

Example 1 – the argument

```
a is even
b is even
──────────
a × b is even
```


19

Example 1 – internal structure

- Remember: all proofs are implications
- So, we will assume the both premises are true and show the conclusion must be true

```
a is even ∧ b is even → a × b is even
```


20

Example 1

```
Assume that x and y are even integers.

So, by the definition…

a = 2i and b = 2j (for some i, j)

Note: use different arbitrary variables
or you are assuming they are equal!
```


21

Example 1 - Proof

```
So, the product is:

a × b = 2i × 2j
      = 4 × i × j
      = 2 × (2 × i × j)

So, by definition, a × b is even
```


22

Example 2

- The following is a theorem about the product of an odd and even number
- The proof is straight-forward using the definitions

If a is even and b is odd, then a × b is even

23

Example 2

```
Assume:
a is an even integer and
b is an odd integer.

Then a = 2i and b = 2j+1 for some
integers i and j
```


24

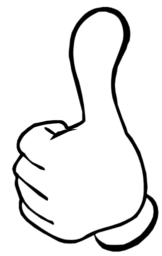
Example 2

```
Multiplying, we get:

a × b = (2i) × (2j + 1)
      = 4ij + 2i
      = 2 × (2ij + i)

...which is even
```


25

Proof Tips

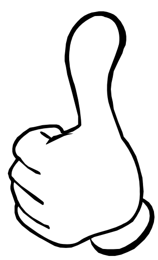
- Begin your proof with what you assume to be true (the hypothesis)
- Don't argue the truth of a theorem *by example*
  - stay abstract
  - e.g. you know x and y are even integers – that's all you know



26

Proof Tips

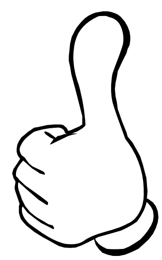
- Your proof must work forward from your assumptions to your goal
- You may work backward on scratch paper to help you figure out how to work forward



27

Proof Tips

- Write your proof in prose
- Then read out loud, it should sound like well-written paragraph
- Quite often you'll use definitions to make progress



28

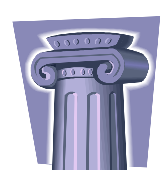
Proof by Contrapositive

Proof with inverse logic

29

Proof by Contrapositive

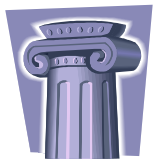
- There are several techniques that can be employed to prove an theorem
- The direct approach, like before, is quite common, but its not the only path you can take



30

Proof by Contrapositive

- *Proof by Contrapositive* has you prove the opposite of the original theorem
- Quite impressively, this will also prove the original theorem



31

Getting the Contrapositive

- First…
  - negate both the assertion and conclusion of the implication
  - so, basically, put "not" in front of both operands
- Second…
  - reverse the implication
  - you basically swap the left-hand and right-hand operand of the implication

32

Getting the Contrapositive

- So, both operands swap positions and are negated
- Are they equal? Let's confirm in a Truth Table

```
for p → q
contrapositive is ¬q → ¬p
```


33

Contrapositive Truth Table

p	q	¬ q	¬ p	p → q	¬ q → ¬ p
T	T	F	F	T	T
T	F	T	F	F	F
F	T	F	T	T	T
F	F	T	T	T	T

34

Modus Ponens Contrapositive

Prove always true

$\neg$ B → $\neg$ A

$\neg$ B

If this is true (i.e. B is false)

$\neg$ A

35

How it Works

- So, if we prove the contrapositive, we also prove the original theorem
- For the original A → B
  - suppose that if B is false
  - show that A must be false
- It does make sense, if you think about it

36

Example

- The following theorem should look familiar
- This theorem states that the square of a odd number is also odd
- *Direct proof is near impossible!*

If x^2 is odd then x is odd

37

Example Contrapositive

- The contrapositive negates each operand in the implication A → B
- The following shows the reverse of each

A = x^2 is odd

¬A = x^2 is not odd = x^2 is even

38

Example Contrapositive

- The contrapositive negates each operand in the implication A → B
- The following shows the reverse of each

B = x is odd

¬B = x is not odd = x is even

39

Example Contrapositive

- Finally, we reconstruct our theory with B → A rather than A → B
- This expression is equivalent to the original

if x is not odd then x^2 is not odd

or rewritten as…

if x is even then x^2 is even

40

Example Contrapositive

We assume x is not odd

x is not odd means x is even

x = 2k for some integer k

41

Example Contrapositive

We assume x is not odd (even)

$$x^2 = (2k)^2 = 4k^2 = 2(2k^2)$$

So, x^2 is even which is *not odd*

42

Example Result

We proved:

if x is even then x^2 is even

By contrapositive, we proved:

if x^2 is odd then x is odd

Proof by Contradiction

Welcome down the rabbit hole

Proof by Contradiction

- *Proof by Contradiction* takes a novel approach
- It uses the approach of *reductio ad absurdum*
- So what is it? Well, it proves the theorem by showing it can't be false

But, How?

- Assume it is false
- Show that (if it is false) something impossible results
- Therefore, it can't be false and, thus, true!

Proving $A \rightarrow B$

- Argue: $A \land \neg B$ …which is $\neg(A \rightarrow B)$
- Show that something *impossible* results!
- Since $A \rightarrow B$ cannot be false, it **must** be true

Contradiction

A	B	¬ B	A ∧ ¬B	¬ (A → B)
T	T	F	F	F
T	F	T	T	T
F	T	F	F	F
F	F	T	F	F

Example

- The following is a classic Proof By Contradiction
- The theorem covers if the square-root of 2, is an irrational number

The square root of 2 is irrational

49

Example

- To prove by contradiction, we need to show that the opposite cannot be true (i.e. false)
- The sentence bellow is the theorem negated
- So, how do we go about proving this?

The square root of 2 *is rational*

50

Example

- Well, what is a rational number?
- A rational number can be expressed as "a / b" where a and b are *integers with no common factors* (aka "lowest terms")

The square root of 2 *is rational*

51

Examples of Rational Numbers

- **1 / 3**
- **7 / 1**
- **22 / 7**
- **7734 / 10001**
- **1 / 123456789**
- **5 / 3**

52

The Proof: Prove Opposite

$\sqrt{2}$ is rational

$\sqrt{2}$ = a / b where a, b $\in$ Z
and b $\neq$ 0
and a, b have no common factors

53

Analyzing a

$\sqrt{2} = a / b$

Let's look at the properties of a

$\sqrt{2} \times b = a$

$2 \times b^2 = a^2$

54

Analyzing a – It is even

$2 \times b^2 = a^2$

So… a^2 is an even number

therefore, we know a is also even (previous proof – even × even)

Analyzing b

Since a is even and a / b is in lowest terms, then b must be odd

Why? If b is even, then a / b would have common factors – namely 2.

Example: Oh ohhhhh

However… look again at $2 \times b^2 = a^2$

Since a is even, we can use the definition. So…

$2 \times b^2 = (2k)^2$
$= 4k^2$

Example: Oh ohhhhh

Solving for b^2…

$2 \times b^2 = 4k^2$
$b^2 = 2k^2$

Since b^2 is even, b is even

Result

Since b must be both odd and even, we have a contradiction

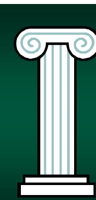
The theorem "square root of 2 is rational" cannot be true

Therefore, "square root of 2 is irrational" is true

Predicate Logic

Part 9

1



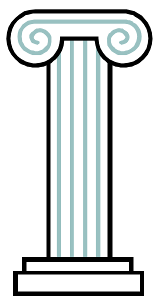
Predicates

Just the facts…

2

Predicates

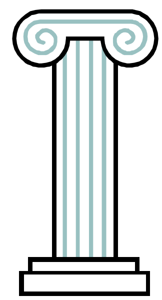
- A predicate is a statement about one or more variables
- It is stated as a fact – being true for the data provided



3

Predicates

- Predicates express *properties*
- These can apply to a single entity or *relations* which may hold on more than one individual



4

Predicate Notation

- It follows the same basic syntax as function calls in Java (and most programming languages)
- However, type case is important:
  - constants start with lower case letters
  - predicates start with upper case letters

5

Single variable predicate

- Predicates can have one variable (at a minimum)
- The following sentence states one that the cat named Pattycakes has the "sleepy" property

```
"Pattycakes The Cat is sleepy"
```


6

Single variable predicate

- Alternatively, we can write that property in predicate form
- "Sleepy" predicate for "Pattycakes" is true
- Note the uppercase and lowercase!

```
Sleepy(pattycakes)
```


7

Two Variable Predicate

- Predicates can have multiple variables (unlimited actually… well within reason)
- The following is a classic example of a two-variable relationship

```
x < y
```


8

Two Variable Predicate

- The LessThan predicate is true for x, y

```
LessThan(x, y)
```


9

Predicates Summary

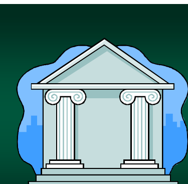
- 1-place predicates assign properties to individuals:
  - ___ is a cat
  - ___ is sleepy
- 2-place assign relations to a pair
  - ___ is sleeping on ___
  - ___ is the capitol of ___

10

Predicates Summary

- 3-place predicates assign relations to triples
  - ___ wants ___ to _____
  - Cat named _____ likes to ___ on _____
- Etc…

11



Quantified Statements

More Symbols

12

Quantified Statements

- Sometimes we want to say that *every element in the universe* has some property
- Let's say the universe is the people in this Zoom "room" & we want to say "*everyone in the room is awake*"

Spring 2020 Sacramento State - Cook - CSc 28 13

13

Propositional Approach #1

- We can write out a descriptive sentence
- Shortcomings:
  - it is monolithic an inflexible
  - not "mathematical" enough

```
Everyone in this room is awake.
```

Spring 2020 Sacramento State - Cook - CSc 28 14

14

Propositional Approach #2

- We can also write out that sentence using a long list of predicates
- So, we list them all or make a pattern
- Shortcomings: cumbersome & verbose

```
P(moe) and P(larry) and P(curly) and …
```

Spring 2020 Sacramento State - Cook - CSc 28 15

15

Limitations of Propositional Logic

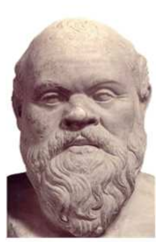
- While propositional logic, which we covered, can express a great deal of complex logical expressions, it ultimately is insufficient for all arguments
- Why? The premises (and conclusions) in propositional logic have no internal structure.

Spring 2020 Sacramento State - Cook - CSc 28 16

16

Limitations of Propositional Logic

- For example, it cannot show the validity of Socrates Argument
- This arguments states: *"All humans are mortal. Socrates is a human. Therefore, he is mortal."*



Spring 2020 Sacramento State - Cook - CSc 28 17

17

Socrates Argument

- The following is the propositional logic form of the Socrates Argument
- Can we prove the conclusion?

```
All humans are mortal
Socrates is a human
______________________________
Therefore, Socrates is mortal
```

Spring 2020 Sacramento State - Cook - CSc 28 18

18

The Socrates Argument

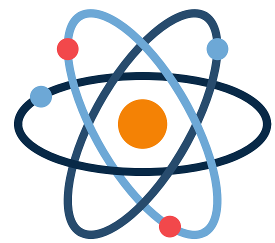
- The following is the argument in normal form
- A problem arises since the validity of this argument comes from the internal structure which propositional logic cannot "see"

```
H
S
───
M
```


19

Solution

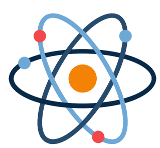
- It's time to break apart the logic and see the internal structure
- So, ***we are splitting the atom!***



20

Why Go Nuclear?

1. Expose the internal structure of those "atomic" sentences
2. Create new terminology to describe the semantics
3. Introduce laws to use and manage them



21

New Notation: For All

- The "For-All" symbol states every element *x* in the universe makes P(*x*) true
- So, it is true if and only if the every element *x* in the universe has P as true

$\forall_x$ **P(x)**

22

New Notation: Exists

- The "Exists" States at least one element *x* in the universe makes P(*x*) true
- True if just a single P is true

$\exists_x$ **P(x)**

23

Example: Pineapple Pizza

- Let's create the quantified statement for *"Someone doesn't like pineapple pizza!"*
- Let's create a predicate P(x) means *"x likes pineapple pizza"*
- What does someone mean? At least one person?

24

Pineapple Pizza: Try #1

- How about the following expression?
- It's not true if at least one person likes pineapple
- This means *"nobody likes pineapple pizza"*

$\neg (\exists_x P(x))$

25

Pineapple Pizza: Try #2

- We can also negate the predicate
- Means, for at least one person, they dislike pineapple pizza
- *"Someone doesn't like pineapple pizza!"*

$\exists_x \neg P(x)$

26

Quantifier Equivalence

Quantifier Conversion

27

Equivalence

- Just like propositional logic, quantitative expressions have equivalencies
- They follow the same basic logic we have seen before

28

Example: Opposite Expression

- Example: "Everyone in the room is awake"
- Let's create the reverse of this expression *(that still says the same thing)*

"Everyone in the room is awake."

29

Example: Opposite Expression

- So, let's just negate the predicate "is awake" into "is asleep"
- Does that work? No.

"Everyone in the room is asleep."

30

Example: Opposite Expression

- Now, let's negate the quantifier "everyone'" (for-all) into "someone" (exists)
- The expression below works: almost

"Someone in the room is asleep."

31

Example: Opposite Expression

- Well, what if we change the quantifier "everyone'" (for-all) into "someone" (exists)
- The expression below works: yes

"There is no one in the room that is asleep."

32

Exists and For-All

- The previous two laws allows us to extrapolate two additional laws
- Note: we simply push the negative and remove the double-negation

$\forall_x\ P(x) \equiv \neg \exists_x \neg P(x)$

$\exists_x\ P(x) \equiv \neg \forall_x \neg P(x)$

33

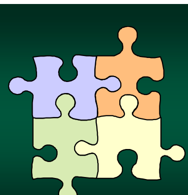
Equivalence – Moving Negation

- You can push a negation through a quantifier by toggling the quantifier
- Read the expressions below carefully…

$\neg \exists_x\ P(x) \equiv \forall_x \neg P(x)$

$\neg \forall_x\ P(x) \equiv \exists_x \neg P(x)$

34



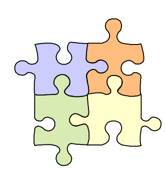
Conjunction & Disjunction

Breaking Apart and Combining

35

Conjunction & Disjunction

- Both the Exists and For-All quantifiers can be broken apart (and combined)
- This can occur if the expression contains an AND or an OR



36

Exists Disjunction

- If the Exists quantifier is used on a disjunction, it can be broken into two Exists
- This only works with $\lor$

$$\exists_x (P(x) \lor Q(x)) \equiv \exists_x P(x) \lor \exists_x Q(x)$$

37

For-All Conjunction

- If the For-All is used on a conjunction (and), it can be broken into two For-All
- This only works with $\land$

$$\forall_x (P(x) \land Q(x)) \equiv \forall_x P(x) \land \forall_x Q(x)$$

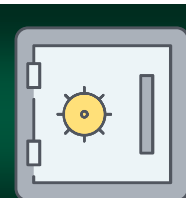
38

Please Wait

CSC 28
will begin shortly

(open the chat window)

39



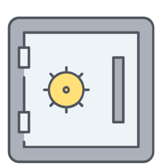
Bound & Free Variables

Some variables are not variable

40

Bound & Free Variables

- Not all variables used in a quantified expression is treated the same
- Each variable in an expression is either considered "bound" or "free"



41

Bound & Free Variables

- A variable is *free* if a value must be supplied to it before expression can be evaluated
- A variable is *bound* if it not free (usually a dummy variable) and contains values that are not needed to be evaluated

42

Example 1

- Which variables need we supply a value before the expression can be evaluated?
- Both x and c
- Without knowing both we cannot evaluate the expression (both are free)

```
(x ^ 2 < 4 * c)
```


43

Example 2

- Which variables need to be supplied before the expression can be evaluated?
- x: no, it is a dummy variable
- c: yes, once we give a value for c, we can evaluate the expression

$\forall_x$ `(x ^ 2 < 4 * c)`

44

Multiple Quantifiers

Many E's and A's doing headstands

45

Multiple Quantifiers

- A quantified statement may have more than one quantifier
- In fact, most of the time, statements will contain several

46

Example

- x > y is an expression with two variables
- The expression is true if an x is supplied which is greater than y

$\forall_x \exists_y$ `P(x, y)`

47

Example

- ∃y (x > y) is an expression with <u>one</u> free variable
- Evaluates to true if <u>x is supplied</u> and there is a y greater than the supplied x

$\forall_x \exists_y$ `P(x, y)`

48

Example

- $\forall x\ \exists y\ (x > y)$ contains no free variables
- Evaluates to true if $\exists y\ (x > y)$ is true for every x in the universe.

$\forall_x\ \exists_y$ `P(x, y)`

49

Example 2

- The following in an implication with two quantifiers as operands
- It states that whenever "$\forall x$ P(x)" is a true statement, then so is "$\forall x$ Q(x)".

$\forall_x$ `P(x)` $\rightarrow$ $\forall_x$ `Q(x)`

50

Converting English to Logic

Do it bit by bit

51

Difficult Example

- Let's create a quantified statement for the following logical statement
- *We will go slowly, since this is not easy*

Everyone who has a friend who has Covid will have to be quarantined

52

Difficult Example

- "Everyone" is a For-All relationship
- What is everyone referring to? People
- So, the abstract object is a person

$\forall_x$ `(if x has a friend with Covid,`
`then x must be quarantined)`

53

Difficult Example

- So, we can factor it out into the expression below – x is a person
- *Now*, let's look at the sub expression…

$\forall_x$ `(if x has a friend with Covid,`
`then x must be quarantined)`

54

Difficult Example

- The sentence *" if x has a friend with Covid, x must be quarantined"* is an implication!
- Let's look at the antecedent (hypothesis)

```
if x has a friend with Covid,
   then x must be quarantined
```


Difficult Example

- How do we write the concept: *"x has **a** friend with Covid"* ?
- They just need a single friend
- So, this is an Exists quantifier

```
∃y (x is friends with y, and y has Covid)
```


Difficult Example

- Now that we have a basic form of the final version, let's make some predicates
- We will use single letter names for brevity

```
F(x, y) means "x and y are friends"
C(x)    means "x has Covid"
Q(x)    means "x must be quarantined"
```


Difficult Example

- This says: "There exists a *person y* where *y* is friends with *person x*, and *y* has Covid"
- Note: *x* is not bound in this expression

$$\exists_y \ (F(x, y) \wedge C(y))$$

Difficult Example

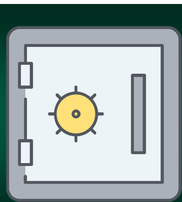
- So, what happens if a friend has Covid?
- Then, they must be quarantined
- Note: implication is outside the exists

$$\exists_y \ (F(x, y) \wedge C(y)) \rightarrow Q(x)$$

Difficult Example

- Now we can put it all together…
- The following is the quantified expression for our original statement

$$\forall_x \ (\exists_y \ (F(x, y) \wedge C(y)) \rightarrow Q(x))$$



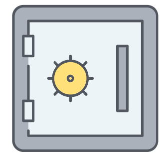
Bounded Quantifiers

Hidden Implication
(for those who hate to type)

61

Bound Quantifiers

- Some quantifiers can be more than meets the eye
- For brevity, many predicate and propositional expresses are merged with the $\forall$ and $\exists$



62

Shorthand Notation

- The following type of expression is quite common
- So much so that a shortcut notation is often employed

$$\forall_x \ (\mathbf{R(x)} \rightarrow \mathbf{P(x)})$$

63

Shorthand Notation

- The membership sub-expression is moved to the quantifier's subscript
- This is equivalent to the last

$$\forall_{\mathbf{R(x)}} \mathbf{P(x)}$$

64

Likewise…

- The sub-expression before the implication can be anything
- In this example, x > 5 is moved to the subscript

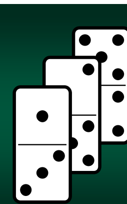
$$\forall_x \ (\mathbf{x} > 5 \rightarrow \mathbf{P(x)}) \equiv \forall_{x > 5} \ \mathbf{P(x)}$$

65

Induction

Part 10

1



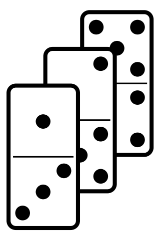
Induction

Proof by Pattern

2

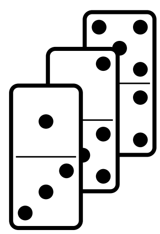
Induction

- Many proofs, in fact a great number of them, are based on "all positive integers"
- *Induction* is a technique of proving a theorem that is based on this criteria



3

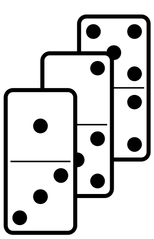
Induction



- The proof by induction is based on the *Well-Ordering Property*
- It states that: given a set of non-negative numbers there is a *least* element

4

Induction



- Induction basically helps prove $\forall_x P(x)$ where the universe is positive numbers
- *Or any range starting at one point and going off to infinity (positive or negative)*

5

How it Works

- It works in 2 steps
  1. proving P(1) and then
  2. proving that $P(n) \rightarrow P(n + 1)$
- As a result...
  - since $P(n) \rightarrow P(n + 1)$
  - then $P(n + 1) \rightarrow P(n + 2)$ and so on…

6

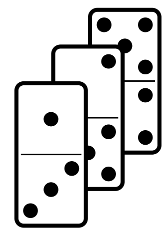
Metaphor: Line

- There is a line of people
- First person tells a secret to the second person
- The second person then tells it to the third
- … and so on until everyone knows the secret

7

Metaphor: Dominos

- You have a long row of Dominos
- The first domino falls over and hits the second domino
- The second hits the third
- … and so on until they are all knocked over



8

Foundation of Induction

- The following summarizes the proof technique
- It is important to understand the approach since it is so commonly used

$$P(1) \land \forall_n \, (P(n) \rightarrow P(n+1)) \rightarrow \forall_n \, P(n)$$

9

Steps to Proof

- Step 1: *Basis*
  - show the proposition P(1) is true
  - very easy to do – just plug in the values
- Step 2: *Induction*
  - assume P(n) is true (which is your theorem)
  - show that P(n + 1) must be true

10

Example: Sum of Odds

Using induction…

Show that the sum of n odd numbers equals n^2

11

Example: Sum of Odds

- So, the sum of odd numbers is a square?
- Is that true?
  - 1 + 3 = 4
  - 1 + 3 + 5 = 9
  - 1 + 3 + 5 + 7 = 16
- Okay, that's just odd! *(pun intended)*

12

Sum of Odds

```
P(n) is written as:

   1 + 3 + 5 + ... + (2n – 1) = n²

P(n + 1) is written as:

   1 + 3 + … + (2n–1) + (2n+1) = (n+1)²
```


13

Basis: Sum of Odds

- The sum of odds, for just 1 number is simply 1
- Of course, this is also 1 squared

```
P(1) = 1 = 1²
```


14

Induction: Sum of Odds

```
P(n) is written as:

   1 + 3 + 5 + ... + (2n – 1) = n²
```

We assume P(n) is true. So, we are assuming that this equality is valid.

```
Now we prove P(n) → P(n + 1)
```


15

Induction: Sum of Odds

```
P(n + 1) is written as:

   1 + 3 + … + (2n–1) + (2n+1) = (n+1)²

Can we show this equality is valid?

Let's look at the left side of the equals …
```


16

Show following equals: **$(n + 1)^2$**

```
1 + 3 + … + (2n – 1) + (2n + 1)

  = 1 + 3 + … + (2n – 1)  + (2n + 1)

  = n² + (2n + 1)

  = (n + 1)²
```

P(n) assumed true, so the equality is true. You can replace!

17

Induction: Sum of Odds

So, we have shown that when P(n) is true, then P(n + 1) is true.

P(n) → P(n + 1)

Since P(1) is true, we have proved $\forall_n$ P(n)

18

Example: Divisible by 3

Using induction…

Show that $n^3 - n$ is divisible by 3 whenever n is a positive integer

Basis: Divisible by 3

- For our basis, we plug 1 into our expression and get the result
- The result, 0, is divisible by 3.

$$P(1) = 1^3 - 1 = 1 - 1 = 0$$

Induction: Divisible by 3

P(n) is written as:

$$n^3 - n$$

P(n + 1) is written as:

$$(n + 1)^3 - (n + 1)$$

Show following is: Divisible by 3

$(n + 1)^3 - (n + 1)$

$= n^3 + 3n^2 + 3n + 1 - (n + 1)$

$= n^3 + 3n^2 + 3n + 1 - n - 1$

$= n^3 + 3n^2 + 3n - n$

$= n^3 - n + 3n^2 + 3n$ *Rearranged*

$= (n^3 - n) + 3(n^2 + n)$

Induction: Divisible by 3

So, for $(n^3 - n) + 3(n^2 + n)$

Since we assumed P(n) is true, then $(n^3 - n)$ is divisible by 3.

… and $3(n^2 + n)$ is divisible by 3 since 3 is a factor

Hence, $P(n) \rightarrow P(n + 1)$

Example: Sum of 2^n

Using induction…

Show that $2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1$ whenever n is a positive integer

Basis: Sum of 2^n

- For our basis, we plug 1 into our expression and get the result
- The result is 1 – which is true

$$P(0) = 2^0 = 1 = 2^1 - 1$$

25

Induction: Sum of 2^n

P(n) is written as:

$$2^0 + 2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$$

P(n + 1) is written as:

(n+1) + 1

$$2^0 + 2^1 + \ldots + 2^n + 2^{n+1} = 2^{n+2} - 1$$

26

Show following equals: $2^{n+2} - 1$

$$\begin{aligned}
&2^0 + 2^1 + \ldots + 2^n + 2^{n+1} \\
&= (2^0 + 2^1 + \ldots + 2^n) + 2^{n+1} \\
&= (2^{n+1} - 1) + 2^{n+1} \\
&= 2^{n+1} + 2^{n+1} - 1 \\
&= 2^n(2^1 + 2^1) - 1 \\
&= 2^n(4) - 1 \\
&= 2^n(2^2) - 1 \\
&= 2^{n+2} - 1
\end{aligned}$$

P(n) assumed true, so the equality is true

27

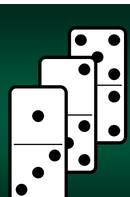
Induction: Sum of 2^n

Since we assumed P(n) is true…

$2^0 + 2^1 + \ldots + 2^n + 2^{n+1}$ **is equal to** $2^{n+2} - 1$

Hence, $P(n) \rightarrow P(n + 1)$

28



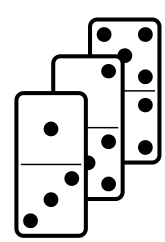
Strong Induction

Another approach

29

Strong Induction

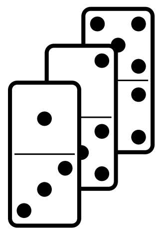
- *Weak* induction assumes that P(*n*) is true, and then uses that to show P(*n* +1) is true
- *Strong* induction assumes P(1), P(2), …, P(*n*) are all true and then uses that to show that P(*n* +1) is true



30

Using the Domino Metaphor…

If all the dominos (1 to n) fell over, will it also have knocked over n+1?



31

Strong Induction

- So, strong induction uses more "dominoes" than weak induction – *which just uses one*
- Both proof techniques are equally valid

```
( P(1) ∧ P(2) ∧ … ∧ P(k) ) → P(k + 1)
```


32

Steps to Proof

- Step 1: *Basis*
  - show the proposition P(1) is true
  - very easy to do – just plug in the values
- Step 2: *Induction*
  - assume that P(1), P(2), …, P(n) are all true
  - show that P(n + 1) is true
  - *or, changing the math slightly: show P(n) is true by assuming P(n – 1), P(n – 2), etc…*

33

Example: Product of Primes

Using strong induction…

Show that any number $n \geq 2$ can be written as the product of primes

34

Basis: Product of Primes

- For 2, we can show that 2 is a product of two primes
- Namely, 1 and itself

```
P(2) = 1 * 2 = 2
```


35

Induction: Product of Primes

- There are two cases for n + 1:
- P(n + 1) is prime
- P(n + 1) is composite
  - it can be written as the product of two composites, a and b
  - where $2 \leq a \leq b < n + 1$

36

Induction: Product of Primes

```
n + 1 prime:

   it is a product itself and 1

n + 1 is composite:

   both P(a) and P(b) are assumed to be
   true

   so, there exists primes where
   a * b = n + 1
```


37

Result

- We showed that any number can be either prime or composite of two numbers
- … and that number holds the same
- As a result, we can keep moving backwards to show that everything must be whittled down to primes

38

Circuits

Part 11

1

Circuits

Boolean Hardware

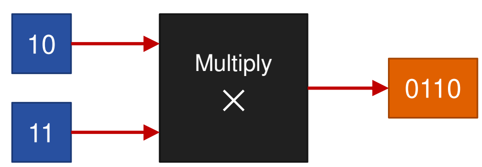
2

Circuits

- Boolean algebra gives designers tools to design & analyze complex solutions
- Can we create hardware that performs solutions?
- Can electronic circuits allow Boolean logic to exist in the physical world?

3

Two Bit Multiplier? Can we make it?



4

Designing It

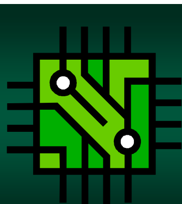
- To design a circuit that multiplies two 2-bit numbers, we can use Boolean algebra
- We need to figure the logic – given that bits of 1 and 0 will map directly to truth values
- The result of the algebra will be the desired output

5

It Takes the Following Skills

1. Design a truth-table to represent the different inputs and the desired output
2. Convert the truth-table into a Boolean function
3. Simplify the Boolean function
4. Finally, convert it into a circuit

6



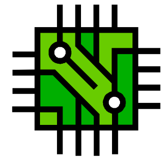
Gates

Boolean Hardware

7

Gates

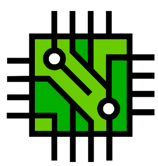
- Electronic devices are made up of *gates*
- Gates take in two inputs and produce a single output
- This is how hardware is used to implement Boolean logic (or any logic)



8

Gates

- Gates can be combined into circuits with *any number of input wires* and a single output wire
- We will chain these together much like subexpressions can be combined into larger expressions



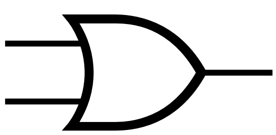
9

Graphical Representation

- Gates are typically represented using graphical shapes – much like flowcharts
- There are two different competing symbol standards
- We will use the standard, distinct, symbols rather than the IEC (European) ones

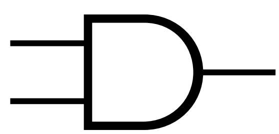
10

Or Gate



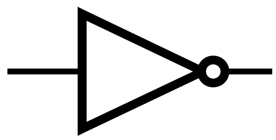
11

And Gate



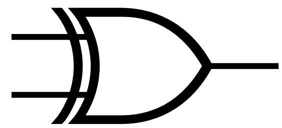
12

Not Gate



13

Exclusive Or Gate (aka XOR)



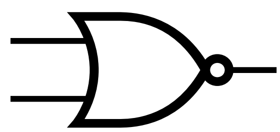
14

Some Other Gate Symbols

- There are also gate symbols for negated operators
- I won't use these much in class, but it's good to be aware of them (since they are quite common in computer engineering
- For each, note the circle on the output line – it means "not"

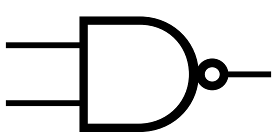
15

Not Or Gate (aka NOR)



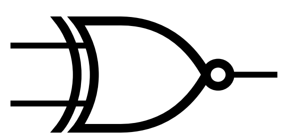
16

Not And Gate (aka NAND)



17

Not Exclusive Or Gate (aka XNOR)



18

Computer Engineering Notation

Same thing, different paint job

19

Computer Engineering Notation

- Gates, used to computer engineering, form a Boolean Algebra
- i.e. they can define the three operations (and, or, not) and share the same axioms

20

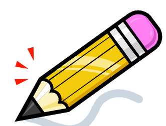
Computer Engineering Notation

- The notation used in computer engineering is a *tad different* that what we have covered
- … and certainly different than any programming language you have used

21

Computer Engineering Notation

- But is serves the same purpose
- And, not surprisingly, it works better for writing expressions in this discipline



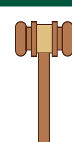
22

Computer Engineering Notation

1	≡	T
0	≡	F
*	≡	∧
+	≡	∨
'	≡	¬

23

Commutative Law



$$x + y \equiv y + x$$

$$x * y \equiv y * x$$

24

Identity Law

$$x * 1 \equiv x$$

$$x + 0 \equiv x$$

25

Complement Law

$$x * x' \equiv 0$$

$$x + x' \equiv 1$$

26

Distributive Law

$$x * (y + z) \equiv (x * y) + (x * z)$$

$$x + (y * z) \equiv (x + y) * (x + z)$$

27

Associative Law

$$(x + y) + z \equiv x + (y + z)$$

$$(x * y) * z \equiv x * (y * z)$$

28

Absorption Law

$$x * (x + y) \equiv x$$

$$x + (x * y) \equiv x$$

29

Idempotent Law

$$x * x \equiv x$$

$$x + x \equiv x$$

30

Involution Law

$x'' \equiv x$

31

Domination Law

$x + 1 \equiv 1$

$x * 0 \equiv 0$

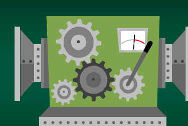
32

DeMorgan's Law

$(x * y)' \equiv x' + y'$

$(x + y)' \equiv x' * y'$

33



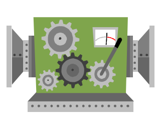
Converting Boolean to Circuits

From Logic to Wires

34

Converting Boolean to Circuits

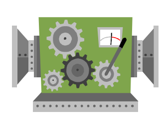
- Converting from Boolean to circuits maintains a *one-to-one* correspondence between gates and operators in the equation
- But, given an *arbitrary* Boolean expression, how do we realize a circuit for it?



35

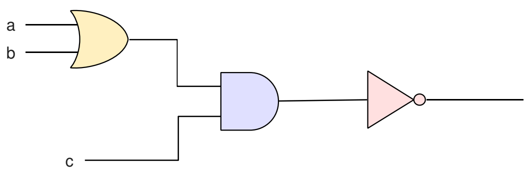
Steps

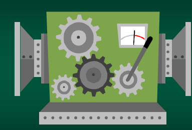
1. Choose the last operation evaluated
2. Draw a gate and hook up its output
3. Goto 1 until all operations have associated gates
4. Attach the expression inputs



36

((a + b) * c)'



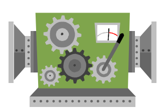


Converting Circuits to Boolean

From Logic to Wires

Converting Circuits to Boolean

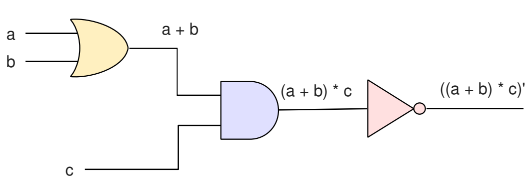
- The other direction is easy too
- Any circuit can be realized as a Boolean expression using the same basic algorithm



Converting Circuits to Boolean

1. Pick a wire that has known Boolean values
2. Write *on the wire* a Boolean expression for its value
3. Goto 1 until all wires are complete
4. Circuit's expression written on the circuit's output wire

Example Circuit

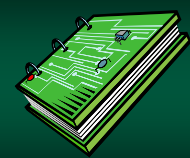


Arbitrary Circuits

Part 12

1

Creating an Arbitrary Circuit

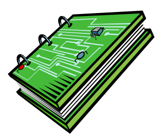


From Truth Table to Wires

2

Creating an Arbitrary Circuit

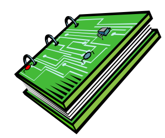
- We converted between Boolean expressions and circuits
- It maintained a one-to-one correspondence between gates in the circuit and operators in the equation



3

Creating an Arbitrary Circuit

- Given an arbitrary logic table, how do we realize a circuit for it?
- Simple, we look at the inputs that make it true, and write them out in an expression using or's.



4

Example: 1 Bit Add Mod 2

5

Example: 1 Bit Add Mod 2

a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

6

Example: 1 Bit Add Mod 2

```
We want a circuit that is true
when:

(a = F and b = T) or
(a = T and b = F)

out = a' * b + a * b'
```


7

Example 2: One Bit Adder

8

Example 2: One Bit Adder

a	b	Out $_1$	Out $_0$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

9

Example: One Bit Adder (Logic)

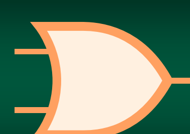
```
out1 = (a = T and b = T)

out0 = (a = F and b = T) or
       (a = T and b = F)
```


10

Example: One Bit Adder (algebra)

```
out1 = a * b

out0 = a' * b + a * b'
```


11



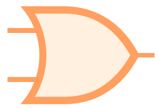
Disjunctive Normal Form

Express Logic With Ease

12

Disjunctive Normal Form

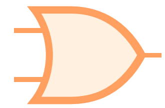
- Best approach to converting tables into circuits is use *Disjunctive Normal Form*
- In this form, the expressions consists of OR's (disjuncts) connecting AND sub-expressions



13

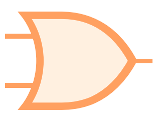
Definitions

- A *literal* is a Boolean variable *v* or its complement (e.g. v *or* v')
- A *minterm* of Boolean product $v_1 * v_2 * \ldots v_n$



14

Definitions



- Hence, a minterm is a “product” of *n* literals, with one literal for each variable
- An equation written only as the "OR" of minterms is in *disjunctive normal form* (also called *sum-of-products* form)

15

Algorithm

1. Find the rows that indicates a 1 for output
   - ignore the ones with 0 as output
   - we are making an equation based on true
2. Write a minterm for each of them
3. "OR" all the minterms

16

Example

a	b	y (out)
0	0	1
0	1	1
1	0	0
1	1	0

17

Example

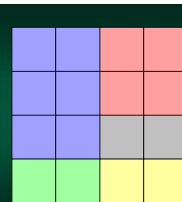
```
DNF of the table is:

y = (a' * b') + (a' * b)

For brevity, for this point on, let's
write as:

y = a'b' + a'b
```


18

Example

```
We can simply using Boolean algebra:

y = a'b' + a'b
  = a' (b' + b)         Distributive
  = a' (1)              Complement
  = a'                  Identity
```


19



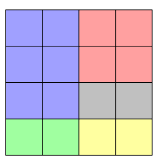
Karnaugh Maps

The Right-Brain Gets to Help

20

Karnaugh Maps

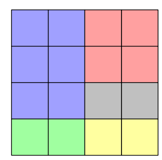
- A *Karnaugh Map* (pronounced "car-no") is a visual tool to help see relations between minterms.
- A K-Map for *n* variables is a grid of 2^n squares



21

Karnaugh Maps

- Every possible minterm of *n* variables is represented
- *Every square is a minterm*
- It is arranged is such a way that we can simplify our table



22

Gray Code

- Literals are ordered using *gray code*
  - values in the table are not ordered in normal ascending order
  - each square differs in exactly one literal
  - why? we will cover this later
- NOTE: squares wrap-around to the sides

23

Two Variable Example

a	b	out
0	0	1
0	1	1
1	0	1
1	1	0

24

Two Variable K-Map

		A: 0	A: 1
B	0	1	1
	1	1	0

Each combination of A

Squares have the output of the circuit

25

Three Variable Example

a	b	c	out
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

26

Three Variable K-Map

		AB: 00	01	11	10
C	0	1	0	1	1
	1	0	0	1	1

27

Four Variable K-Map

		AB: 00	01	11	10
CD	00	0	0	1	1
	01	1	1	0	0
	11	1	1	1	0
	10	0	1	1	0

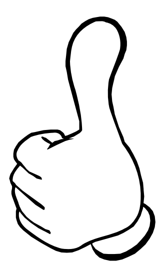
28

How to Use a K-Map

1. Mark the squares of a K-map corresponding to the function
2. Select a minimal set of rectangles where
   - each rectangle has a power-of-two area and is as large as possible
   - cover every marked square
3. Translate each rectangle into a single midterm and sum (or) all these

29

Converting a Rectangle to Minterm



- If any literal contains both 1 and 0, in the rectangle, it is eliminated
- The goal is to draw the **biggest** rectangles possible

30

Example Square: 1×1

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

A' B C' D

31

Example Square: 2×1

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

B C' D

32

Example Square: 1×4

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

C' D

33

Example Square: 2×2

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

A' D

34

Example Square: 2×2: Wrapped

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

B' D

35

Example Square: 2×2: Wrapped

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	0	1
10	1	0	0	1

B' D'

36

Example Square: 4×2

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

D

37

Tips

- There is no magic way to do Step 2. Look and play around until you find the answer
- You can overlap squares – just as long as you "cover" all the 1's

38

Four-Variable K-Map

CD \ AB	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	1	1	1	0
10	0	1	1	0

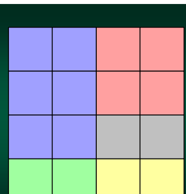
A'D + BC + AC'D'

39

Efficiency of K-Maps

- A K-Map does not necessarily make the *best* expression/circuit
- All expressions made this way are sums-of-products and some can be made simpler
- For example: a(b+c) is the same as ab+ac, but uses fewer gate inputs

40



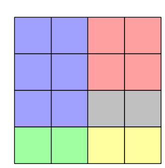
How K-Maps Work

You are doing more than you think

41

How K-Maps Work

- The order of gray code, and the 2^n squares allow us to factor out literals
- Every time you eliminate a literal, you are performing **three** Boolean algebra laws
- This is done visually, so it is invisible!



42

How K-Maps Work

1. First you use the *Distribution Law* on the minterms leaving **(v + v')** - which is the terminal that *changed*
2. You then use the *Complement Law* on **(v + v')** leaving 1
3. Finally, you remove the 1 using the *Identity Law*

43

Let's Look at This Again...

CD \ AB	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	1	1	1	0
10	0	1	1	0

ABC'D' + AB'C'D'

B changes

44

Let's Look at This Again...

```
ABC'D' + AB'C'D'

AC'D'(B + B')      Distributive

AC'D'(1)           Complement

AC'D               Identity
```


45

How About Another Rectangle?

CD \ AB	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	1	1	1	0
10	0	1	1	0

Both A and D change

A'BCD + ABCD +
A'BCD' + ABCD'

46

How About Another Rectangle?

```
A'BCD  + ABCD + A'BCD' + ABCD'

BCD(A' + A) + BCD'(A' + A)

BCD(1) + BCD'(1)

BCD + BCD'
```

A eliminated

47

… and it keeps going…

```
BCD + BCD'

BC(D + D')

BC(1)

BC
```

D eliminated

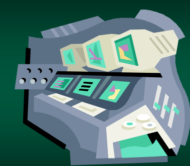
48

Please Wait

CSC 28
will begin shortly

Please open the chat window.

49



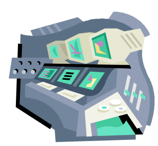
K-Maps and Programming

Using it to simplify code

50

K-Maps and Programming

- The Boolean expressions, that you use in your Java programs, are the same as the expressions we cover
- So, you can apply K-Maps to your Java code to simplify expressions



51

K-Maps Can Simplify Expressions

- The following is a complex expression that, on the surface, looks difficult to simplify
- K-Maps can help

```
if (a && !b && c  || a && b && !c ||
    a && !b && !c || a && b && c)
```


52

K-Maps Can Simplify Expressions

- First, let's put the expression in the Computer Engineer notation
- Ah, we can see the structure now!

```
ab'c' + abc' + ab'c + abc
```


53

K-Maps Can Simplify Expressions

ab

c	00	01	11	10
0				
1				

ab'c' + abc' + ab'c + abc

54

K-Maps Can Simplify Expressions

ab

c \ ab	00	01	11	10
0				1
1				

ab'c' + abc' + ab'c + abc

55

K-Maps Can Simplify Expressions

ab

c \ ab	00	01	11	10
0			1	1
1				

ab'c' + abc' + ab'c + abc

56

K-Maps Can Simplify Expressions

ab

c \ ab	00	01	11	10
0			1	1
1				1

ab'c' + abc' + ab'c + abc

57

K-Maps Can Simplify Expressions

ab

c \ ab	00	01	11	10
0			1	1
1			1	1

ab'c' + abc' + ab'c + abc

58

K-Maps Can Simplify Expressions

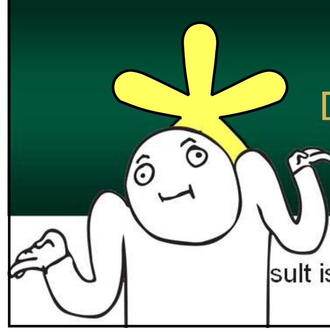
ab

c \ ab	00	01	11	10
0			1	1
1			1	1

ab'c' + abc' + ab'c + abc = a

59

Don't Care

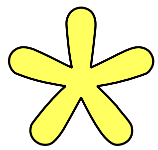


sult is Meaningless

60

Don't Care

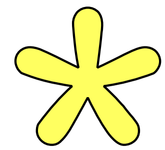
- Sometimes *we don't really care* what output the circuit generates for some combinations of inputs
- So, for those inputs, the results are simply not significant



61

Don't Care

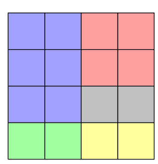
- In truth tables, the value "Don't Care" is represented with an asterisk
- It can be considered True or False – whichever is more *convenient* for the circuit



62

Karnaugh Maps and Don't Care

- We can construct a Karnaugh Map like before
- Except the squares corresponding to don't care outputs are marked (with an asterisk)



63

Karnaugh Maps and Don't Care

- Then, when outlining blocks, we can (at our convenience) consider the "don't care" squares as either 0 or 1
- Since we want to make the largest outlines possible, we will sometimes consider a don't care to be true, and sometimes false

64

Example

- We want to guarantee that the output of a circuit is 1 if both inputs are 1
- And 0 when both inputs are 0
- But otherwise we do not care

65

Example

a	b	out
0	0	0
0	1	*
1	0	*
1	1	1

66

K-Map For The Example

	A = 0	A = 1
B = 0	0	*
B = 1	*	1

We don't need B!

out = A

67

… or we can do this

	A = 0	A = 1
B = 0	0	*
B = 1	*	1

Just B!

out = B

68

Four-Variable (with Don't Care)

CD \ AB	00	01	11	10
00	0	1	1	*
01	1	1	*	0
11	1	1	1	0
10	*	1	1	0

out = B + A'D

69

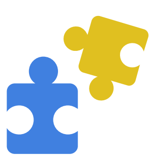
Functional Completeness

Just How Much Do We Need?

70

Functional Completeness

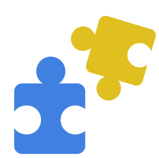
- We can construct a circuit for any Boolean expression using and / or / not
- This means the set of gates {and, or, not} is *functionally complete*



71

Function Completeness

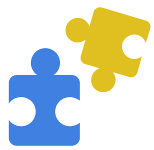
- However, we don't need all three gates
- DeMorgan's laws shows us that we can construct:
  - an OR using an AND
  - and AND using an OR



72

We Don't Need Or!

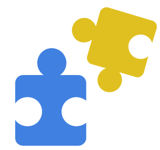
- So {and, not} are also complete because by DeMorgan's Law: $x + y = (x'y')'$
- So, any expression that can be written using {and, or, not} can be written using just {and, not}



73

or… We Don't Need And!

- Also {or, not} is functionally complete since $xy = (x'+y')'$
- So, any expression that can be written using {and, or, not} can be written using just {or, not}



74

Functional Completeness

- So, are any of the singular sets {and}, {or}, {not} functionally complete?
- In other words, can and/or/not all be converted into a single type of gate?
- **No.** Neither {and} or {or} can be converted to a {not}

75

NAND

- So, is there a gate that can, alone, be functional complete?
- What about NAND (negated And)?
  - x nand y = $(xy)'$
  - Note: the NAND gate is not implemented with an AND gate and a NOT gate. It just has the same truth table as $(xy)'$

76

NAND

- To show that {nand} is functionally complete, we need to show that we can implement {and, or, not} using it
- The result would be greatly beneficial!
  - we would have to just construct 1 gate to create any circuit
  - this would greatly aid construction

77

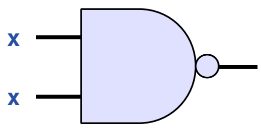
Not → Nand

```
Converting not to nand:

x'  =  x'
    =  (xx)'          Idempotent
    =  x nand x       nand format
```

We can implement NOT by using a NAND. Both input will be x

78

Not → Nand



79

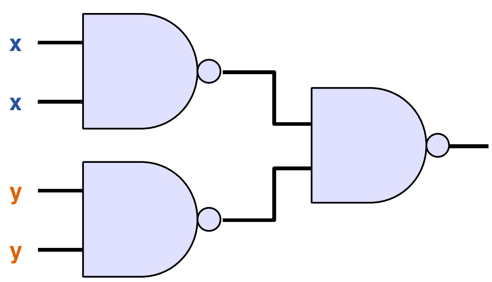
Or → Nand

```
Note: x' = x nand x

x + y =  x + y
      =  (x'y')'                DeMorgan
      =  x' nand y'             nand format
      =  (x nand x) nand (y nand y)
```

Last proof let us convert NOT into NAND

80

Or → Nand



81

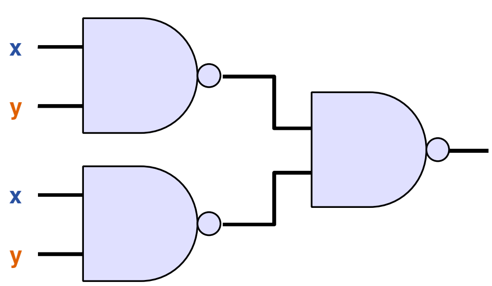
And → Nand

```
Note: x' = x nand x

xy    =  xy
      =  ( (xy)')'              Involution
      =  (x nand y)'            Negate nand
      =  (x nand y) nand (x nand y)
```

Last proof let us convert NOT into NAND

82

And → Nand



83

Summary

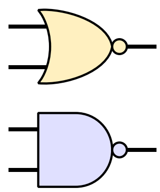
- The expressions below show that nand can be used to implement NOT, OR, AND
- So, we can just use NAND since it is *functionally complete*

```
x'    =  x nand x
xy    =  (x nand y) nand (x nand y)
x + y =  (x nand x) nand (y nand y)
```


84

How Hardware Works

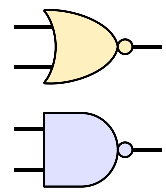
- Also NOR is functionally complete
- P NOR Q = (P + Q)'
- Hardware can alternatively use this gate rather than NAND



85

How Hardware Works

- If our hardware can just implement NAND or NOR, then we can create a circuit with just <u>one gate</u>
- In fact, many fabrication processes use only NAND or NOR gates



86