

# Discrete Structures in Python

Ganesh Gopalakrishnan, Bruce Bolick, Charles Jacobsen, Tony Tuttle

December 8, 2013



# Contents

<b>0 Python Basics</b>	<b>1</b>
0.1 Workspace Setup . . . . .	1
0.2 The Python Shell (REPL) . . . . .	2
0.3 Numbers, Strings, & Booleans . . . . .	2
0.4 Lists, Tuples, Dictionaries, & Sets . . . . .	3
0.5 Writing a Script . . . . .	5
0.6 Objects & Variables . . . . .	5
0.7 Control Flow & List Comprehensions . . . . .	7
0.8 Functions & Modules . . . . .	10
0.8.1 Functions . . . . .	10
0.8.2 Modules . . . . .	13
0.9 Functional Programming . . . . .	15
0.9.1 Functions are Values . . . . .	15
0.9.2 Lambdas . . . . .	16
0.9.3 Higher Order Functions . . . . .	18
0.10 Recursion . . . . .	22
<b>1 Propositional/Boolean Logic</b>	<b>27</b>
1.1 Historical Perspective . . . . .	28
1.1.1 History of Boolean Functions in Computing . . . . .	28
1.1.2 History of Digital Design and Circuit Simulation . . . . .	28
1.1.3 History of Propositional Reasoning . . . . .	29
1.2 Truth Values and Functions . . . . .	30
1.2.1 Boolean Functions . . . . .	30
1.2.2 Case Study: the Nand Function Represented . . . . .	35
1.2.3 How Many Functions over One Input Exist? . . . . .	37
1.2.4 How Many Functions over Two Inputs Exist? . . . . .	37
1.2.5 Counting All Possible $N$ -input Functions . . . . .	37

<b>2 Gate Types, Boolean Identities</b>	<b>41</b>
2.1 Syntax for Boolean Formulae . . . . .	41
2.2 Boolean Identities . . . . .	41
2.3 Uses of Universal Gate Sets . . . . .	43
2.3.1 Realizing Inclusive and Exclusive Or using Nands . . . . .	44
2.3.2 An “Implication Gate” . . . . .	44
2.3.3 A 2-to-1 Mux Using Nand Gates . . . . .	45
2.4 Realizing all 2-input Functions via Muxes . . . . .	45
2.4.1 A 2-to-1 Multiplexor or an if-then-else gate . . . . .	46
2.4.2 Creation of a general 2-input function module . . . . .	49
2.4.3 Counting Boolean Functions: the Mux Approach . . . . .	52
<b>3 Review of Boolean Stuff</b>	<b>55</b>
3.1 Short-circuiting Evaluation . . . . .	55
3.2 Short-circuiting aids Programming . . . . .	56
3.3 Boolean Expression Evaluation . . . . .	56
3.3.1 Expressions and their Complements . . . . .	57
3.4 History Readings . . . . .	57
3.4.1 Boole . . . . .	57
3.4.2 Shannon . . . . .	57
3.4.3 Boolean Algebra versus Propositional Calculus . . . . .	58
3.4.4 Errors in Microprocessors . . . . .	58
3.4.5 Futility of Brute-Force Testing . . . . .	58
3.5 Practice Defining Gates . . . . .	58
3.6 Nand using Nor . . . . .	58
3.7 Nor Using Nand . . . . .	59
3.8 Counting Multi-output Functions . . . . .	59
3.9 Xor Using Nor . . . . .	60
3.10 Universality Proofs . . . . .	61
3.11 Universal Gates . . . . .	61
3.11.1 Alternative Proof . . . . .	62
3.11.2 Showing NAND and NOR to be Universal . . . . .	62
3.11.3 Showing that a gate is NOT Universal . . . . .	63
3.12 Boolean Identities and Simplification . . . . .	64
3.12.1 A Simple Identity . . . . .	64
3.12.2 Boolean Simplification . . . . .	64

<b>CONTENTS</b>	<b>5</b>
<b>4 Sets</b>	<b>65</b>
4.1 All of Mathematics Stems from Sets . . . . .	66
4.2 Cardinality (or “size”) of sets . . . . .	66
4.3 Operations on Sets . . . . .	67
4.4 Comprehension, Characteristic Predicates . . . . .	69
4.4.1 Set Comprehension in Python . . . . .	70
4.5 Cartesian Product . . . . .	71
4.5.1 Cardinality of a Cartesian Product . . . . .	71
<b>5 Venn Diagrams, and Proofs of Set Identities</b>	<b>73</b>
5.1 Venn Diagrams . . . . .	74
5.1.1 The number of regions in a Venn diagram . . . . .	75
5.1.2 Alternative Derivation: Number of Venn Regions . . . . .	78
5.2 Formal Proofs of Set Identities . . . . .	78
5.2.1 Checking the Proofs Using Python . . . . .	82
<b>6 Review of Sets</b>	<b>85</b>
6.1 Exercises Pertaining to Sets . . . . .	85
6.2 Set Comprehension involving Booleans . . . . .	86
6.2.1 Comprehension Example . . . . .	86
6.2.2 One More Comprehension Example . . . . .	86
6.3 Defining Primes via Recursion . . . . .	87
6.4 More Examples . . . . .	90
6.5 A Boolean, Set, and Lambda Wringer . . . . .	92
6.6 A Useful Set Identity . . . . .	94
6.6.1 Set Identity $A \cup (B - C)$ . . . . .	94
6.6.2 Checking the proof in §6.6.1 . . . . .	94
<b>7 Systematic Design of Gate Networks</b>	<b>95</b>
7.1 The Disjunctive Normal Form . . . . .	97
7.2 DNF Realizations using And/Or and Nand/Nand . . . . .	97
7.2.1 The And/Or form of the DNF equation $out = m + (a.!b + !a.b)$ : . . . . .	99
7.2.2 The Nand/Nand form of the DNF equation $out = m + (a.!b + !a.b)$ : . . . . .	99
7.2.3 “Bubble Pushing” for Circuit Inter-conversion . . . . .	99
7.3 The If-Then-Else Normal Form . . . . .	100
7.4 The ITENF form of $out = m + (a.!b + !a.b)$ . . . . .	101
7.5 The Conjunctive Normal Form . . . . .	103

<b>8 Propositional Proofs</b>	<b>107</b>
8.1 The notion of Proving . . . . .	107
8.2 Proving Propositional Logic Formulae . . . . .	109
8.2.1 Truth-table method . . . . .	109
8.2.2 Propositional Identities . . . . .	109
8.2.3 Showing Bi-implication . . . . .	110
8.3 Modeling and <i>then</i> Proving . . . . .	110
8.4 Proof by Contradiction . . . . .	112
8.5 Z3Py: A Proof Assistant . . . . .	115
8.6 Lewis Carroll : Young Wise Pigs . . . . .	117
8.7 A Circuit Interpretation of Babies and Crocodiles . . . . .	120
8.8 Inference Rule Summary; More Examples . . . . .	121
8.8.1 Inference Rules . . . . .	121
8.8.2 More Examples . . . . .	123
<b>9 Midterm Review</b>	<b>127</b>
9.1 Sets . . . . .	127
9.1.1 Operators . . . . .	127
9.1.2 Proof of $((A - C) \cup (B - C)) \subseteq \overline{C}$ . . . . .	128
9.2 Functions . . . . .	128
9.2.1 Calculating the Number of Ternary Functions . . . . .	129
9.2.2 Universal Ternary Gates? . . . . .	130
9.3 Boolean Algebra and Propositional Logic . . . . .	130
9.3.1 Boolean Simplification 1 . . . . .	130
9.3.2 Boolean Simplification 2 . . . . .	131
9.3.3 If-Then-Else Normal Form . . . . .	131
9.4 Proofs, and What They Mean . . . . .	131
9.4.1 Validity (or Tautology) and Satisfiability . . . . .	131
9.4.2 Rules of Inference . . . . .	131
9.5 Miscellaneous . . . . .	132
9.5.1 Connections to Practice . . . . .	132
9.5.2 For extra insight into the material . . . . .	132
9.5.3 All the ways to implement 0 and 1 . . . . .	132
9.5.4 Historic Video from the MULTICS Era . . . . .	133
9.6 Midterm-1 and Its Solution . . . . .	133

<b>10 Karnaugh Maps</b>	<b>139</b>
10.0.1 Example of K-Map Usage . . . . .	140
10.0.2 There are $2^{2^3}$ 3-variable K-maps! . . . . .	143
10.0.3 Generalized Simplification . . . . .	143
10.0.4 K-maps and Gray Code Sequences . . . . .	144
<b>11 Binary Decision Diagrams</b>	<b>149</b>
11.1 BDD Basics . . . . .	150
11.1.1 BDD Guarantees . . . . .	152
11.1.2 BDD-based Comparator for Different Variable Orderings	152
11.1.3 BDDs for Common Circuits . . . . .	152
11.1.4 A Little Bit of History . . . . .	156
11.2 Reading CNF and DNF Off BDDs . . . . .	157
11.3 Designing using BDD . . . . .	160
11.3.1 Map Coloring: Canadian Map . . . . .	160
11.3.2 Walking a Grid using BDDs . . . . .	162
11.4 Solving Puzzles using BDDs . . . . .	163
11.5 Review and Labs . . . . .	169
<b>12 Graphs</b>	<b>171</b>
12.1 Undirected Graphs, and Graph Coloring . . . . .	171
12.1.1 Map Coloring . . . . .	171
12.2 Brute-force Solving of Graph Coloring . . . . .	173
12.3 Constraint-based Solving . . . . .	173
12.4 Applications: Scheduling and Coloring . . . . .	176
12.4.1 Application of Coloring: Committee Scheduling . . . . .	176
12.4.2 Application of Coloring: Register Allocation . . . . .	178
<b>13 Basic First Order Logic</b>	<b>181</b>
13.1 Universe, Individuals, and Predicates . . . . .	181
13.2 Individuals, Quantifiers over them . . . . .	182
13.3 Quantifiers over a Domain . . . . .	184
13.3.1 A Helpful Example to Understand Quantifiers . . . . .	187
13.3.2 Illustration on Fermat's Last Theorem . . . . .	187
13.4 Nonplussed Barbers, Non-quack Doctors . . . . .	188
13.4.1 Keep your hair on! . . . . .	188
13.4.2 If it quacks like a duck, it ain't a doctor . . . . .	189
13.5 Prolog, Canadian Map! . . . . .	190

13.5.1 Prolog Basics . . . . .	190
13.5.2 Maps of Canada . . . . .	191
13.5.3 Logical Basics of Prolog . . . . .	194
13.6 <i>Review and Labs</i> . . . . .	196
<b>14 Relations</b>	<b>201</b>
14.1 Binary Relations . . . . .	201
14.1.1 Types of binary relations . . . . .	202
14.1.2 Preorder (reflexive plus transitive) . . . . .	206
14.1.3 Partial order (preorder plus antisymmetric) . . . . .	206
14.1.4 Total order, and related notions . . . . .	207
14.1.5 Relational Inverse . . . . .	207
14.2 Equivalence (Preorder plus Symmetry) . . . . .	208
14.2.1 Intersecting a preorder and its inverse . . . . .	208
14.2.2 Identity relation . . . . .	209
14.2.3 Universal relation . . . . .	209
14.2.4 Equivalence class . . . . .	209
14.2.5 Reflexive and transitive closure . . . . .	210
14.2.6 Illustration of Transitive Closure Using Prolog . . . . .	211
14.3 Powersets . . . . .	211
14.3.1 Application: Electoral Maps . . . . .	214
14.4 The <i>Power</i> Relation between Machines . . . . .	214
14.4.1 The equivalence relation over machine types . . . . .	216
14.5 <i>Review and Labs</i> . . . . .	218
<b>15 Mastermind Game Using Logic</b>	<b>221</b>
15.1 One Popular Version of Mastermind . . . . .	221
15.1.1 Formulation of Scoring: Approach 1 . . . . .	222
15.1.2 Formulation of Scoring: Approach 2 . . . . .	224
15.2 A Simple Mastermind Program . . . . .	225
15.2.1 Usage Scenarios . . . . .	225
15.2.2 Mastermind: Preliminary Implementation . . . . .	230
15.3 Project Specification . . . . .	235
15.3.1 Extra Credit Items . . . . .	235
<b>16 Recursion and Induction</b>	<b>237</b>
16.1 What is Induction? How many kinds? . . . . .	239
16.1.1 Arithmetic Induction . . . . .	241

<b>CONTENTS</b>	<b>9</b>
-----------------	----------

16.1.2 Complete Induction . . . . .	242
16.1.3 Structural Induction . . . . .	242
16.1.4 Subgoal Induction . . . . .	242
16.2 How to present inductive proofs? . . . . .	243
16.2.1 Case 1: When you have a program to stare at . . . . .	243
16.2.2 Case 2: When you are doing it just over “data” . . . . .	243
<b>17 Permutations and Combinations</b>	<b>245</b>
17.1 Illustrations on Using Venn Diagrams . . . . .	247
17.1.1 Venn Diagram Regions and the Binomial Theorem . . . . .	248
17.1.2 Alternative Derivation . . . . .	249
17.2 Illustration in the Context of Powersets . . . . .	249
<b>18 Probability</b>	<b>253</b>
18.1 Basic Definitions with Illustrations . . . . .	255
18.1.1 One Die: Sample Space, Elementary Outcomes, Events, Probabilities . . . . .	255
18.1.2 Events . . . . .	256
18.1.3 Event Probability . . . . .	256
18.1.4 Defining New Events Using Set Operations . . . . .	256
18.1.5 Independent Events . . . . .	257
18.1.6 Illustrations on One Die Tossed Twice . . . . .	258
18.2 Conditional Probability . . . . .	260
18.2.1 Bayes’ Rule . . . . .	260
18.3 Illustration: Medical Testing . . . . .	261
18.4 Applications: Birthday “Paradox” . . . . .	261
18.4.1 Python-based Evaluation of Probabilities . . . . .	262
18.5 Conditional Probabilities thru Gobblers . . . . .	264
<b>19 Lecture Notes</b>	<b>267</b>
19.1 K-maps . . . . .	267
19.2 Chapter on BDDs . . . . .	269
19.3 First Order Logic . . . . .	270
19.3.1 Exercises . . . . .	274
19.4 Relations: Chapter 14 . . . . .	275
19.5 How to think about induction? . . . . .	278
19.5.1 Induction: An Approach to Proving General Facts . . . . .	278
19.5.2 Taking Stock of $\forall x : good(x)$ . . . . .	279

19.5.3 Inductive Proof of $\forall x: good(x)$ . . . . .	280
19.5.4 Induction Principles in Second Order Logic . . . . .	281
19.5.5 Connection Between Induction and Recursion . . . . .	281
19.5.6 Example: Proving that $\sqrt{2}$ is Irrational . . . . .	282
19.5.7 Proving a Recursive $gcd(x, y)$ Correct . . . . .	283
19.5.8 McCarthy's "91 function" . . . . .	285
19.5.9 Collatz's Conjecture or "3n+1" Problem . . . . .	285
19.5.10 Connection With Termination Arguments . . . . .	290

# Chapter 0

## Python Basics

This chapter is a survival guide for setting up your workspace and programming in Python (2.7). A broad spectrum of programs can be written in Python, from quick scripts to large server backends. We will use Python to apply the mathematical concepts we will be studying and explore some cutting-edge tools.

Python is concise and easy to read—proper indentation is required—and the documentation and resources for Python is outstanding. We will provide references and links throughout this chapter that expand on certain points. We enjoy programming in Python, and hope you will too.

### 0.1 Workspace Setup

IDLE is the recommended programming environment for Python for students with little Emacs or Vim experience. To obtain Python 2.7 and IDLE, visit [www.python.org/getit](http://www.python.org/getit) and download the correct package for your platform. IDLE should be in the package.

IDLE is available in the CADE lab and Undergrad lab.

We encourage students to learn Emacs or Vim and set up their own workspace, preferably on a Linux or OS X platform, because the experience is invaluable. We can provide some limited help and guidance, but you will need to set aside ample time.

We will show you how to use Python and IDLE in the first lab.

## 0.2 The Python Shell (REPL)

Locate and open IDLE. IDLE will fire up a Python shell when initialized. You will see a prompt, waiting for a snippet of Python code. Type in `5 + 3` and hit enter; the shell prints the result, `8`.

Another name for the Python shell is a *read-eval-print-loop* (REPL): It reads a snippet of Python code, evaluates it using the Python interpreter, prints the result, and prompts for the next snippet of code. You will find this convenient for quickly testing pieces of code while you write programs in Python. Execute `13 if 2 < 1 else -13`; the shell prints `-13`.

## 0.3 Numbers, Strings, & Booleans

All standard operations on numbers are available. Strings are written using either single ('), double (""), or triple ("""") quotes. Single and double quotes are interchangeable unless the contained string contains a conflicting quote character. Triple quotes are used for function and module docstrings (see below) and for strings that contain multiple lines. For example,

```
>>> "I'm not exaggerating."
"I'm not exaggerating."
>>> """this little piggy had roast beef
... this little piggy had none"""
'this little piggy had roast beef\nthis little piggy
had none'
```

Strings can be concatenated with `+`, like Java; substrings can be extracted using Python's elegant slice notation; string length can be calculated with `len`:

```
>>> "this" + " little" + " piggy" + " went" + " to" + \
" market"
'this little piggy went to market'
>>> "this little piggy stayed home"[0:4]
'this'
>>> "this little piggy stayed home"[12:17]
'piggy'
>>> "this little piggy stayed home"[-4:]
'home'
>>> len("cellar")
6
```

The constants `True` and `False` are Booleans and can be combined using `not`, `and`, `and or`:

```
>>> True or False
True
>>> True and (not 2 < 1)
True
```

See [An Informal Introduction to Python](#) for a great overview of numbers and strings, and [Boolean Operations](#) for more details about Booleans.

## 0.4 Lists, Tuples, Dictionaries, & Sets

Lists behave like arrays and can contain a mixture of data (including lists). Lists are written using square brackets and elements are accessed using the index and slice notation:

```
>>> [1, 2, "buckle", "my", "shoe"]
[1, 2, 'buckle', 'my', 'shoe']
>>> [3, 4][1]
4
>>> [[1, 2], [3, 4, 5], [6, 7, 8, 9]][1]
[3, 4, 5]
>>> [[1, 2], [3, 4, 5], [6, 7, 8, 9]][1][2]
5
>>> [[1, 2], [3, 4, 5], [6, 7, 8, 9]][1:]
[[3, 4, 5], [6, 7, 8, 9]]
```

List length can also be calculated with `len`. Tuples behave like lists, but are written using parentheses instead of square brackets; the differences between tuples and lists is beyond the scope of this chapter.

Dictionaries map keys of arbitrary type<sup>1</sup> to values and are written using curly braces. A value is accessed using index notation with the correct key:

```
>>> {"zero":0, "one":1, "two":2}
{'zero': 0, 'two': 2, 'one': 1}
>>> {"zero":0, "one":1, "two":2}["one"]
1
```

A dictionary's keys, values, and key-value pairs (as tuples) are accessed using the `keys`, `values`, and `items` methods of the dictionary type:

```
>>> {"zero":0, "one":1, "two":2}.keys()
['zero', 'two', 'one']
>>> {"zero":0, "one":1, "two":2}.values()
[0, 2, 1]
>>> {"zero":0, "one":1, "two":2}.items()
[('zero', 0), ('two', 2), ('one', 1)]
```

---

<sup>1</sup>The keys must be immutable (e.g., they cannot be lists).

See [Data Structures](#) for a great overview of these four types. [An Informal Introduction to Python](#) also has some additional information on lists.

## 0.5 Writing a Script

Python programs are written in script files and passed through the Python interpreter to execute them. The interpreter reads each line and evaluates it, as if it had been read from the shell; this means there is no need to enclose the program in any extra boilerplate. A script file can be created with any text editor.

In IDLE, go to the File menu and choose New Window. Inside the file, type the command `print "Hello world!"`. Save the file under any name with a `.py` extension, then press F5 to execute it (or choose Run > Run Module from the menu). `Hello world!` should be printed in the shell. Wow, wasn't that easy for a hello world program? Try experimenting with some other simple commands, `print 2 + 3 + 4`.

Comments are written using `#` and all characters on the same line and after the `#` will be ignored.

## 0.6 Objects & Variables

Variables are easy to use in Python. Compared to C/C++ and Java, variables are used and treated a lot differently in Python, but the technical details aren't important for the everyday programmer<sup>2</sup>. Here is a rough guide to get you started; it should be all you need to know:

- Everything is an object in Python (including numbers and Booleans).
- Some objects are mutable (lists and dictionaries), others are not (numbers, strings, Booleans, tuples).
- Roughly, all variables ‘store a reference’ to an object.
- Variables are not declared, and variables are used without type annotations. Compare:

---

<sup>2</sup>The interested reader should see [Objects, values, and types](#) and [Naming and binding](#) for more technical details. You may want to compare this with Java's specification, [Variables](#) and [Run-Time Data Areas](#).

**Python:**

```
s = "xddata2"
```

**Java:**

```
// (could collapse in one line)
string s;
s = "xddata2";
```

- A variable can refer to objects of various types throughout the program, and the type of object a variable refers to may remain unknown until the program is executed! For example,

```
c = raw_input()
s = "I'm a string!"
if c == "list":
    s = ["Now I'm a list with a string!"]
else:
    s = { "m": "Now I'm a dict with a string!" }
# What type of object does s refer to here?
```

- Global variables are not visible unless they are flagged with the `global` keyword (but we discourage the use of global variables!). See the section on functions.

For example, inside the shell,

```
>>> x = [1,0,1]
>>> x = 2
>>> y = [1,2,3]
>>> z = y
>>> z[0] = -150
>>> z
[-150, 2, 3]
>>> y
[-150, 2, 3]
```

The following is a common error; it's legal in Java but not in Python:

```
>>> a = 5
>>> a + " + five = 10"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and
'str'
```

We can use `str` to convert the integer referred to by `a` to a string:

```
>>> str(a) + " + five = 10"
'5 + five = 10'
```

**EXERCISE.** Write a script that reads a temperature in Celsius and prints out the temperature in Fahrenheit. You can use `raw_input` to read in the temperature and `float` to convert strings to floats. Don't worry about user errors. To convert from Celsius to Fahrenheit, use the equation  $F = \frac{9}{5}C + 32$ .

## 0.7 Control Flow & List Comprehensions

Here is a side-by-side comparison with Java:

### Python:

```
# if/else

if x == 3:
    print "x is 3!"
else:
    print "x is not 3!"
```

### Java:

```
// if/else

if (x == 3) {
    System.out.println("x is 3!");
} else {
    System.out.println("x is not 3!");
```

```
# for // for
sum = 0
for i in range(10):
    sum += i
print sum
int sum = 0;
for (int i = 0; i < 10; i++) {
    sum += i;
}
System.out.println(sum);

# while // while
sum = 0
i = 0
while i < 10:
    sum += i
    i += 1
print sum
int sum = 0, i = 0;
while (i < 10) {
    sum += i;
    i++;
}
System.out.println(sum);
```

Unlike Java, proper indentation is *required* in Python. Statements inside a block are indented one level deeper than the outside. See [Control Flow](#) for more information.

Notice the `range` function in the `for` loop; in general, `range(n)` produces a list, `[0, 1, 2, ..., n - 1]`. So, the `for` loop is more similar to Java's `for-each` loop. Another example will illustrate:

**Python:**

```
x = [2, 0, 1, 3]
for n in x:
    print n
```

**Java:**

```
int[] x = new int[]{2, 0, 1, 3};
for (int n : x) {
    System.out.println(n);
}
```

(Like Java, this works for all kinds of containers and general objects that conform to certain interfaces. For those interested, see [Iterators](#) and [Generators](#).)

List comprehensions, while not control flow statements, compact and replace loops for building lists. To build a list of squares, we could use

```
squares = []
for i in range(6):
    squares[i] = i * i
print squares
```

Alternatively, we can use a list comprehension and accomplish the same thing in one line:

```
squares = [i * i for i in range(6)]
print squares
```

Here are some other examples:

```
>>> [i for i in range(10) if i % 2 == 0]
[0, 2, 4, 6, 8]
>>> [[i,j] for i in range(3) for j in range(3)]
[[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2], [2, 0],
 [2, 1], [2, 2]]
```

See [List Comprehensions](#) for more examples and information.

**EXERCISE.** Write a script that prompts for a path to a file and prints the number of characters and lines in the file. You should read [Reading and Writing Files](#) for tips on how to get started.

## 0.8 Functions & Modules

### 0.8.1 Functions

Here is a side-by-side comparison again with Java:

#### Python:

```
def square(x):
    return x * x
```

#### Java:

```
public static int square(int x) {
    return x * x;
}
```

In general,

- the return type and parameter types are not specified
- indenting the block inside the function is required
- a function need not be a class method<sup>3</sup>

Here is a more complicated example to illustrate some other nuances. It contains a simple representation for a library of books and common operations. Type this in a script and try it out.

---

<sup>3</sup>Python does have classes, but we won't create any classes of our own. For those interested, see [Classes](#).

```
# A library of books is represented using a dictionary
# mapping book titles, as strings, to availability,
# as a Boolean. True means the book is checked in,
# False checked out.

numBooks = 0

def addNewBook(library, bookTitle):
    """ Adds bookTitle to the library dictionary, and
    sets the new book's status to checked in. Updates
    count of books in library. """
    global numBooks
    library[bookTitle] = True
    numBooks += 1

def checkoutBook(library, bookTitle):
    """ Confirms the book is currently checked in; sets
    the status to 'checked out'."""
    assert( library[bookTitle] == True )
    library[bookTitle] = False

def checkinBook(library, bookTitle):
    """ Confirms the book is currently checked out; sets
    the status to 'checked in'."""
    assert( library[bookTitle] == False )
    library[bookTitle] = True

def findBook(library, bookTitle):
    """ Returns true if the library has bookTitle
    and it is checked in, false otherwise. """

    # Loop through key-value pairs in library
    for (title, status) in library.items():
        # Check if title matches and book's status is
        # checked in
        if title == bookTitle:
            if status == True:
                return True
            else:
                return False

    # Loop terminated; book wasn't found
    return False
```

```
# Main entry point for the script file. This
# is not required, but makes the script look more
# like the source code for other languages. Notice
# the call to main at the very bottom.
def main():

    global numBooks

    # Create a new empty library
    lib = {}

    # Add some books
    addNewBook(lib, "Our Man in Havana")
    addNewBook(lib, "Through the Looking Glass")
    addNewBook(lib, "The Time Machine")

    # Check count of books
    print ( "Total number of books in library..." +
           str(numBooks) )

    # Find a book
    print "Searching for The Time Machine...",
    if findBook(lib, "The Time Machine"):
        print "available."
    else:
        print "unavailable."

    # Check out a book
    checkoutBook(lib, "Through the Looking Glass")

    # Confirm it is not found
    print "Searching for Through the Looking Glass...",
    if findBook(lib, "Through the Looking Glass"):
        print "available."
    else:
        print "unavailable."

    # Call main
    main()
```

## Discussion

- The interpreter parses the function definitions, starting from the top. By the time it reaches the call to `main`, `main` has been defined, so it executes without error.
- In order to use the global variable `numBooks` inside a function, it must be flagged with `global`.
- The strings in triple quotes are docstrings that document the function. A user of your module can print your docstring using `help` (see the section on modules below).
- `assert` throws an exception<sup>4</sup> if the test evaluates to false.
- The `lib` dictionary in `main` is modified inside each function, and the results are visible outside the function. (This is just like Java, but the technical details are different.)

### 0.8.2 Modules

A single script file is a module. Modules can be imported into other modules and even into the Python shell so that functions and variables can be re-used or quickly tested. The most important thing to know for now is the `import` statement for importing a module. It's a lot like `import` for Java. Use it for importing Python libraries and modules of your own.

Save the library script from the previous section in a file called `marriott.py`. Open the Python shell and execute

```
>>> import marriott
Total number of books in library...3
Searching for The Time Machine... available.
Searching for Through the Looking Glass... unavailable.
```

The interpreter automatically executes all top-level expressions when a module is imported, so the call to `main` was fired. To use a function, prepend it with the module name:

---

<sup>4</sup>The interested reader is referred to [Errors & Exceptions](#) for information about exceptions. We will not be using them much.

```
>>> newLib = {}
>>> marriott.addNewBook(newLib, "The Wind in the \
Willows")
>>> newLib
{'The Wind in the Willows': True}
```

Remember, everything in Python is an object. Use the `dir` command to print all of the methods and member variables of an object. Modules are objects too; `dir` will print out the top-level variables and functions in the module. For example,

```
>>> x = [1,2,3]
>>> dir(x)
['__add__', '__class__', ... (etc.) ...,
'append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
>>> dir(marriott)
['__builtins__', '__doc__', '__file__', '__name__',
'__package__', 'addNewBook', 'checkinBook',
'checkoutBook', 'findBook', 'main', 'numBooks']
```

By convention in Python, members with double underscores are not intended for public use. (Python does not have access control—public, private, protected, etc.)

To print out the docstring for a function in a module, use `help`:

```
>>> help(marriott.findBook)
Help on function findBook in module marriott:

findBook(library, bookTitle)
    Returns true if the library has bookTitle
    and it is checked in, false otherwise.
```

If you'd like certain parts of your module to not execute when it is imported, wrap the code inside of an if statement, like this:

```
if __name__ == "__main__":
    main()
```

using the library example again. The call to main will not fire when the library module is imported.

There are a vast number of modules that come with Python, and we will be exploring some of them throughout the semester. For more information about modules, see [Modules](#). For more information about modules that come with your Python distribution, see [The Python Standard Library](#).

## 0.9 Functional Programming

### 0.9.1 Functions are Values

Remember, everything in Python is an object—including functions. Functions can be assigned to variables, passed as arguments to a function, returned by a function, placed in lists, and so on. (What happens when you call dir on a function? Try it.) To motivate some key ideas, we'll build a running example.

Suppose we would like to filter out the odd numbers in a list. The following function does the job, using a list comprehension:

```
def removeOdds(x):
    return [n for n in x if n % 2 == 0]
```

Now, if we need to filter out the even numbers in a list, we have to re-write the entire function:

```
def removeEvens(x):
    return [n for n in x if n % 2 != 0]
```

What about all negative numbers? The structure of the function remains the same each time; it's the condition, or predicate, that is changing:

```
def remove... (x):
    return [n for n in x if ...]
```

Writing a function for each type of filter is error prone. Instead, we can write a function that captures the above structure and that uses a function, passed as an argument, to check the condition:

```
def filter(c, x):
    return [n for n in x if c(n)]
```

Python already has `filter` built in. Here is an example that replaces `removeOdds`:

```
def isEven(n):
    return n % 2 == 0

>>> filter(isEven, [1,2,3,4])
[2, 4]
```

We define `isEven` and pass it as an argument, along with a list, to `filter`.

### 0.9.2 Lambdas

Question: What if `isEven` is used only once? What if we called `filter` multiple times with different conditions? The program could get cluttered with simple ‘condition checkers’. Alternatively, we can use an anonymous function, or `lambda`. For example, this `lambda` adds two numbers  $x$  and  $y$  together:

```
lambda x, y: x + y
```

Notice that arguments are not enclosed in parentheses. Returning to our running example, we can extract all even numbers via:

```
>>> filter(lambda n: n % 2 == 0, [1,2,3,4])
[2, 4]
```

As lambdas are objects, they can be assigned to variables, added to lists, etc. Here are a couple examples:

```
>>> f = lambda x, y: x + y
>>> f(2, 3)
5
>>> sword = lambda health: 0 if health < 5 else \
health - 5
>>> axe    = lambda health: 0 if health < 10 else \
health - 10
>>> wand   = lambda health: 0 if health < 7 else \
health - 7
>>> weapons = {"sword": sword, "axe": axe, "wand": wand}
>>> weapons
{'wand': <function <lambda> at 0x7f3198502230>,
 'sword': <function <lambda> at 0x7f3198502140>,
 'axe': <function <lambda> at 0x7f31985021b8>}
>>> enemy_health = 100
>>> enemy_health = weapons["sword"](enemy_health)
>>> enemy_health
95
```

We create three lambdas with a single argument, `health`. Each lambda returns an updated value of the enemy's health, depending on how low it is. Notice that the shell only prints the lambda's memory address when it is evaluated (the same goes for functions). `weapons["sword"]` evaluates to the sword lambda, so we can call it with `enemy_health` as an argument.

(Note: Unlike Scheme, Javascript, and C#, lambdas in Python can only contain a single expression.)

### 0.9.3 Higher Order Functions

A function that takes a function as an argument, like `filter`, is termed a *higher order function*. In this section, we introduce two more built-in higher order functions—`map` and `reduce`.

`map` takes a function and a list and creates a new list by applying the function to the original list, element by element. Here is a simple example, comparing three different approaches to the same problem; the last approach uses `map`:

```
# Simple goal: Add 1 to each element of oldList
oldList = [1,2,3,4]

# Use a for loop
newList = []
for n in oldList:
    newList.append(n+1)

# Use a list comprehension
newList = [n+1 for n in oldList]

# Use map
newList = map(lambda n: n + 1, oldList)
```

All three are equivalent.

`reduce` also takes in a function and a list. The function must take two arguments. `reduce` calls the function with the first two elements of the list to produce an initial accumulated value. It then calls the function again with the accumulated value and the third element in the list to produce a new accumulated value, and so on. Figure 1 illustrates the behavior of `reduce` for

```
reduce(lambda acc, next: acc + next, [1,2,3,4])
```

Experiment with this in the shell.

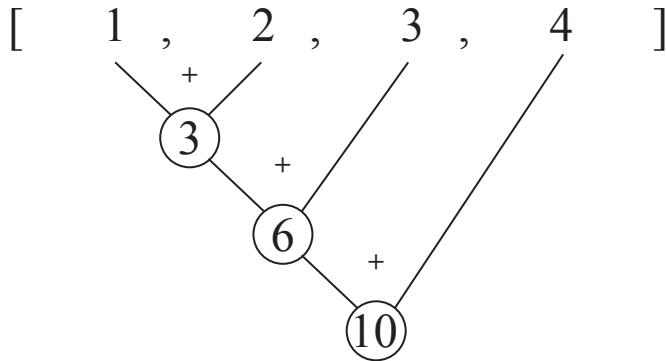


Figure 1: Reduce example. The first pair is added, and the remaining elements in the list are added to the accumulated result.

- What happens when the list is empty?
- When the list contains one element?

Here is a more complicated example that uses both map and reduce to calculate the number of vowels in the Gettysburg address:

```
Gettysburg = """Four score and seven years ago our  
fathers brought forth on this continent, a new nation,  
conceived in Liberty, and dedicated to the proposition  
that all men are created equal.
```

```
Now we are engaged in a great civil war, testing  
whether that nation, or any nation so conceived and  
so dedicated, can long endure. We are met on a great  
battle-field of that war. We have come to dedicate  
a portion of that field, as a final resting place  
for those who here gave their lives that that nation  
might live. It is altogether fitting and proper that  
we should do this.
```

```
But, in a larger sense, we can not dedicate --  
we can not consecrate -- we can not hallow -- this  
ground. The brave men, living and dead, who struggled  
here, have consecrated it, far above our poor power  
to add or detract. The world will little note, nor  
long remember what we say here, but it can never  
forget what they did here. It is for us the living,  
rather, to be dedicated here to the unfinished work  
which they who fought here have thus far so nobly  
advanced. It is rather for us to be here dedicated  
to the great task remaining before us -- that from  
these honored dead we take increased devotion to that  
cause for which they gave the last full measure of  
devotion -- that we here highly resolve that these  
dead shall not have died in vain -- that this nation,  
under God, shall have a new birth of freedom --  
and that government of the people, by the people,  
for the people, shall not perish from the earth."""
```

```
vowels = {'a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U'}
```

```
reduce(lambda x,y:x+y,  
      map(lambda x: 1 if x in vowels else 0,  
           Gettysburg))
```

The inner call to `map` translates vowels into 1 and all other characters into 0. The outer call to `reduce` adds up the 1's and 0's in the resulting list.

For more information about `filter`, `map`, and `reduce`, see [Functional Programming Tools](#).

**EXERCISE.** Write a function that uses `reduce` to compute the factorial of an integer  $n$ . Recall that the factorial of  $n$ , or  $n!$ , is

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdots 2 \cdot 1$$

So,  $3! = 3 \cdot 2 \cdot 1 = 6$ .  $0!$  is defined to be 1.

### Curried Functions and Partial Evaluation

Consider defining function `add2`, and invoking it on a number:

```
>>> add2 = lambda x: x + 2
>>> add2(3)
5
```

Suppose we don't wish to hard-wire 2 into the body of the `lambda`, but rather keep that option open. Here is a variant of the above:

```
>>> add_curried = lambda x: lambda y: x+y
```

Notice carefully how this `lambda` is phrased; it takes an `x` and *returns another lambda*, which is `lambda y: x+y`. So the *return type* of `add_curried` is another function. Given this, we can define `add2` as follows:

```
>>> add2 = add_curried(2)
```

This makes sure that `add2` is indeed bound to `lambda y: 2+y`. Now if we wanted to change the 2 to a 7, we can simply use

```
>>> add7 = add_curried(7)
```

Here are the takeaway points:

- A function such as `add_curried` is *curried* in that it takes a single argument (a value in this case) and returns a function.
- We can now apply this function to another value.

```
>>> add2(3)  
5
```

With a curried function, we can also feed all the values at-once:

```
>>> add_curried = lambda x: lambda y: x+y  
>>> add_curried(2)(3)  
5
```

Thus, in summary, a curried function allows more flexible usages:

- A function such as `add_curried` allows you to partially evaluate (by feeding a value, say 2) and later fully evaluate (by feeding another value, say 3); or
- It allows you to fully evaluate as in `add_curried(2)(3)` also.
- All curried functions are higher-order functions, in the sense that when we feed a value to them, it returns a function. (This function is typically later applied to another value.)

## 0.10 Recursion

This section is a brief introduction to recursion. The concepts in this section will be expanded on in later chapters.

Recursion works when a computation can be broken down into smaller computations of the same form, and these smaller computations can themselves be broken down in the same manner, until the smallest possible computations are reached. These are called the *base cases*. A computation is broken down into smaller computations by making one or more *recursive calls*.

Let's give a recursive implementation for a function that computes the factorial. We'll start out with the following template:

```
def factorial(n):
    ...
```

`factorial` expects `n` to be non-negative. What are the possible values of `n`? If `n` is 0, `factorial` should return 1. We can code this part:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        ...
```

If `n` is greater than 0, we can make a recursive call and use `factorial(n-1)` because

$$n! = n \cdot (n - 1) \cdots 2 \cdot 1 = n \cdot (n - 1)!$$

Our recursive implementation now becomes<sup>5</sup>

---

<sup>5</sup>Technical note: After entering the definition for `factorial`, try executing `factorial(1000)`, and note the stack overflow runtime error. Even after optimizing `factorial` so that it uses a tail call, it still produces a stack overflow. This is just like Java, C, and others.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        n * factorial(n - 1)
```

Reversing a list can also be done recursively: To reverse a list, we can first reverse its tail by making a recursive call and then append the first element of the list on the end. If the list is empty, there is nothing to do. Here is an implementation:

```
def rev(L):
    if L == []:
        return []
    else:
        reversedTail = rev(L[1:])
        reversedTail.append(L[0])
        return reversedTail
```

As a final example, let's write a function that concatenates two strings recursively. Recursive functions are easier to write after looking at a few cases:

<i>First String</i>	<i>Second String</i>	<i>Result</i>
"back"	+	"" → "back"
"back"	+	"s" → "backs"
"back"	+	"space" → "backspace"

If the second string is empty, there is nothing to do but return the first string. If the second string is non-empty, we can append its first character to the first string, and then recurse. Here is an implementation:

```
def concat(s1, s2):
    if s2 == "":
        return s1
    else:
        return concat( s1 + s2[0], s2[1:] )
```

In conclusion, we encourage you to use recursion often in this course: Many topics covered in this course and in other courses in theoretical computer science use recursion and induction, a related topic. A number of data structures and algorithms lend themselves well to recursion and may in fact be easier to implement recursively. In our experience, thinking about and designing programs using recursion leads to better design and fewer errors; but be aware of practical limits when using recursion (e.g., the stack in many programming language runtimes).



# Chapter 1

## Propositional/Boolean Logic

We shall use all these three terms interchangeably:

- Propositional logic
- Boolean logic
- Boolean algebra

In this chapter, we will focus on *Boolean* functions, and the notion of truth values (“True” and “False”, or 1 and 0) that underlies all of computing. Boolean functions are widely used. Even the simplest of programs employ them, as for example in:

```
if ((x == 0) and (y < 0)) or (z > w):
    ...do something...
else:
    ...do something else...
```

Here, and and or are Boolean functions, and the relations (<, > and ==) are built up using Boolean functions acting on bits in computer words.

It must be intuitively clear that the “else” part will be executed when the following condition is true:

```
((x != 0) or (y >= 0)) and (z <= w)
```

It is very important to be sure that such conclusions are correct, or else the program will not work as expected.

## 1.1 Historical Perspective

### 1.1.1 History of Boolean Functions in Computing

The Mormon pioneers were marching into the Salt Lake City valley around 1850. Exactly at the same time, across the Atlantic, Prof. George Boole (whose name “Boolean” algebra takes on) of the University of Cork, Ireland, published a book called *The Laws of Thought*.<sup>1</sup> Unfortunately, the world largely yawned, not knowing the significance of what was happening in Salt Lake City, or Cork, Ireland.

Picking up on Prof. Boole’s work, around 1930, a young MS student by the name of Claude Shannon at MIT, found Boole’s work to be foundational in designing relay-based computers. Those were the days when incorrectly designed relay based circuits were responsible for quite a few light-shows (shorting relays causing copious sparks). Again, despite the brilliance<sup>2</sup> of Shannon’s work, the world largely yawned, content with its *ad hoc* ways of designing relays.<sup>3</sup> Fortunately, the design of digital computers in the 1940s sparked (in a good way) interest in Boole’s calculus, and the rest is history.

### 1.1.2 History of Digital Design and Circuit Simulation

It is clear that relay-based circuits led to vacuum tube based digital circuits, which then led to transistors and subsequently integrated circuits. Nowadays, some Graphical Processing Unit (GPU) chips contain over *seven billion* transistors—one per human on earth.<sup>4</sup> It is these transistors that are powering the modern information-based world. They are also behind the world’s supercomputers that help us understand nature—by simulating from first principles how things work (including cells, the human brain, and supernovae). All these digital devices now *define* how we live as a species.

---

<sup>1</sup>Some people have observed that these *laws of thought* have since then helped prevent the *loss of thought*. The exact title of Boole’s book is “An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities.”

<sup>2</sup>pun intended...

<sup>3</sup>...and staying sufficiently far from the relays.

<sup>4</sup>To gain better appreciation for the scale of this number, a human lives around 3 billion seconds.

### 1.1.3 History of Propositional Reasoning

While Professor Boole's work was in the 1850s, the logicians were busy at work beginning with Socrates (around 400 BC) and his pupil<sup>5</sup> Euclid. Their work consisted of "syllogisms" (conclusions) inferred from logical premises.

This line of work matured in the earliest 20th century giving rise to various mathematical logics and associated proof techniques. These proof techniques have been extensively studied from a computational point of view. They in fact help ensure that hardware and software systems are defect-free. You may not realize this: but one of the reasons microprocessors have become so affordable (and hence are found in every device ranging from toys to tanks) is because we can now make them (relatively) bug-free.<sup>6</sup> This is achieved through *automate reasoning* in the following ways:

- The main logic blocks used in microprocessors is formally verified at the gate netlist level using *functional equivalence methods*.<sup>7</sup>
- Virtually all of the arithmetic (floating-point) units are subject to more rigorous theorem-proving based methods.
- Cache coherence protocols are verified using finite-state model-checking methods.

All these developments were set in motion on an aggressive scale in the mid 1990s following a widely publicized microprocessor bug. This was when Intel's Pentium microprocessor had a fatal arithmetic flaw. For instance, from Nicely's article at <http://www.trnicely.net/pentbug/pentbug.html>:

$$\begin{aligned} 4195835.0 - 3145727.0 * (4195835.0 / 3145727.0) &= 0 \quad (\text{Correct value}) \\ 4195835.0 - 3145727.0 * (4195835.0 / 3145727.0) &= 256 \quad (\text{Flawed Pentium}) \end{aligned}$$

This, and scores of similar errors caused a massive recall of Intel Pentiums, costing Intel half a billion dollars in revenues to repair. It is very easy to see why "crap shoot" testing is not scalable. Even to test a 64-bit adder, one must go through  $2^{128}$  Boolean combinations of inputs—something that will take  $10^{22}$  years to test, with even one test run every nano second!<sup>8</sup> Using

<sup>5</sup>Did the term pupil come from the fact that it is they who let the teacher clearly see?

<sup>6</sup>Microprocessors do have published errata ranging over several pages; but we have learned how to avoid fatal mistakes.

<sup>7</sup>A company in Austin called Centaur Inc. builds x86 processors where the register-transfer logic of the entire microprocessor is represented in a language called ACL2, and daily verification regressions are run on the whole design.

<sup>8</sup>This calculation was performed using Wolfram Alpha available at <http://www.wolframalpha.com>.

mechanized proof methods, we can essentially consider all relevant states and inputs at once—not case by case. For example, modern microprocessor verification tools avoid this case analysis and *directly* execute this calculation using symbolic inputs. In the paper “Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation,”[8], the authors describe how this verification technology has scaled up and become part of standard industrial practice.

## 1.2 Truth Values and Functions

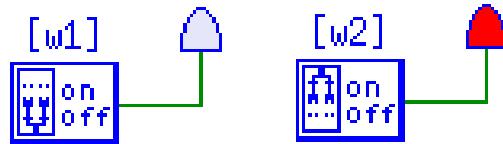


Figure 1.1: A switch and LED represented in TkGate

We start with Boolean or truth values. They are generally represented by 0 (“off”, or False as in Python) and 1 (“on”, or True as in Python); see Figure 1.1 for a more vivid representation using TkGate. We will employ TkGate 2.0-b10 (beta) and mainly focus on the creation of simple projects. For more details, consult the excellent online help as well as accompanying tutorial material coming with TkGate.<sup>9</sup>

### 1.2.1 Boolean Functions

There is a set of fundamental Boolean functions that are well-known and which get used frequently. In this section we will introduce these functions and their truth tables. Familiarity with these functions and understanding why the truth tables are as they are will help tremendously in developing strong intuitions in Boolean logic and Boolean algebra. The functions we

---

<sup>9</sup>Note: Since TkGate is a free tool, it occasionally segfaults, but saves your work in a file called PANIC.v. If you hit “save” frequently, the contents of PANIC.v is likely to be fairly current.

will cover in this section are not, and, or, if-then, if-and-only-if, xor, nor, nand.

### not

not is the only *unary* operator we will study in this section. This simply means that it operates on one Boolean statement instead of two. The definition of not is straight-forward and as one would expect. Applying not to any operand will invert its truth value. not may be represented with any of the following symbols: ( $!$ ,  $\sim$ ,  $\neg$ ). The truth table for not is:

$P$	$!P$
0	1
1	0

### and

and statements are true only when both operands are true. If either of the operands is false, then the whole statement is false. Like not the formal meaning for and is intuitive. and may be represented with either of ( $\cdot$ ,  $\wedge$ ). The truth table for and is:

$P$	$Q$	$P \cdot Q$
0	0	0
0	1	0
1	0	0
1	1	1

### or

or statements are true when *at least* one of the operands is true. An or statement is only false when both of its operands are false. Note that this definition of or is different from the notion of *or* wherein only one of the two options can be true. For example, if somebody tells you that you can have soup *or* salad, typically they mean that you may have one or the other but not both. This second meaning of *or* will be defined later in this section. or

may be represented with either of  $(+, \vee)$ . The truth table for or is:

$P$	$Q$	$P + Q$
0	0	0
0	1	1
1	0	1
1	1	1

### if-then

if-then statements are true when the first operand is false *or* the second operand is true. An if-then statement is only false when the first operand is true *and* the second operand is false. if-then statements may also be referred to as implications. if  $P$  then  $Q$  is equivalent to  $P$  implies  $Q$ .

An if-then statement is made up of two parts, the *antecedent* and the *consequent*. The antecedent is the first statement of the implication, the piece that does the implying. The consequent is the second statement and is what is implied by the antecedent. In the statement if  $P$  then  $Q$ ,  $P$  is the antecedent and  $Q$  is the consequent.

There is some subtlety to the definition of if-then that should be addressed. It can be puzzling to try and work out why an implication is always true when the antecedent is false. We will attempt to make this clear via a simple example. Take the statement, “If it is sunny then I will ride my bicycle to class.” Clearly, if it is sunny and I ride my bicycle to class then the statement is true. Conversely, if it is sunny and I *don’t* ride my bicycle then the statement is false. Consider the case when it is not sunny and I ride to class anyhow. I have not violated any terms of the original statement, therefore it is still true. Likewise if it is not sunny and I do not ride to class. I made no promise under such circumstances and so my original claim remains true. This is how we arrive at the truth values for implication.

if-then may be represented with either of  $(\Rightarrow, \rightarrow)$ . The truth table for if-then is:

$P$	$Q$	$P \Rightarrow Q$
0	0	1
0	1	1
1	0	0
1	1	1

**if-and-only-if**

if-and-only-if statements are true when the first operand has the same truth value as the second operand. if-and-only-if is frequently abbreviated iff. It may also be referred to as a bi-implication. This alternate name is telling and hints at the true nature of iff statements. Namely, the statement  $P$  iff  $Q$  is true exactly when  $(P \Rightarrow Q) \cdot (Q \Rightarrow P)$ , i.e. when  $P$  implies  $Q$  and  $Q$  implies  $P$ . It is left to the reader to confirm this fact. if-and-only-if may be represented with either of  $(\Leftrightarrow, \leftrightarrow)$ . The truth table for if-and-only-if is:

$P$	$Q$	$P \Leftrightarrow Q$
0	0	1
0	1	0
1	0	0
1	1	1

**xor**

xor (*exclusive or*) statements are true when *exactly* one of the operands is true. Recall the soup or salad example given above. If you are asked whether you want soup or salad, the usual implication is that you may have one or the other but not both. The definition of xor is similar: the statement is true if one of the operands or the other is true but not both. xor is represented with  $\oplus$ . The truth table for xor is:

$P$	$Q$	$P \oplus Q$
0	0	0
0	1	1
1	0	1
1	1	0

**nor**

nor statements are true only when both the left and right operands are false. nor is true exactly when or is false, and vice versa. Symbolically,  $P$  nor  $Q$  is the same as  $!(P + Q)$ . nor is usually just represented as nor. The truth table for nor is:

$P$	$Q$	$P$ nor $Q$
0	0	1
0	1	0
1	0	0
1	1	0

**nand**

nand statements are true when the left operand and the right operand are not both true. Similarly to nor, nand is true exactly when and is false. Symbolically,  $P$  nand  $Q$  is equivalent to  $!(P \cdot Q)$ . nand is typically represented simply as nand. The truth table for nand is:

$P$	$Q$	$P$ nand $Q$
0	0	1
0	1	1
1	0	1
1	1	0

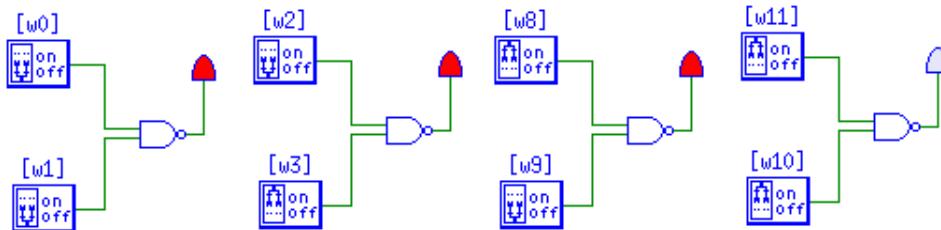


Figure 1.2: The Nand function illustrated over 00, 01, 10, and 11

x	y	z	nand(x,y)
0	0	1	>> nand(0,0) 1
0	1	1	>> nand(1,1) 0
1	0	1	>> nand(0,1) 1
1	1	0	>> nand(1,0) 1

```

def nand(x,y):
    return { (0,0):1,
             (0,1):1,
             (1,0):1,
             (1,1):0 }[(x,y)]

```

Figure 1.3: The Nand Truth-table, Lambda Representation, and Hash-table Representation

### 1.2.2 Case Study: the Nand Function Represented

Now that we have our first “digital circuit” (a simple on/off circuit), it is time to turn our attention to the notion of a *function* and more specifically, a *Boolean function*. A function is an entity (a “black box”) that maps its input to its output. Given a function, a specific output is defined for each input. In Figures 1.2, the icon there represents the Nand function, and its output on every possible input is defined. Called the *Nand gate*, this function behaves as follows:

- If both inputs are a 1 (switch “on”), the output is a 0 (“LED off”).
- Otherwise (if any input is a 1), the output is a 0.

The term “Nand” comes from “Not” plus “And”, as will be made clear soon.

There is never a situation in which a function (*any function*) produces more than one output for the same input. In other words, a function produces a *unique* output for every input. For example, given input 1,1, the Nand gate always produces 0. Of course, it is possible for a function to pro-

duce the same output on several inputs. For instance, for inputs  $0,0$ , “ $0,1$ ”, and “ $1,0$ ”, Nand produces a 1 output. Such functions are called *many-to-one* functions. Therefore, Nand is a many-to-one function.

We will use the notation of a *truth table* to denote functions. We can equivalently capture the behavior using lambdas or hash tables as shown in Figure 1.3. Hash-tables in Python are data structures that store “key:value” pairs. In the example here, the hash-table contains keys matching the truth-table inputs (notice that the keys are pairs) and the value part is the truth-table output. Given  $x$  and  $y$ , the hash-table based nand simply indexes into the hash-table as shown.

## Functions and Signature

A function is a mathematical object that expresses how items called “inputs” can be turned into other items called “outputs.” To adequately describe functions, we need to introduce the notion of *signature*.

A function maps its *domain* to its *range*; and hence, the inputs of a function belong to its *domain* and the outputs belong to its *range*. **The domain and range of a function are always assumed to be non-empty sets.** The expression “ $f : T_D \rightarrow T_R$ ” is called the signature of  $f$ , denoting that  $f$  maps the *domain* set  $T_D$  to the *range* set  $T_R$ . In case of the Nand gate, its signature is

$$\text{Nand} : \text{Bool} \times \text{Bool} \rightarrow \text{Bool}$$

if we assume that  $\text{Bool} = \{0, 1\}$ . (Sometimes we employ True/False in lieu of 1/0.) The signature of *Nand* is evident from Figure 1.3 where it is clear that *Nand* takes two Boolean inputs and yields a Boolean output. Here is how you can read the signature of *Nand*:

- “Gimme a pair  $(x, y)$ ” where they are both Booleans
- “I’ll return a result that is a Boolean”

The signature does not say *how* the result is computed. It only says *what type* the result has (here, “type” means the same as “set”).

Writing signatures down for functions makes it very clear as to what the function “inputs” and what it “outputs.” Hence, this is a highly recommended practice. As a simple example,  $+ : \text{Nat} \times \text{Nat} \rightarrow \text{Nat}$  denotes the signature of natural number ( $\{0, 1, \dots\}$ ) addition.

### 1.2.3 How Many Functions over One Input Exist?

Before we study two-input functions thoroughly, let us study all possible one-input functions, *i.e.*, those with signature

$$\text{Bool} \rightarrow \text{Bool}$$

Let us try and crank out *all* of those functions that take two inputs. Figure 1.4 summarizes all possible functions of this signature. There are 4 of them! Why 4? Notice that each function is *totally determined* by what bits (0 or 1) that we put into the output ( $z$ ) column. Since there are two spots there, there are  $2^2$  or 4 such ways to fill the output column—resulting in 4 functions. The only(?) interesting function in this list is the *Inverter* or the *Not* gate that inverts the bit coming into the input.

### 1.2.4 How Many Functions over Two Inputs Exist?

It is clear that *Nand* is only *one of the possible functions* of signature  $\text{Bool} \times \text{Bool} \rightarrow \text{Bool}$ . Let us try and crank out *all* of those functions that take two inputs. Figure 1.5 summarizes all possible functions of this signature. There are 16 of them! Why 16? Notice that each function is *totally determined* by what bits (0 or 1) that we put into the output ( $z$ ) column. Since there are four spots there, there are  $2^4$  or 16 such ways to fill the output column—resulting in 16 functions.

### 1.2.5 Counting All Possible $N$ -input Functions

Let us now generalize a bit, and answer how many distinct  $N$ -input truth-tables there are. Well, there are  $2^N$  output positions, and each position can be filled in two ways. Thus, **there are  $2^{2^N}$   $N$ -input Truth tables!** Students often say  $2^N$ —*wrong!*.

#### Product Catalog of the Purveyor of *all* $N$ -input Gate Types!

Let's drive this point home: how many entries will there be in the product catalog of a (mindless) purveyor of *all possible gate types* of  $N$  inputs for various  $N$ ? Thanks to Wolfram Alpha (which I profusely thank for most of these calculations—see [www.wolframalpha.com](http://www.wolframalpha.com)):

- 16 – 2-input gate types (of the kind shown in Figure 1.5) are possible.

<i>Constant</i>	$\overline{\overline{x \ z}}$	<i>Constant</i>	$\overline{\overline{x \ z}}$	$z = !x$	$\overline{\overline{x \ z}}$	$z = x$	$\overline{\overline{x \ z}}$
0	0 0	1	0 1	1 0	0 1	0 0	1 1
	1 0		1 1		1 0		

Figure 1.4: All possible 1-input Boolean Functions

<i>Constant</i>	$\overline{\overline{x \ y \ z}}$	$z = x \cdot y$ AND	$z = x \cdot !y$	$z = x$
0	0 0 0	0 0 0	0 0 0	0 0 0
	0 1 0	0 1 0	0 1 0	0 1 0
	1 0 0	1 0 0	1 0 1	1 0 1
	1 1 0	1 1 1	1 1 0	1 1 1

$z = !x \cdot y$	$z = y$	$z = !x \cdot y + x \cdot !y$ XOR	$z = x + y$ OR
0 0 0	0 0 0	0 0 0	0 0 0
0 1 1	0 1 1	0 1 1	0 1 1
1 0 0	1 0 0	1 0 1	1 0 1
1 1 0	1 1 1	1 1 0	1 1 1

$z = !(x+y)$ NOR	$z = xy + !x \cdot !y$ XNOR or =	$z = !y$	$z = x + !y$
0 0 1	0 0 1	0 0 1	0 0 1
0 1 0	0 1 0	0 1 0	0 1 0
1 0 0	1 0 0	1 0 1	1 0 1
1 1 0	1 1 1	1 1 0	1 1 1

$z = !x$	$z = !x + y$ IMPLICATION	$z = !(x \cdot y)$ NAND	<i>Constant</i>
0 0 1	0 0 1	0 0 1	0 0 1
0 1 1	0 1 1	0 1 1	0 1 1
1 0 0	1 0 0	1 0 1	1 0 1
1 1 0	1 1 1	1 1 0	1 1 1

Figure 1.5: All possible 2-input Boolean Functions

- 256 – 3-input gate types possible
- 65,536 – 4-input gate types possible
- 4,294,967,296 or over 4 billion – 5-input gate types possible (and a human lives less than this many seconds; so by the time you've read thru them, you are dead!)
- $1.8 \cdot 10^{19}$  6-input gate types
- $3 \cdot 10^{38}$  7-input gate types
- $10^{77}$  8-input gate types
- $10^{154}$  9-input gate types (now you are approaching the number of fundamental particles in the universe...)
- $10^{308}$  10-input gate types (now have surely exceeded the number of fundamental particles in the universe... go count, if you doubt me!)

### Universal Gates

Clearly the (mindless) purveyor of *all possible gate types* of  $N$  inputs will soon be put out of business, because they cannot finish printing their catalog for even  $N = 5$ . Fortunately, they don't have to do this! There are certain gates that are *universal* in that they can be used to build *any* Boolean function of *any desired number of inputs*. The *Nand* function is one such: by building a two-input *Nand* gate (and enough copies thereof), one can build a Boolean function of any input size (or *input arity*).

The term *arity* of a gate (or a function) stands for the number of inputs of that gate (or function).

In the next chapter, we shall discuss the notion of universal gates in greater depth.



# Chapter 2

## Gate Types, Boolean Identities

### 2.1 Syntax for Boolean Formulae

There are many different ways of writing down Boolean expressions. Depending on the context, we may use one of these variations. Figure 2.1 summarizes our common usages, and also shows commonly used gate icons.

### 2.2 Boolean Identities

Boolean identities help us simplify Boolean expressions (as well as circuits built out of gates). We list a collection of identities that prove useful in practice (some of these are adapted from [1]). We express these identities as equalities “=”:

- $(x \Rightarrow y) = !x + y$ : This is obvious if one applies the rules of  $!$  and  $+$  for all possible  $x$  and  $y$ .
- $(x \oplus y) = x.!y + !x.y$ : This explains why  $\oplus$  is the Boolean  $\neq$  operator.
- $(x \overline{\oplus} y) = x.y + !x.!y$ : This explains why  $\overline{\oplus}$  is like the Boolean equality operator.
- $x + (y + z) = (x + y) + z$ , or associativity of  $+$
- $x.(y.z) = (x.y).z$ , or associativity of  $.$
- $x + y = y + x$ , or commutativity of  $+$
- $x.y = y.x$ , or commutativity of  $.$
- $x.(y + z) = (x.y) + (x.z)$ , or distributivity of  $.$  over  $+$
- $x + (y.z) = (x + y).(x + z)$ , or distributivity of  $+$  over  $.$
- $x + 0 = x$ , identity for  $+$

Quantity	Name	Variant	Variant	Variant	Examples
Value	“Zero”	0	False	“Off”	0 or False
Value	“One”	1	True	“On”	1 or True
Function	“And”	$\wedge$	.	Conjunction	$x \wedge y, x.y$
Function	“Or”	$\vee$	$+$	Disjunction	$x \vee y, x + y$
Function	“Not”	$\neg$	!	Negation	$\neg x, \neg y$
Function	“Implication”	$\Rightarrow$	If-Then		$x \Rightarrow y, \text{if } x \text{ then } y$
Function	“XOR”	$\oplus$	$\neq$	“Inequality”	$x \oplus y, x \neq y$
Function	“XNOR”	$\overline{\oplus}$	$=$	“Equality”	$x \overline{\oplus} y, x = y, x \Leftrightarrow y$

Function	Gate Icon	inputs
“And”		on the left
“Nand”		on the left
“Or”		on the left
“Not”		on the left
“Implication”		i on left s is beneath
“XOR”		on the left
“XNOR”		on the left

Figure 2.1: Different Syntaxes as well as Gate Icons for Boolean Functions

- $x \cdot 1 = x$ , identity for  $\cdot$
- $x + x = x$ , idempotence of  $+$
- $x \cdot x = x$ , idempotence of  $\cdot$
- $x \cdot (x + y) = x$ , absorption 1
- $x + (x \cdot y) = x$ , absorption 2
- $x + 1 = 1$ , annihilator for  $+$
- $x \cdot 0 = 0$ , annihilator for  $\cdot$
- $x \cdot !x = 0$ , complementation 1
- $x + !x = 1$ , complementation 2
- $!!x = x$ , double negation
- $(!x + !y) = !(x \cdot y)$ , **De Morgan 1**
- $(!x \cdot !y) = !(x + y)$ , **De Morgan 2**

**The last two identities above are called De Morgan's Laws.**

Now, let us derive some rules specific to *Nand*, *Nor*, *XOR*, and  $\Rightarrow$ :

- $nand(0, x) = nand(y, 0) = 1$ , for any  $x$  and  $y$ : For a Nand, a “0 forces a 1.”
- $nor(1, x) = nor(y, 1) = 0$ , for any  $x$  and  $y$ : For a Nor, a “1 forces a 0.”
- $1 \oplus x = x \oplus 1 = !x$
- $0 \oplus x = x \oplus 0 = x$
- $x \oplus x = 0$
- $x \oplus !x = 1$
- $0 \Rightarrow x = 1$
- $1 \Rightarrow x = x$

We will have occasions to use these propositional identities in later sections.

## 2.3 Uses of Universal Gate Sets

In this section, we wish to demonstrate that the NAND gate is universal. This is achieved by showing that an Or gate can be realized using Nand §2.3.1, an Implication gate can be realized using Nand §2.3.2, and a 2-to-1 mux can also be realized using Nand §2.3.3. While these demos are, by themselves, not a formal proof that *Nand* is universal, we will be driving towards such a proof in this discussion.

### 2.3.1 Realizing Inclusive and Exclusive Or using Nands

An *inclusive* Or (or simply “or”) has the truth table discussed in Figure 1.5. The Or function can be realized using the Nand function as follows:

$$Or(a, b) = Nand(\bar{a}, \bar{b})$$

Here, the inversions realizing  $\bar{a}$  and  $\bar{b}$  can also be realized using a Nand. In particular,

$$\bar{x} = Nand(x, x).$$

We leave it to the reader to draw and simulate this circuit using TkGate. Figure 2.2 shows how to realize the Exclusive OR functionality using Nands.

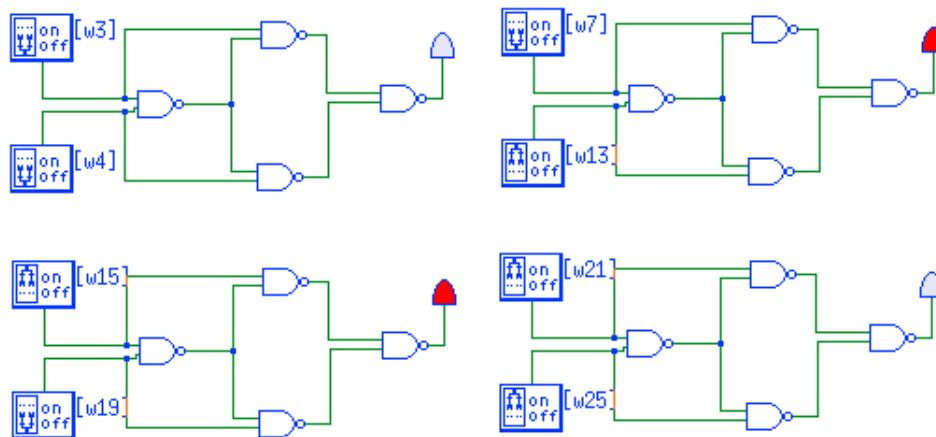


Figure 2.2: XOR via Nand

### 2.3.2 An “Implication Gate”

The Implication function has truth-table described in Figure 1.5. It can be realized using Nands as follows:

$$Imp(a, b) = Nand(a, \bar{b})$$

We leave it to the reader to draw and simulate this circuit using TkGate.

### 2.3.3 A 2-to-1 Mux Using Nand Gates

A 2-to-1 multiplexor is shown in Figure 2.4. This multiplexor can be realized as follows:

$$mux21(i1, i0)(s) = \text{Nand}(\text{Nand}(\bar{s}, i0), \text{Nand}(s, i1))$$

If you expand this out, it becomes the following equation:

$$mux21(i1, i0)(s) = \bar{s}.i0 + s.i1$$

That is, “IF  $s$  THEN  $b$  ELSE  $a$ .” This is why in §2.4.1, we call the gate an if-then-else gate.

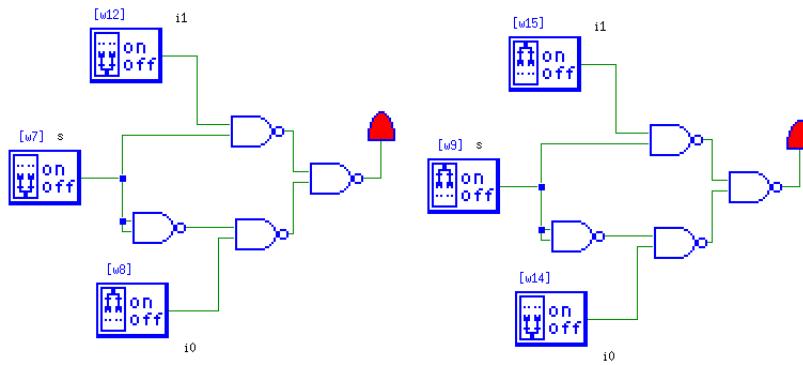


Figure 2.3: A 2-to-1 multiplexor Realized Using Nands

## 2.4 Realizing all 2-input Functions via Muxes

In the remainder of this chapter, we discuss the use of a device called a *multiplexor* to build all two-input gate types. In particular, we will be using a **4-to-1** multiplexor. The main idea is to observe that in Figure 1.5, the “personality” of any gate is decided by the output column of four bits. Thus, if we can build a single device where these four bits can be fed as input, we would have the means to realize any 2-input gate at will. This is the goal of the rest of this chapter. We begin by studying a 2-to-1 multiplexor.

The reader will notice that the arguments and constructions we employ here will help them realize *any*  $N$ -ary Boolean function using 2-to-1 multiplexors alone, arranged in the shape of a *decoding tree*. This will also provide a proof of universality of many gate types:

Since *Nand* and *Nor* (and many other gate types) can help build a 2-to-1 multiplexor, and since a 2-to-1 multiplexor can be used to build any  $N$ -ary Boolean function, it will mean that *Nand* and *Nor* are universal.

### 2.4.1 A 2-to-1 Multiplexor or an if-then-else gate

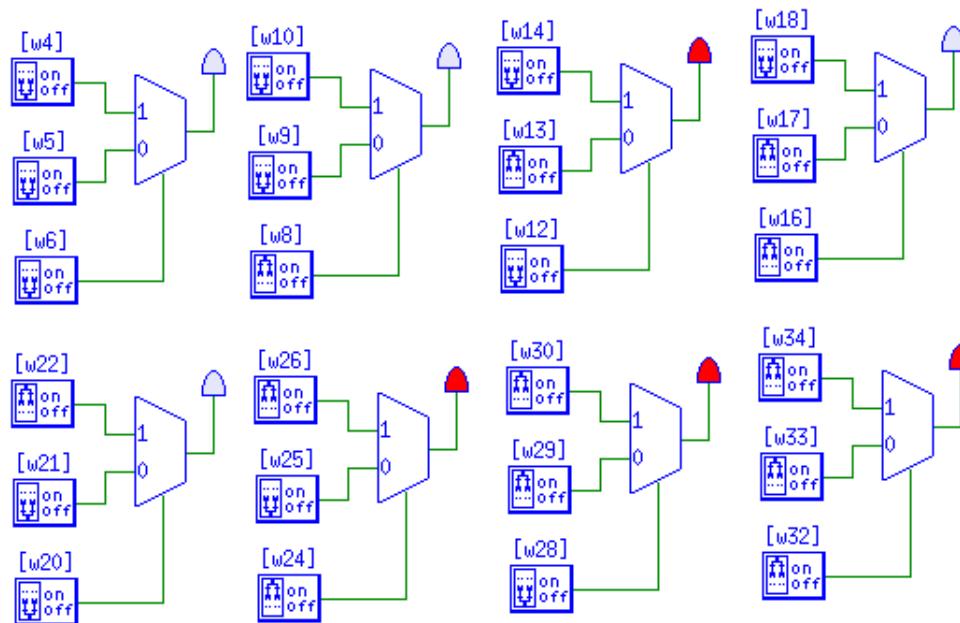


Figure 2.4: A 2-to-1 multiplexor

A 2-to-1 mux is illustrated in Figure 2.4. If we take the inputs of this circuit to be  $i_1$  and  $i_0$  (for “inputs”) and  $s$  (for “select”), the functionality of this circuit is described by the following lambda expression:

```
>>> mux21 = lambda(i1,i0): lambda(s): i0 if not(s) else i1

>>> mux21((0,1))
<function <lambda> at 0x100499f50>

>>> mux21((0,1))(0)
1
```

```
>>> mux21((0,1))(1)
0
```

It is evident that the behavior of Figure 2.4 is being captured by the above lambda expression. It is basically an if-then-else (or if-else) gate: it copies  $i_0$  to the output if  $\text{not}(s)$ ; else, it copies  $i_1$  to the output. Look at Figure 2.4 again: the  $i_0$  input is labeled 0 and the  $i_1$  input is labeled 1. When the input wire on the slant edge of the trapezium (select input or  $s$ ) is 0 (low), the value on the  $i_0$  input is copied onto the output, as is clear in the four cases when the LED is lit. For example, in the top row, when the select input is low, the input  $i_0$  is high, and hence the LED on the output port is lit.

It is also clear that a  $\text{mux21}$  is one of the  $2^{2^3}$  possible three-input Boolean functions. This is because  $\text{mux21}$  is sensitive to all of its three inputs, and maps the inputs as described by Figure 2.4. Let us now see how an if-then gate can be obtained from  $\text{mux21}$ .

### An if-then Gate

The Implication gate in Figure 1.5 can be called an if-then gate. One can use the following lambda to describe it, and realize it as shown in Figure 2.5. The symbol  $\Rightarrow$  will be used to denote implication. Thus, one can express the functionality of this gate also as  $s \Rightarrow i_1$ . Notice that if  $s$  is 0, the output is 1. This gate eminently captures our intuitive English notion of “If-Then”. To understand this, consider these sentences:

- S0: “IF the sun does not rise in the east THEN a year is not roughly 365 days”
- S1: “IF the sun does not rise in the east THEN a year is roughly 365 days”
- S2: “IF the sun rises in the east THEN a year is not roughly 365 days”
- S3: “IF the sun rises in the east THEN a year is roughly 365 days”

Consider these four sentences as well (that are similarly structured):

- T0:  $!(1 = 1) \Rightarrow !(2 = 2)$
- T1:  $!(1 = 1) \Rightarrow (2 = 2)$
- T2:  $(1 = 1) \Rightarrow !(2 = 2)$
- T3:  $(1 = 1) \Rightarrow (2 = 2)$

It is clear that S0, S1, and S3 (likewise T0, T1, and T3) must be true, and both S2 and T2 are false. This is because

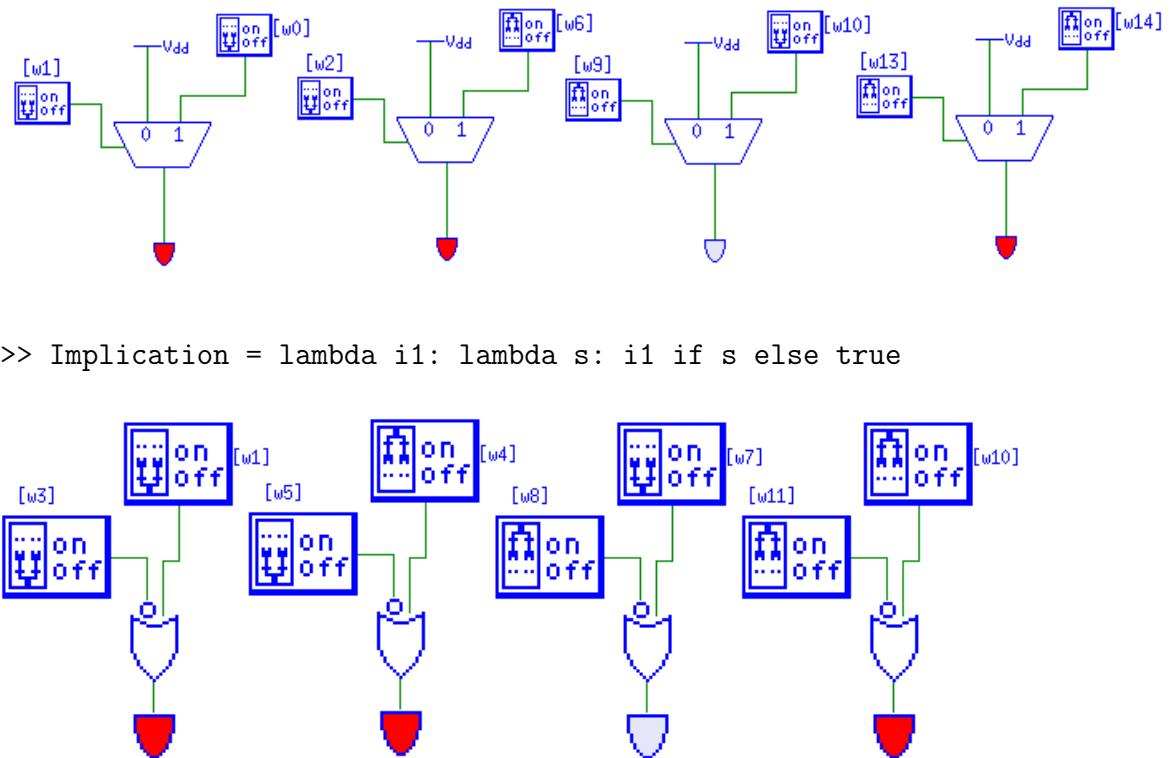


Figure 2.5: An Implication Gate:  $s \Rightarrow i_1$ , shown using two different icons. The “select” input  $s$  is to the left-hand side, while the input  $i_1$  is on top.

- If the “if condition” is false, the whole sentence can be regarded as one true fact. The conclusion in the “then” part cannot be drawn. So it is as if the if-then (or implication based assertion) condition did not even exist.
- If the “if condition” is true, then we *are* going to draw the conclusion from the “then” part. So in this case, this conclusion better be true. Else we would sow falsehood into our reasoning machinery. It would be catastrophic to conclude “false” as a theorem because it would then allow us to prove *anything* at all. This is because  $\text{False} \Rightarrow P$  is *True*, no matter what  $P$  is. Now, with  $\text{False}$  as a theorem, and with  $\text{False} \Rightarrow P$ , we can infer  $P$  for any  $P$ .

More specifically, let us take T2 for discussion. We know that  $(1 = 1)$  is true; therefore, we can pursue T2 and conclude  $!(2 = 2)$ . This is like proving

*false*. Once *false* is proved as a theorem, *all problems* — even open conjectures — can be trivially proved. Not only that, for every logical assertion  $P$ , we can conclude  $P$  as well as  $\neg P$ —something no logical system wants to admit.

### 2.4.2 Creation of a general 2-input function module

We are now ready to construct a multiplexor based realization of all possible 2-input gates. We will build the desired **4-to-1** multiplexor using three **2-to-1** multiplexors. The result is in Figure 2.6.

#### Testing the Lambda Expressions of the Muxes

The encoding of the functions defined in Figure 2.6 as Lambda expressions is also given in the same figure. Notice how the nested Lambda expressions in this figure can be tested step by step. For instance, `mux41((0,1,1,0))` returns a function object `<function <lambda> at 0x100499ed8>` while `mux41((0,1,1,0))((0,0))` tests the returned function for proper behavior. Notice how we “hooked up” the Lambda functions in the same manner as the circuits are wired, and even this “alternative mux” (called `mux41alt`) works the same.

#### Creating module gen2input

In the manner explained above, create a general 2-input function block with the schematic shown in Figure 2.7, arriving its module definition in a new module called `gen2input`.

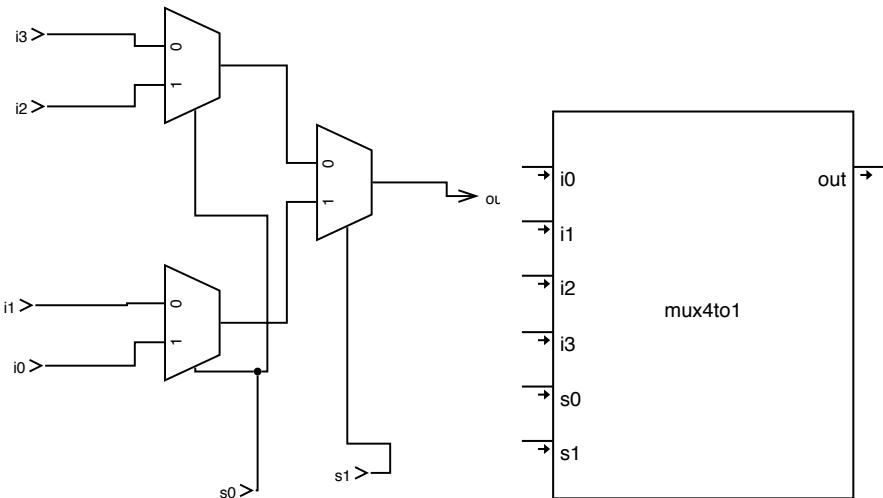
#### Editing gen2input to become an XOR

Notice that we left the “switches” inside Figure 2.7(left) at “all off”.<sup>1</sup> But the image on the interactive screen is always correct.

To arrive at an XOR, all we need to do is to create the truth-table column for XOR, which goes as 0, 1, 1, 0. We accomplish this by making a copy of the module `gen2input`, calling it `xorViaMux41`, and editing the switches inside this module. Figure 2.8 shows all the pertinent details. We can also simulate these XOR functions similar to how we tested the muxes in Figure 2.6. Here are some details:

---

<sup>1</sup>Note: Due to a bug in tkgate’s generate schematic function, the switch states are not accurately reflected in the generated printout. It appears always “on”.



```

>>> mux41 = lambda(i3,i2,i1,i0): lambda(s1,s0):
    i0 if not(s1) and not(s0) else i1 if not(s1) and s0
        else i2 if s1 and not(s0) else i3

>>> mux41((0,1,1,0))
<function <lambda> at 0x100499ed8>

>>> mux41((0,1,1,0))((0,0))
0

>>> mux41((0,1,1,0))((0,1))
1

>>> mux21 = lambda(i1,i0): lambda(s): i0 if not(s) else i1

>>> mux21((0,1))
<function <lambda> at 0x100499f50>

>>> mux21((0,1))(0)
1

>>> mux21((0,1))(1)
0

>>> mux41alt = lambda(i3,i2,i1,i0): lambda(s1,s0):
    mux21((mux21((i3,i2))(s0), mux21((i1,i0))(s0)))(s1)

>>> mux41alt((0,1,1,0))((0,0))
0

>>> mux41alt((0,1,1,0))((0,1))
1

```

Figure 2.6: A 4-to-1 mux: Before and After Interface Generation. You can verify that depending on the input bit combination at inputs  $s_0$  and  $s_1$ , one of the four inputs ( $i_0$  through  $i_3$ ) is copied to the output (out). We also provide ways to define these functions using Lambdas

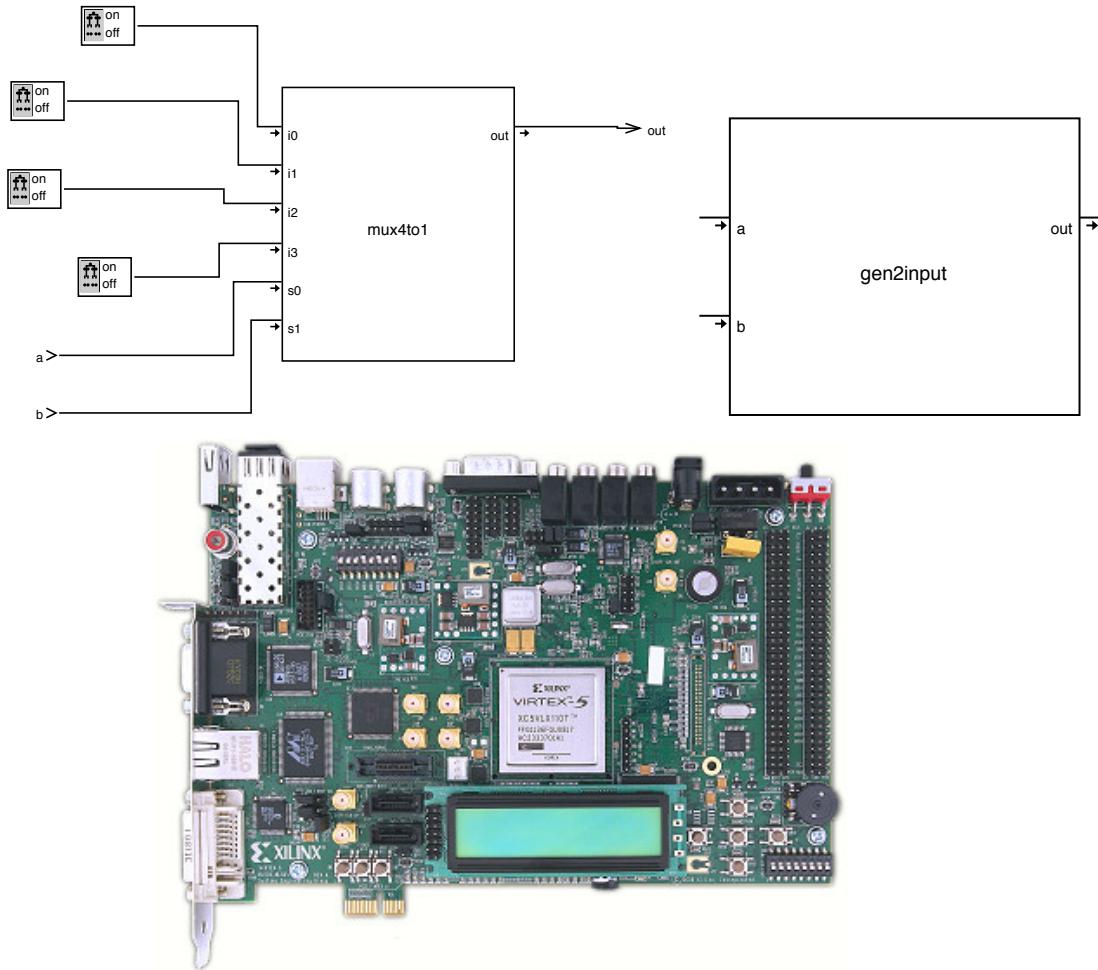


Figure 2.7: A General 2-input Function Module and its Symbol (Interface). This is a “general” builder because by setting the switches, we can realize *any* of the 16 functions in Figure 1.5 by downloading a bit-pattern into their storage units that then set the “input switches.” A prototyping board with Virtex-5 Field Programmable Gate Arrays (FPGAs) consisting of over 300K such *configurable logic blocks* is shown (Image courtesy of Xilinx/Digilent Inc.). In a research project at Utah called XUM (<http://www.cs.utah.edu/fv/XUM>), we have packed in eight MIPS cores plus interconnect into such a board.

```

>>> xor = mux41((0,1,1,0))

>>> xor((0,0))
0

>>> xor((0,1))
1

>>> xor((1,0))
1

>>> xor((1,1))
0

>>> xoralt = mux41alt((0,1,1,0))

>>> xoralt((0,0))
0

>>> xoralt((0,1))
1

>>> xoralt((1,0))
1

>>> xoralt((1,1))
0

```

### 2.4.3 Counting Boolean Functions: the Mux Approach

Figure 2.8 immediately gives us another way to arrive at the number of Boolean functions of a certain number of inputs. Look at this circuit: it is the switch settings that makes this into an XOR gate! How many settings are there for these four switches? Why, of course, 16!

In general, to realize  $N$ -input Boolean functions, we will need to be employing a  $2^N$ -to-1 multiplexor. Such a multiplexor will take a  $2^N$  input bit vector to realize a *single*  $N$ -input Boolean function. The number of possible functions of  $N$  inputs that can be realized using this multiplexor will, again, be  $2^{2^N}$ !

### Prototyping Large-scale Digital Systems

Figure 2.7 shows a prototyping board from Xilinx/Digilent that uses the principles shown in this chapter. In particular, by downloading bit patterns into the configurable logic blocks of such a circuit, we can make pretty much any

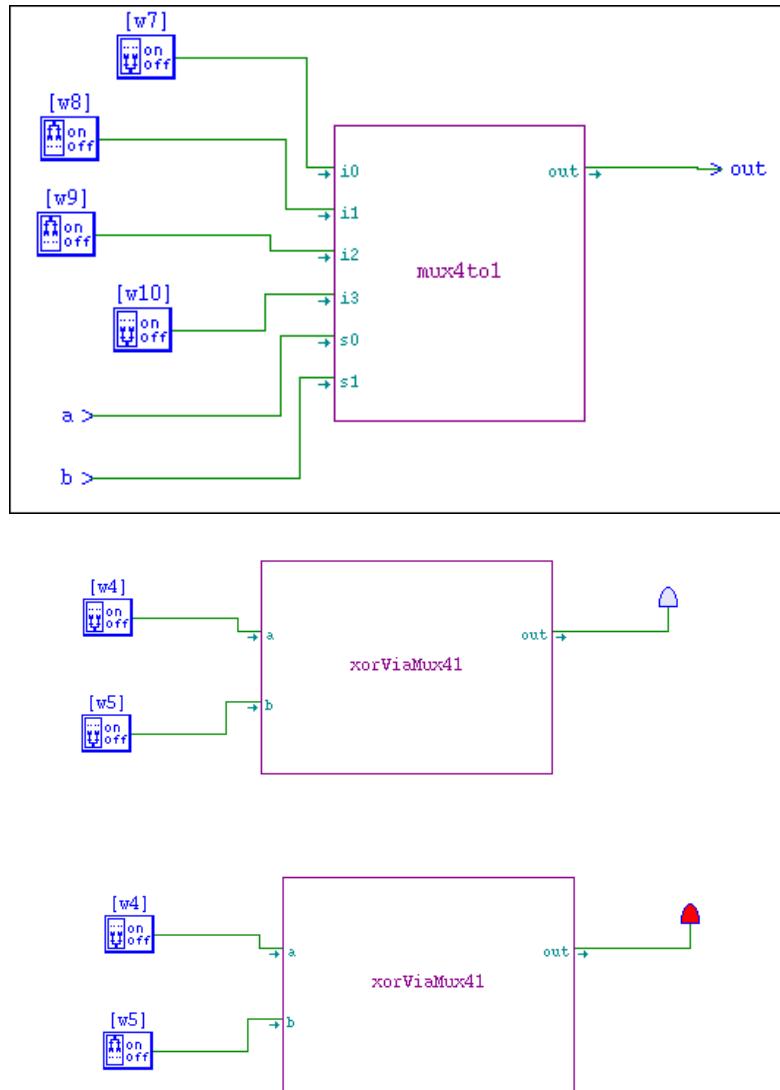


Figure 2.8: An XOR realization (top) and its simulation states (middle, bottom)

digital system. In our research group, we have realized an entire multiprocessor of 8 cores that fits comfortably into such a board.

# Chapter 3

## Review of Boolean Stuff

In this chapter, we will go through the contents of Chapters 1 and 2 by posing problems and solving them. If you have any doubts about Python expressions given here, please evaluate them and thus confirm your understanding.

### 3.1 Short-circuiting Evaluation

This discussion pertains to Short Circuiting and Non Short Circuiting and/or.

There is a variant of the Python and operator, written &. There is also variant of the Python or operator, written |. The variants & and | are non short-circuiting, *i.e.* even if the first argument is evaluated, the second argument will be evaluated. As an example, consider the Python test program:

```
def test():
    print 'fired'
    return True
>>> True | test()
fired!
True

>>> True or test()
True
```

Notice that in the second usage with an or, the printout fired! does not occur. Since True is the first argument of or, it short-circuited the evaluation and returned True without bothering to call function test().

## 3.2 Short-circuiting aids Programming

In programming, one often prefers the short-circuiting approach. For instance, to implement the “quotient greater than one” test shown below, it is handy to have this behavior (using `|`, one will have to write extra code to first test for the denominator being 0).

```
def quotient_gt_one_good(dividend, divisor):
    return (divisor != 0) and (dividend / divisor > 1.0)

def quotient_gt_one_bad(dividend, divisor):
    return (divisor != 0) & (dividend / divisor > 1.0)

>>> quotient_gt_one_good(4, 1)
True

>>> quotient_gt_one_good(4, 0)
False

>>> quotient_gt_one_bad(4, 1)
True

>>> quotient_gt_one_bad(4, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in quotient_gt_one_bad
ZeroDivisionError: integer division or modulo by zero

>>>
```

Suppose Python did not provide the short-circuiting variety of `and` and `or` (namely `and` and `or`). In other words, only `&` and `|` are provided. Rewrite `quotient_gt_one_good` such that it retains its good behavior (without crashing with a “divide by zero” when the divisor is 0). *Hint:* Use an explicit conditional test that avoids division by 0.

## 3.3 Boolean Expression Evaluation

On Page 27, we presented the expression

```
if ((x == 0) and (y < 0)) or (z > w):
```

Provide five different combinations of values for `x`, `y`, `z`, and `w` such that this expression will be true for these values. Make sure that your answer is of this form:

- “For  $x=c1$ ,  $y=c2$ ,  $z=c3$ , and  $w=c4$ , the whole if-condition is rendered true. This happens since `exprn1`, `exprn2`, ... become true.” Here, `exprn1`, `exprn2`, ... refer to the constituent sub-expressions of the if.
- (More such lines, and try to choose a different expression to make true.)

### 3.3.1 Expressions and their Complements

Argue that whenever you find a combination of values for  $x$ ,  $y$ , and  $z$  that makes the expression  $((x \neq 0) \text{ or } (y \geq 0)) \text{ and } (z \leq w)$  true, expression  $((x == 0) \text{ and } (y < 0)) \text{ or } (z > w)$  will be false.

## 3.4 History Readings

Some smart-aleck said that *learning history helps us realize mistakes when we make them again*. Regardless, history provides context and motivation to engage in pursuits that would otherwise seem boring and aimless. Please study the topics suggested below.

### 3.4.1 Boole

Write a paragraph summarizing the work of George Boole. Use web resources to obtain relevant information. You must focus on a section that discusses his algebra (“Boolean algebra”, but Prof. George Boole would not have called it as such.)

### 3.4.2 Shannon

Write two paragraphs summarizing the work of Claude Shannon. More specifically, download his Master’s Thesis [3] (also kept on the class Moodle page), read his section describing these sections, and write summaries in one paragraph each:

- His description of the Delta-Wye (also known as the Delta-Star) transformation
- His description of the construction of a voting machine.

### 3.4.3 Boolean Algebra versus Propositional Calculus

Summarize the connections between Boolean algebra and propositional calculus in a paragraph. Focus on who uses them and for what purposes.

### 3.4.4 Errors in Microprocessors

Read the article <http://www.trnicely.net/pentbug/pentbug.html> to appreciate the fact that incorrectly functioning microprocessors may wreak havoc on the modern society. Imagine using such a microprocessor in the design of a modern aircraft or in calculating the ingredients for a life-saving drug.

### 3.4.5 Futility of Brute-Force Testing

Still not convinced, Prof. Krapshootix Bugnuke decides to use random testing of 64-bit adders. Unfortunately, Prof. Bugnuke has been handed a microprocessor that is fine except for 90 times 90, it returns 6300. Calculate the probability of generating this exact input. All 64-bit combinations have to be tested to ensure that the adder is entirely bug-free.

## 3.5 Practice Defining Gates

Construct a series of diagrams as in Figure 1.2 for an X-gate where X is NOR, XOR, OR, and AND. For achieving this, you must employ TkGate and arrange for these gates to be simulated. Next, recreate Figure 1.3 also for these gates. For achieving this, you must write down a truth-table (as on the left-hand side of Figure 1.3), a definition using Python conditionals (as in the middle), and a hash-table based description (as on the right-hand side).

We will provide a solution for XOR in Figure 3.1.

## 3.6 Nand using Nor

Realize a NAND gate using a collection of NOR gates. Hint: use De Morgan's laws and methods to use inverters using NOR gates. Use no more than four NOR gates.

x	y	z	def xor(x,y): return {0,0}:0, (0,1):1, (1,0):1, (1,1):0}[(x,y)]
0	0	0	>> xor(0,0) 0
0	1	1	>> xor(1,1)                          >> xor(0,0) 0    0
1	0	1	>> xor(0,1)                          >> xor(1,1) 1    0
1	1	0	>> xor(1,0)                          >> xor(0,1) 1    1 1    >> xor(1,0) 1    1

Figure 3.1: The XOR Truth-table, Lambda Representation, and Hash-table Representation

Given inputs  $a$  and  $b$ , one can describe the functionality of a NAND gate over these inputs as  $\overline{a.b}$ . Using De Morgan's laws, this can be rewritten as  $\overline{\overline{a} + \overline{b}}$ . This is nothing but  $nor(P,P)$  where  $P = nor(abar,bbar)$  where  $abar = nor(a,a) = \overline{a}$  and  $bbar = nor(b,b) = \overline{b}$ . The resulting circuit is shown in Figure 3.2.

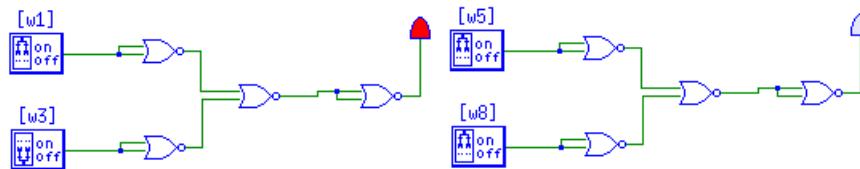


Figure 3.2: NAND using NOR

## 3.7 Nor Using Nand

Realize a NOR gate using a collection of NAND gates. Hint: use De Morgan's laws and methods to use inverters using NAND gates. Use no more than four NAND gates.

## 3.8 Counting Multi-output Functions

Suppose one defines a function of type  $(Bool \times Bool) \mapsto (Bool \times Bool)$ . For example,

```

def nand_nor(x,y):
    def nand(x,y):
        return {(0,0):1,
                 (0,1):1,
                 (1,0):1,
                 (1,1):0}[(x,y)]
    def nor(x,y):
        return {(0,0):1,
                 (0,1):0,
                 (1,0):0,
                 (1,1):0}[(x,y)]
    return (nand(x,y), nor(x,y))

```

How many functions of this signature exist?

There are  $2^{2^2} \times 2^{2^2}$  such functions or actually  $2^8 = 256$  such functions. This is because the “first function” and the “second function” that determine the paired outputs can be independently selected.

### 3.9 Xor Using Nor

Implement an XOR gate using NOR gates. Use De Morgan’s Laws and methods to make inverters from NOR gates.

$xor(a,b) = a.\!b + \!a.b$ . This can be written as  $a.\!b = \!(\!a + b)$  while  $\!a.b = \!(a + \!\!b)$ . The final circuit puts these together using a NOR and an inverter. The resulting circuit is shown in Figure 3.3

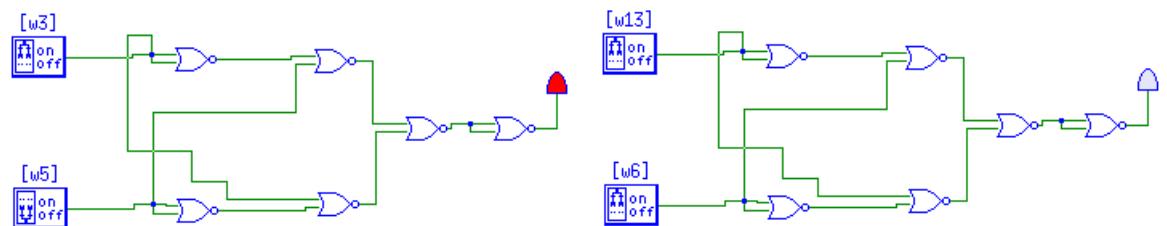


Figure 3.3: XOR using NOR

## 3.10 Universality Proofs

Show that  $\{Nor\}$  forms a universal set of gates, while  $\{XNOR, XOR\}$  is not a universal set. Also, show that *and* by itself is **not** universal.

We can show that AND is not universal by enumerating all possible Boolean functions that AND gates can help realize. Here is how the construction goes for inputs  $x$  and  $y$ :

- We can take an AND gate and feed it 1, 0,  $x$ , or  $y$ . The resulting set of functions are
  - $x.y$
  - $x.1 = x$
  - $x.0 = 0$
  - $y.1 = y$
  - $y.0 = 0$
- Adding one more AND gate in all possible ways, we can obtain no more functions.
- We find that we never exit the set; we *saturate* (or close off) into this set.

$$AND_{realized} = \{1, 0, x.y, x, y\}$$

- Since none of these are a 2-to-1 mux, AND is not universal.

## 3.11 Universal Gates

In this section, we will show that the Implication gate described in Figure 1.5 is universal.

Let the implication gate be  $imp(a, b)$  or  $(a \Rightarrow b)$ . We can realize a 2-to-1 mux as follows:

- 2-to-1 mux: Realized by  $(s \Rightarrow i1).(!s \Rightarrow i0)$
- Inversion of  $x$ : Realized by  $(x \Rightarrow 0)$ .
- And gate for  $and(p, q)$ : Realized by  $((p \Rightarrow !q) \Rightarrow 0)$ .
- Putting it all together, we get
  - $mux21 = and(P, Q)$
  - $P = (s \Rightarrow i1)$
  - $Q = (negs \Rightarrow i0)$
  - $negs = (s \Rightarrow 0)$
  - $and(P, Q) = ((P \Rightarrow negQ) \Rightarrow 0)$
  - $negQ = (Q \Rightarrow 0)$

See Figure 3.4 for details of this circuit.

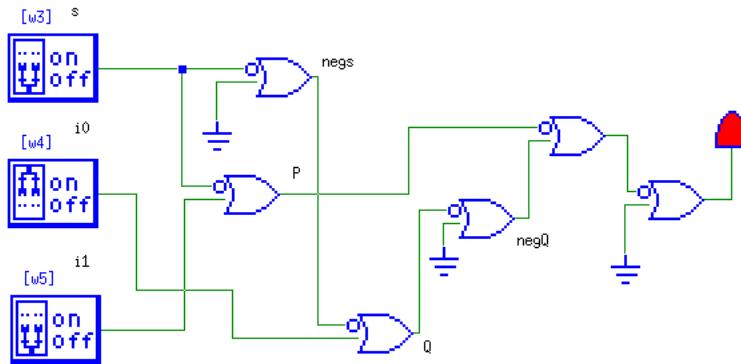


Figure 3.4: mux21 using Imp

### 3.11.1 Alternative Proof

Show that the Implication gate described in Figure 1.5 is universal (alternate proof):

Use  $\Rightarrow$  to realize a NAND gate as follows.

- $nand(P, Q) = \neg P \text{ and } Q$
- $\neg P \text{ and } Q = P \Rightarrow Q$
- $P \Rightarrow Q = ((P \Rightarrow Q) \Rightarrow 0)$
- $newQ = (Q \Rightarrow 0)$

Once a NAND is realized, we have a universal gate. The NAND realized using IMP gates is shown in Figure 3.5.

### 3.11.2 Showing NAND and NOR to be Universal

In §2.3.3, we showed that NAND is universal. We also have introduced De Morgan's law which allows you to derive a NOR using NAND. This immediately allows you to conclude that both NAND and NOR are also universal. **Thus, to summarize, here is how you tell whether a gate is universal or not:**

- Using the gate and Boolean identities, realize a NAND, NOR or MUX21.
- If you succeed, you are done.
- If not, read §3.11.3 to see whether the given gate is *not* universal.

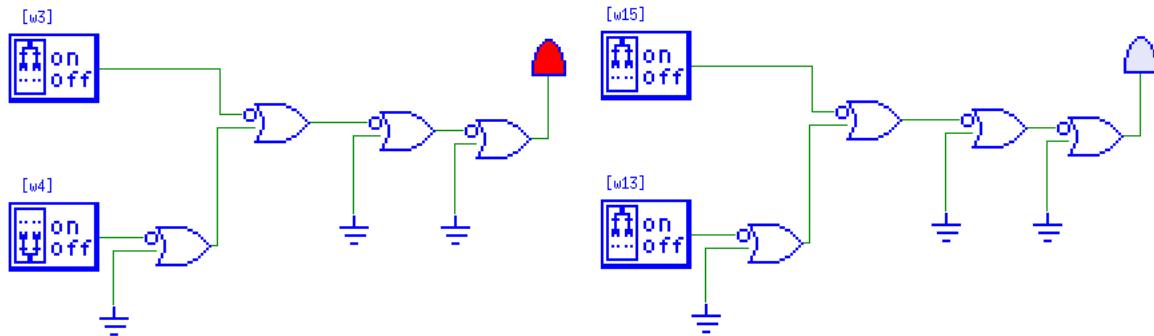


Figure 3.5: NAND using IMP

### 3.11.3 Showing that a gate is NOT Universal

Let us consider 2-input Boolean functions in developing this approach (the proof generalizes easily). A set of 2-input gates  $S$  is not universal if, given the Boolean inputs  $x$  and  $y$  and an unlimited supply of gates from  $S$ , we cannot realize any of the universal gates. If you suspect that a gate is *not* universal, here is how you can systematically proceed:

- Start with inputs  $x, y$ , and have also constants 0 and 1 around. Call this set  $R$  (“already realized gates”). Thus,  $R = \{x, y, 0, 1\}$ .
- Take any two members  $p$  and  $q$  of  $R$  and any gate  $g \in S$ . Apply the logic of  $g$  to  $p$  and  $q$ , and deposit the result back into  $R$  after simplification. Thus, add back  $g(p, q)$ .
- If  $R$  saturates (does not grow further) without having realized a universal gate, the original set  $S$  is not universal.

Here is an illustration using  $S = \{\cdot, +\}$  (where  $\cdot$  is the same as  $\wedge$  and  $+$  the same as  $\vee$ ):

- $R = \{0, 1, x, y\}$
- $R = \{0, 1, x, y, x \cdot y\}$
- $R = \{0, 1, x, y, x \cdot y, x + y\}$
- Now generate  $x \cdot y + y$ , which simplifies to  $y$ .
- Now generate  $(x + y) \cdot x$ , which simplifies to  $x$ .
- Now generate  $(x + y) \cdot x \cdot y$ , which simplifies to  $x \cdot y$ .
- No more possible new entries in  $R$ , and we stop without realizing NAND or NOR.
- Hence  $S = \{\cdot, +\}$  is not universal.

## 3.12 Boolean Identities and Simplification

### 3.12.1 A Simple Identity

Establish the identity  $x + !x.y = x + y$ .

Answer:

$$\begin{aligned} & (x + !x.y) \\ &= (x + !x).(x + y) \\ &= 1.(x + y) \\ &= x + y \end{aligned}$$

### 3.12.2 Boolean Simplification

Simplify the expression by moving the negation operator innermost, applying the De Morgan's law, and applying propositional identities.  $!(x \Rightarrow y) + (x.y)$

Answer:

$$\begin{aligned} & !(x \Rightarrow y) + (x.y) \\ &= !(\neg x + y) + (x.y) \\ &= (x.\neg y) + (x.y) \\ &= x.(\neg y + y) \\ &= x \end{aligned}$$

# Chapter 4

## Sets

Consider the real-world situation wanting to record which of three cities one has visited. One can employ a data structure called a *set* for this purpose. Specifically for this problem, we will maintain a *visited set* (VS) to record the cities visited. If one visits a city more than once, VS is not expected to remember that: either you've visited the city, or not; that is all a set cares to record.

Mathematically, sets are defined *over* domains or universes (or types of objects). Let  $D$  be the universe or value-domain (or simply “domain”) or type called Natural number (which we write  $\text{Nat} = \{0, 1, 2, \dots\}$ ). *By convention, domains are non-empty, but can be infinite* (e.g.,  $\text{Nat}$  is infinite). Now we can define sets *over*  $D$ . Each set contains some, none, or all of the members of  $D$ . For example,  $s_1 = \{1, 2\}$ ,  $s_2 = \{1, 3, 2\}$ ,  $s_3 = \text{Nat}$ ,  $s_4 = \{0, 2, 4, 6, \dots\}$ ,  $s_5 = \{1, 3, 5, 7, \dots\}$ , and  $s_6 = \{\}$  are all sets of natural numbers (sets over  $\text{Nat}$ ). Note that  $s_3$ ,  $s_4$ , and  $s_5$  are infinite while  $s_6$  is empty.

Computationally, sets are implemented in various ways. For example, consider sets definable over the universe (or domain) of  $\{1, 2, 3\}$ . One can then represent all possible sets definable over this domain through three bits, treating them as on/off switches. The empty set  $\{\}$  is represented by keeping all switches off, the set  $\{1, 2, 3\}$  represented by keeping all switches on, and (say) set  $\{2\}$  by keeping just the second switch on. There is no way to record that some of the elements occur *multiple* number of times: there is exactly one switch associated with each element that helps record whether an item is present or not.

## 4.1 All of Mathematics Stems from Sets

Sets may contain other sets. In fact, this little story shows you how, if you have sets, you don't even need numbers! To drive this point home, consider the story of Professor Sayno Toplastix who wants to illustrate to his class how numbers are represented using sets, which he simulates using supermarket plastic bags he recycles (good use of recycled bags to teach math). Prof. Toplastix shows the class “Look, 0 is represented by this empty plastic bag (he inflates and explodes the bag for emphasis; he pops it so that it truly would models  $\emptyset$  that can't hold anything any more). Representing 1 takes two bags: it is modeled by a bag within a bag. 2 needs 4 bags: it is a bag containing (i) an empty bag *i.e.* 0, and (ii) a bag containing an empty bag *i.e.*, 1. You can now wonder how many plastic bags are needed to represent 8 in this fashion. The answer will turn out to be  $2^8$ . More specifically, consider natural numbers (the set  $\{0, 1, 2, \dots\}$ ):

- 0 is modeled as  $\{\}$ , the empty set;
- 1 is modeled as  $\{\{0\}\}$ , or  $\{\{\}\}$ , the set containing 0;
- 2 is modeled as  $\{\{0, 1\}\}$ , or  $\{\{\}, \{\{\}\}\}$ ;
- 3 is modeled as  $\{\{0, 1, 2\}\}$ , or  $\{\{\}, \{\{\}\}, \{\{\}, \{\{\}\}\}\}$ ; and so on.

This exponentially growing number of bags is of no real concern to a mathematician; all they care is that one can represent everything using sets, *i.e.*, numbers are a derived concept. *All* of mathematics can be derived from set theory (more precisely, the Zermelo-Frankel Set theory with the axiom of Choice – often abbreviated ZFC).

## 4.2 Cardinality (or “size”) of sets

An empty set can be written in two ways:  $\{\}$  and also  $\emptyset$ . Both forms can be used interchangeably, but prefer the form that makes it easier to read. For instance, if asked to write “a set containing an empty set,” we would encourage you to write it as  $\{\emptyset\}$  and not  $\{\{\}\}$ . (Of course, all rules have justifiable exceptions!) At any rate,  $\emptyset$  has cardinality 0 because it has *nothing* in it (as Prof. Toplastix has already told you!).

The cardinality of a (finite) set is its size expressed as a number in *Nat*. We will not define the notion of cardinality (at this point, at least) for infinite sets. The operator used is  $|S|$ . For example,  $|\{\}| = 0$  and  $|\{2, 3, 1\}| = 3$ . Notice that we have seen some sets defined by `range()` in Python. For instance,

`set(range(3))` is the set  $\{0, 1, 2\}$ . We have to wrap the `range(3)` call inside a `set()` call; otherwise, we are left with a *list*, not a set.

**NOTE:** I deliberately change around the listing order of the contents of a set—to prevent you from taking advantage of this order. Thus,  $\{1, 2, 3\}$ ,  $\{2, 1, 3\}$ ,  $\{3, 2, 1\}$  are all the same set. By the same token, *please don't take the `str()` (string of) operation of a set and then assume that two equal sets have the same string representation. They often don't!*

## 4.3 Operations on Sets

The basic set operations are the following (please try these in Python also):

- Union, written  $s_1 \cup s_2$  or return `S1 | S2`.  
 Example:  $\{1, 2\} \cup \{1, 3\}$  or  $\{1, 2\} \mid \{1, 3\}$   
 resulting in  $\{3, 1, 2\}$  or  $\{3, 2, 1\}$ .
- Intersection, written  $s_1 \cap s_2$  or return `S1 & S2`.  
 Example:  $\{1, 2\} \cap \{1, 3\}$  or  $\{1, 2\} \& \{1, 3\}$   
 resulting in  $\{1\}$  or  $\{1\}$ .  
 Example:  $\{1, 2\} \cap \{4, 3\}$  or  $\{1, 2\} \& \{4, 3\}$   
 resulting in  $\{\}$  or  $\{\}$ .
- Difference or subtraction written  $s_1 \setminus s_2$  or return `S1 - S2`.  
 Example:  $\{1, 2\} \setminus \{1, 3\}$  or  $\{1, 2\} - \{1, 3\}$   
 resulting in  $\{2\}$  or  $\{2\}$ .  
 Example:  $\{1, 2\} \setminus \{4, 3\}$  or  $\{1, 2\} - \{4, 3\}$   
 resulting in  $\{1, 2\}$  or  $\{1, 2\}$ .  
 Example:  $\{1\} \setminus \{2, 3\}$  or  $\{1\} - \{2, 3\}$   
 resulting in  $\{1\}$  or  $\{1\}$ .  
 Example:  $\{1\} \setminus \{1, 2\}$  or  $\{1\} - \{1, 2\}$   
 resulting in  $\{\}$  or  $\{\}$ .
- Now, *symmetric difference* written `return S1 ^ S2` in Python has the standard mathematical symbol of  $\Delta$ .  $s_1 \Delta s_2$  stands for  $(s_1 \setminus s_2) \cup (s_2 \setminus s_1)$ .  
 Example:  $\{1, 2\} \Delta \{1, 3\}$  or  $\{1, 2\} \wedge \{1, 3\}$   
 resulting in  $\{2, 3\}$  or  $\{2, 3\}$ .  
 Example:  $\{1, 2\} \Delta \{4, 3\}$  or  $\{1, 2\} \wedge \{4, 3\}$   
 resulting in  $\{1, 4, 2, 3\}$  or  $\{2, 1, 3, 4\}$ .
- The *complement* of a set is defined with respect to a universal set  $U$ . Its mathematical operator is written as an “overbar.”  
 Formally, given a set  $S$  and a universal set (or universe)  $U$ , the com-

plement of set  $S$  with respect to  $U$  is given by  $U \setminus S$  (or  $U - S$ ). For instance, with respect to  $U = \text{Nat}$ ,  $\overline{s_4} = s_5$ . In many problems, you will be given a universal set  $U$  that is finite (and quite small). Regardless, you always subtract the set  $S$  from  $U$  using the set subtraction operator in order to *complement*  $S$ .

In a Venn diagram, the universal set  $U$  is drawn as an all-encompassing rectangle. For example, in Figure 5.2, the universe is shown, and the complement of set  $A$  with respect to the universe is the region within this rectangle that is outside of circle  $A$ .

We will rarely (at least in CS 2100) perform a complement operation *in Python*. The main reason is that complementation is often used when the domain is infinite—and representing infinite domains is somewhat non-trivial (hence skipped) in Python. Mathematics, on the other hand, has no such issues.

Notice the spelling: it is **complement** and not *compliment*.<sup>1</sup>

- The subset operation is written  $\subseteq$  ( $<=$ ) and the proper subset operation is written  $\subset$  ( $<$ ).

Example:  $\{1, 2\} \subseteq \{1, 2\}$  or  $\{1, 2\} <= \{1, 2\}$   
resulting in *true* or *True*.

Example:  $\{1, 2\} \subseteq \{1, 2, 3\}$  or  $\{1, 2\} <= \{1, 2, 3\}$   
resulting in *true* or *True*.

Example:  $\{\} \subseteq \{1, 2, 3\}$  or  $\{\} <= \{1, 2, 3\}$   
resulting in *true* or *True*.

Example:  $\{1, 2, 3, 4\} \subseteq \{1, 2, 3\}$  or  $\{1, 2, 3, 4\} <= \{1, 2, 3\}$   
resulting in *false* or *False*.

Example:  $\{1, 2\} \subset \{1, 2\}$  or  $\{1, 2\} < \{1, 2\}$   
resulting in *false* or *False*.

Example:  $\{1, 2\} \subset \{1, 2, 3\}$  or  $\{1, 2\} <= \{1, 2, 3\}$   
resulting in *true* or *True*.

Example:  $\{\} \subset \{1, 2, 3\}$  or  $\{\} <= \{1, 2, 3\}$   
resulting in *true* or *True*.

Example:  $\{1, 2, 3, 4\} \subset \{1, 2, 3\}$  or  $\{1, 2, 3, 4\} < \{1, 2, 3\}$   
resulting in *false* or *False*.

- The superset operation is written  $\supseteq$  ( $>=$ ) and the proper superset operation is written  $\supset$  ( $>$ ).

---

<sup>1</sup>The latter is what I will do if you earn an A grade in this course. The former is what you do to “flip a set.”

Now,  $A \subseteq B$  if and only if  $B \supseteq A$ .

Now,  $A \subset B$  if and only if  $B \supset A$ .

Please infer the related facts about the Python operators. **Try it out.**

- **Almost everything** we define for sets also applies equally to lists.

**Try it out.**

Here is a terminal session illuminating a few things (notice that by default, `range()` creates a list):

```
>>> set(range(2)) <= {0,1}
True
```

```
>>> set(range(2)) >= {0,1}
True
```

```
>>> range(2) == {0,1}
False
```

```
>>> range(2) == [0,1]
True
```

## 4.4 Comprehension, Characteristic Predicates

In this section, we define set operations through mathematical logic. You may wish to review the set operations defined in §4.3. The general approach we take is to *define* a set  $S$  using a characteristic predicate  $p$ . In other words, we will write

$$S = \{x \mid p(x)\}$$

Let us illustrate this notation on set operations.

- We begin with set union. Union, written  $S_1 \cup S_2$  can be defined as follows:

$$S_1 \cup S_2 = \{x \mid x \in S_1 \vee x \in S_2\}$$

It can be observed that the meaning of the operation  $\cup$  on sets is captured using the logical connective  $\vee$ .

- Intersection, written  $S_1 \cap S_2$  can be defined as follows:

$$S_1 \cap S_2 = \{x \mid x \in S_1 \wedge x \in S_2\}$$

The set operation  $\cap$  modeled using the logical connective  $\wedge$ .

- Difference or subtraction written  $S_1 \setminus S_2$  can be defined as follows:

$$S_1 \setminus S_2 = \{x \mid x \in S_1 \wedge x \notin S_2\}$$

- Symmetric difference  $S_1 \Delta S_2$  can be defined as follows:

$$S_1 \Delta S_2 = \{x \mid (x \in S_1 \wedge x \notin S_2) \vee (x \in S_2 \wedge x \notin S_1)\}$$

- $S_1 \subseteq S_2$  can be defined as follows:

$$S_1 \subseteq S_2 \text{ exactly when for all } x : (x \in S_1 \Rightarrow x \in S_2)$$

- Suppose the characteristic predicate for  $S_1$  is  $p_1$  and that for  $S_2$  is  $p_2$ . Then  $S_1 \subseteq S_2$  exactly when  $p_1(x) \Rightarrow p_2(x)$ . This is because for every  $x \in S_1$ —that is  $p_1(x)$ , we have  $x \in S_2$ —that is,  $p_2(x)$ . This matches the meaning of  $\Rightarrow$ .

As a memory aid, flip the direction of  $\subseteq$  to arrive at  $\Rightarrow$ , i.e.,  $S_1 \subseteq S_2$  exactly when, *with respect to their characteristic predicates*, “ $S_1 \Rightarrow S_2$ .”

- $S_1 \subset S_2$  can be defined as follows:

$$S_1 \subset S_2 \text{ exactly when } (S_1 \subseteq S_2) \text{ and } (S_1 \neq S_2)$$

In terms of characteristic predicates, we still have  $p_1 \Rightarrow p_2$ , and  $p_1 \neq p_2$ .

#### 4.4.1 Set Comprehension in Python

A set comprehension expression of Python is shown below:

```
{(x,y) for x in {1,2,3} for y in {"he", "she"} }
```

Here,  $(x,y)$  is called a template expression, and the part `for x in 1,2,3` for `y in "he", "she"` is called the quantifier expression. It can be seen that these ideas correspond directly to those introduced in §4.4.

## 4.5 Cartesian Product

We now introduce a set operator called *cartesian product* (some books call this the “cross product”). Given two sets  $A$  and  $B$ , their cartesian product  $A \times B$  is defined as follows:

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

The notation above defines all pairs  $(x, y)$  such that  $x$  belongs to  $A$  and  $y$  belongs to  $B$ . To understand cartesian products, we can readily obtain some practice with Python:

```
>>> { (x,y) for x in {1,2,3} for y in {11,22} }
set([(1, 22), (3, 22), (2, 11), (3, 11), (2, 22), (1, 11)])

>>> { (x,y) for x in {10,20,30} for y in {"he", "she"} }
set([(10, 'he'), (30, 'she'), (20, 'she'), (20, 'he'), (10, 'she'), (30, 'he')])

>>> { (x,y) for x in {} for y in {"he", "she"} }
set([])
```

### 4.5.1 Cardinality of a Cartesian Product

Notice that the cardinality of the cartesian product of two sets  $S_1$  and  $S_2$  equals the product of the cardinalities of the sets  $S_1$  and  $S_2$ . For this reason, if one of the sets is empty, the cartesian product results in an empty set (we can't find an  $x \in A$  any more).



# Chapter 5

## Venn Diagrams, and Proofs of Set Identities

John Venn, the English mathematician of the 19th century evolved a convention for depicting sets and their relationships that has acquired the name “Venn diagrams.” A good illustration of the use of Venn diagrams is given in [9], a web article: The distinction The distinction between “Tiffany likes

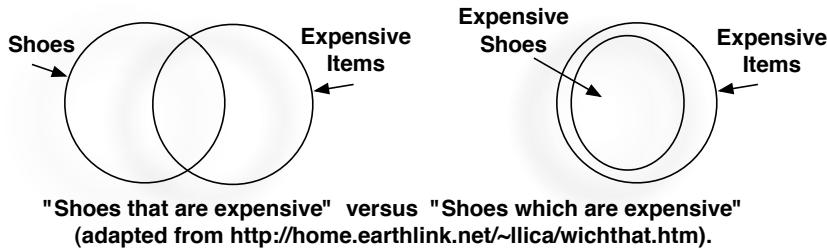


Figure 5.1: “That” versus “Which” in English usage

*shoes that are expensive*” and “*Tiffany likes shoes, which are expensive*”(notice the comma after “shoes”) is best captured by a Venn diagram as in Figure 5.1 on Page 73. The former looks for common elements between “Shoes” and “Expensive items” whereas the latter looks for “Expensive items” and finds a subset within it called “Expensive Shoes.” According to this webpage, the general rule is: *If a clause describes the whole set of the term it modifies, the clause in question should be introduced with which and separated by one or two commas from the rest of the sentence. (This is a nonrestrictive*

clause.) If the clause describes only a subset of the term it modifies, then the clause in question should be introduced by that and should not be separated by commas. (This is a restrictive clause.).

We will of course not be delving too much into English grammar in this course, but it is good to know that Venn diagrams can come in handy even to disambiguate English constructions in technical writing. We will be studying Venn diagrams more in depth later in this chapter.

## 5.1 Venn Diagrams

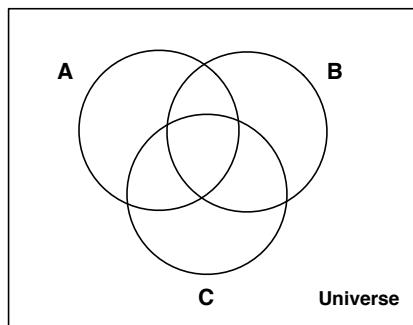


Figure 5.2: The Familiar Venn Diagram of 3 sets

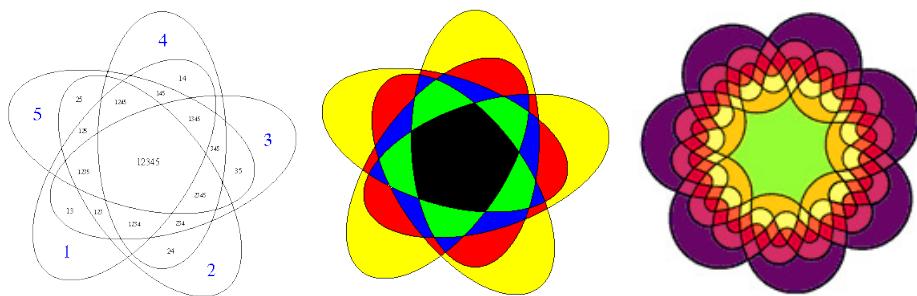


Figure 5.3: Venn Diagrams of order 5 (left); of order 5 with regions colored (middle); and order 7 (right). Images courtesy of <http://mathworld.wolfram.com/VennDiagram.html> and <http://www.theory.csc.uvic.ca/~cos/inf/comb/SubsetInfo.html#Venn>.

Venn diagrams are one of the most widely used of notations to depict sets and their inclusion relationships. Usually one draws the “universal set” as a rectangle, and within it depicts closed curves representing various sets. I am sure you have seen simple venn diagrams showing three circles representing three sets  $A$ ,  $B$ , and  $C$ , and showing all the regions defined by the sets (e.g., Figure 5.2 on Page 74) namely: the eight sets:  $A \cap B \cap C$  (points in all three sets),  $A \cap B$ ,  $B \cap C$ , and  $A \cap C$  (points in any two sets chosen among the three), and then  $A$ ,  $B$ , and  $C$  (points in the three individual sets), and finally  $\emptyset$  (points in no set at all—shown outside of the circles).

Venn diagrams are schematic diagrams used in logic theory to depict collections of sets and represent their relationships [15, 18]. More formally, an order- $N$  Venn diagram is a collection of simple closed curves in the plane such that

1. The curves partition the plane into connected regions, and
2. Each subset  $S$  of  $\{1, 2, \dots, N\}$  corresponds to a unique region formed by the intersection of the interiors of the curves in  $S$  [13].

Venn diagrams involving five and seven sets are beautifully depicted in these websites, and also the associated combinatorics is worked out. Two illustrations from the latter site are shown in Figure 5.3 on Page 74, where the colors represent the number of regions included inside the closed curves.

### 5.1.1 The number of regions in a Venn diagram

While we are discussing Venn diagrams, it is a good idea to start counting the number of regions described by them. This section will also provide us with a taste of how counting using permutations and combinations works in practice. We shall expand on this topic in a significant way in Chapter 10.

Coming back to Venn diagrams, notice that in each of the Venn diagram involving  $N$  sets (an *order N* Venn diagram) shown in full generality,

- There is one region that is not in any set at all
- There are regions that belong to exactly one set
- There are regions that belong to two sets (where these sets are chosen in all possible ways)
- ...
- There are regions belonging to  $k$  of the  $N$  sets.
- ...
- There is one region that belongs to *all N* sets.

How many ways are there to pick  $k$  items (sets) out of  $N$  items (sets)? The way to think about this is as follows, using Pool-table balls as a visualization aid. We introduce two ideas here, namely *Permutations* and *Combinations*.

### Permutations

Imagine there being Pool-table balls numbered  $1 \dots N$  and you are asked to line them up on a straightline<sup>1</sup> Well, you have any of those  $N$  balls that you can begin with. Thereafter, you have  $N - 1$  choices for the second ball, and so on, and when you come to the last ball, you have only one choice. The product  $N \times (N - 1) \times \dots \times 2 \times 1$  is of course the familiar  $N!$  function.

### Combinations

Now imagine there being **black** Pool-table balls numbered  $1 \dots N$  inside a black cloth bag (yes, a new type of Pool with only black balls, with only numbers distinguishing them). Suppose you are allowed to draw out  $k$  of these  $N$  balls<sup>2</sup>

- You might any  $k$  out of the  $N$  balls.
- You don't care about the arrangement of these  $k$  balls. These  $k$  balls feel lucky to have been chosen, not caring how they line up within that "lucky group."

We can break this problem into stages and arrive at the desired formula. First, view this problem as a "line-up" situation where we again aim to line-up  $k$  out of these  $N$  balls:

- $N$  choices for the first of the  $k$  balls.
- $(N - 1)$  choices for the second of the  $k$  balls.
- ...
- $(N - k + 1)$  choices for the  $k$  ball.

This calculation itself yields  $N \times (N - 1) \times \dots \times (N - k + 1)$ . OK, given that we don't care about the line-up itself, we could have gone this way also:

- $N$  choices for the *second* of the  $k$  balls.
- $(N - 1)$  choices for the *first* of the  $k$  balls.

In other words, we don't care about the line-up of the  $k$  balls. It is as if the  $k$  balls emerged with one specific line-up, proudly displaying their numbers

<sup>1</sup>Say, getting ready for the ultimate trick shot.

<sup>2</sup>Also imagine having a large hand to be able to do so.

(“numbering within the lucky group”), and then you peeled off their numbers!

- Thus, you have  $N \times (N - 1) \times \dots \times (N - k + 1)$  ways of obtaining *different line-ups of  $k$  balls*, and
  - You choose to forget the different  $k!$  line-ups that end up being created.
- So finally, the formula we are staring at is called “ $N$  choose  $k$ ,” and is written as  $\binom{N}{k}$ . This formula is

$$\binom{N}{k} = N!/(k! \cdot (N - k)!).$$

**Illustration of the Choose Function:** With respect to Figure 5.3 (middle), we can see how this formula works:

- There are  $\binom{5}{1} = 5$  yellow regions belonging to 1 set;
- there are  $\binom{5}{2} = 10$  red regions belonging to 2 sets;
- there are  $\binom{5}{3} = 10$  blue regions belonging to 3 sets;
- there are  $\binom{5}{4} = 5$  green regions belonging to 4 sets; and finally,
- there are  $\binom{5}{5} = 1$  black region belonging to all 5 sets.

### Back to choosing Venn Diagram Regions

Now let’s come back to the number of ways in which one can choose a subset of size  $k$  out of a set of size  $N$ . This turns out to be  $\binom{N}{k}$ . So finally, the number of regions,  $V_N$ , in an order  $N$  Venn diagram is

$$V_N = \sum_{k=0}^N \binom{N}{k} = 2^N$$

(where the region outside the diagram is included in the count, as it corresponds to  $\binom{N}{0}$  which is 1). This derivation comes from the use of the Binomial theorem which says:

$$(x + y)^N = \sum_{k=0}^N \binom{N}{k} x^{N-k} y^k$$

Specializing the binomial theorem for  $x = y = 1$ , we obtain  $2^N$ .

### 5.1.2 Alternative Derivation: Number of Venn Regions

Each region of a Venn diagram either includes a region of a set or not. Imagine employing  $N$  switches (an  $N$ -bit vector) representing an arbitrary region of a Venn diagram.

- If all the switches are *on*, we end up representing the region of the Venn diagram common to all sets (belonging to all the sets). There is exactly one such region.
- If all the switches are *off*, we end up representing the region of the Venn diagram belonging to none of the sets (it is exterior to all the sets).
- There are  $2^N$  switch settings—*ergo*  $2^N$  Venn diagram regions!

**Illustration of the total number of regions in a Venn diagram:** For the Venn diagram Figure 5.3 (middle), there are a total of  $2^5 = 1 + 5 + 10 + 10 + 5 + 1$  regions.

## 5.2 Formal Proofs of Set Identities

Using the above logical definitions of sets, we will now provide proofs for a few important set identities (we also leave a few as exercises).

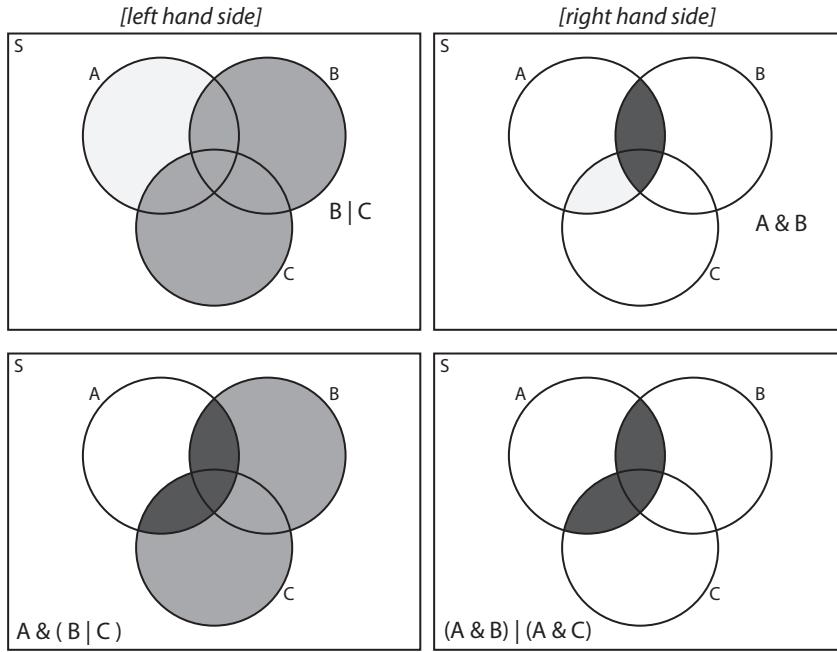
- $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

**A Formal Proof (see Figure 5.4), §5.2.1**

$$\begin{aligned}
 A \cap (B \cup C) &= \{x \mid x \in A \wedge x \in (B \cup C)\} && (\text{definition of } \cap) \\
 &= \{x \mid x \in A \wedge (x \in B \vee x \in C)\} && (\text{definition of } \cup) \\
 &= \{x \mid (x \in A \wedge x \in B) \vee (x \in A \wedge x \in C)\} && (\wedge \text{ distributes}) \\
 &= \{x \mid x \in A \cap B \vee x \in (A \cap C)\} && (\text{definition of } \cap) \\
 &= \{x \mid x \in ((A \cap B) \cup (A \cap C))\} && (\text{definition of } \cup) \\
 &= (A \cap B) \cup (A \cap C)
 \end{aligned}$$

$$\bullet A \cup B = \overline{\overline{A} \cap \overline{B}}$$

**A Formal Proof (see Figure 5.5)**

Figure 5.4: Venn diagram for  $A \cap(B \cup C) = (A \cap B) \cup(A \cap C)$ 

$$\begin{aligned}
 A \cup B &= \{x \mid x \in A \vee x \in B\} && \text{(definition of } \cup\text{)} \\
 &= \{x \mid \neg(\neg(x \in A) \wedge \neg(x \in B))\} && \text{(DeMorgan's Law)} \\
 &= \{x \mid \neg((x \notin A) \wedge (x \notin B))\} && \text{(definition of } \notin\text{)} \\
 &= \{x \mid \neg((x \in \overline{A}) \wedge (x \in \overline{B}))\} && \text{(definition of } \overline{Z}\text{)} \\
 &= \{x \mid \neg(x \in (\overline{A} \cap \overline{B}))\} && \text{(definition of } \cap\text{)} \\
 &= \{x \mid x \notin (\overline{A} \cap \overline{B})\} && \text{(definition of } \notin\text{)} \\
 &= \{x \mid x \in \overline{(\overline{A} \cap \overline{B})}\} && \text{(definition of } \overline{Z}\text{)} \\
 &= \overline{A \cap B}
 \end{aligned}$$

- $(A \Delta B) = (A \cup B) - (A \cap B)$

**A Formal Proof (see Figure 5.6)**

80 CHAPTER 5. VENN DIAGRAMS, AND PROOFS OF SET IDENTITIES

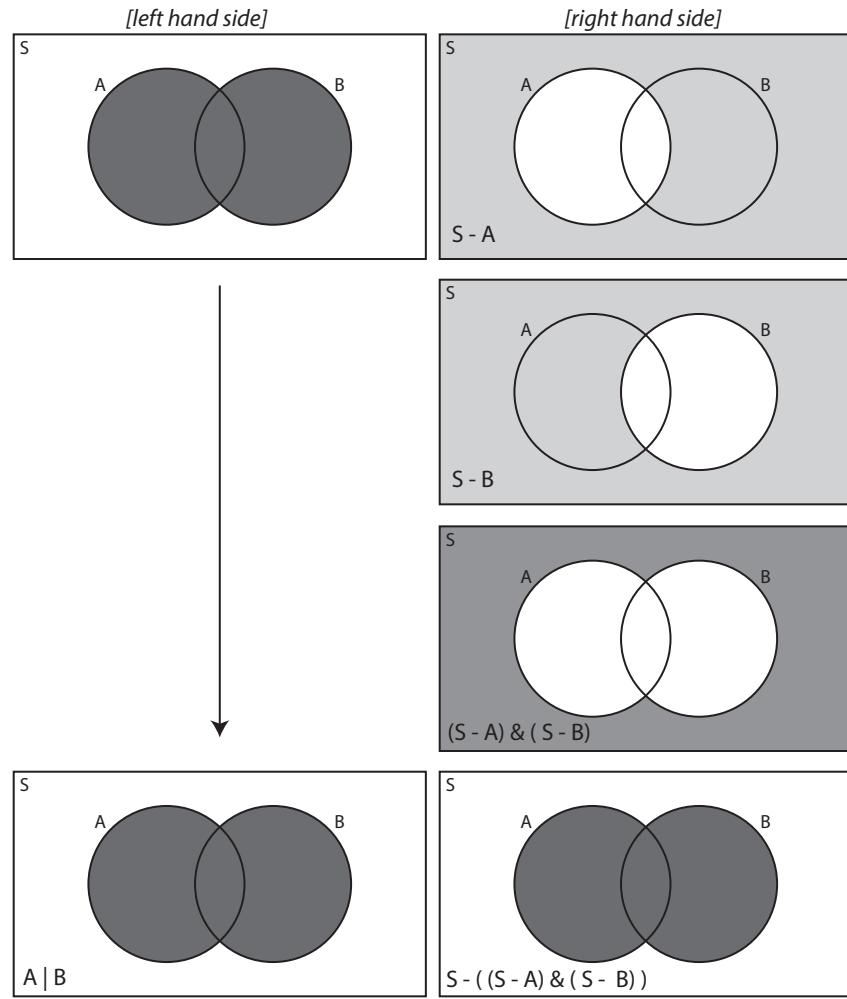


Figure 5.5: Venn diagram for  $A \cup B = \overline{\overline{A} \cap \overline{B}}$

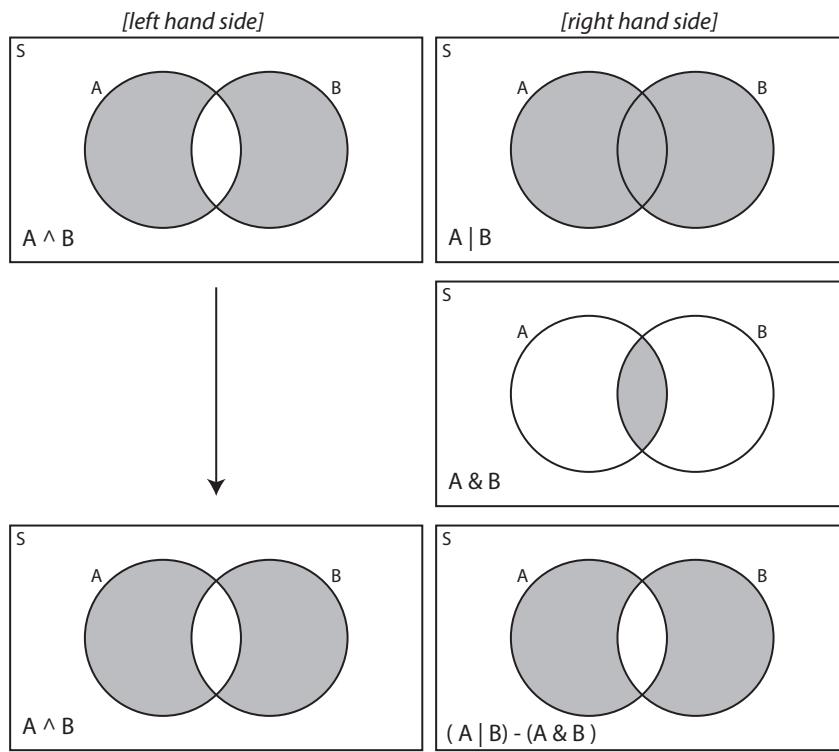


Figure 5.6: Venn diagram for  $(A \Delta B) = (A \cup B) - (A \cap B)$

This one is pretty long. Notes are put below the previous line.

$$\begin{aligned}
 A \Delta B &= \{x \mid (x \in A \wedge x \notin B) \vee (x \in B \wedge x \notin A)\} \\
 &\quad (\text{definition of } \Delta) \\
 &= \{x \mid ((x \in A \wedge x \notin B) \vee x \in B) \wedge ((x \in A \wedge x \notin B) \vee x \notin A)\} \\
 &\quad (\vee \text{ distributes}) \\
 &= \{x \mid ((x \in A \vee x \in B) \wedge (x \in B \vee x \notin B)) \wedge ((x \in A \wedge x \notin B) \vee x \notin A)\} \\
 &\quad (\vee \text{ distributes again, on the left}) \\
 &= \{x \mid ((x \in A \vee x \in B) \wedge \text{true}) \wedge ((x \in A \wedge x \notin B) \vee x \notin A)\} \\
 &\quad (\text{ } p \vee \neg p \text{ is always true}) \\
 &= \{x \mid (x \in A \vee x \in B) \wedge ((x \in A \wedge x \notin B) \vee x \notin A)\} \\
 &\quad (\text{ } p \wedge \text{true} \text{ has the same truth value as } p) \\
 &= \{x \mid (x \in A \vee x \in B) \wedge ((x \in A \vee x \notin A) \wedge (x \notin B \vee x \notin A))\} \\
 &\quad (\vee \text{ distributes again, on the right}) \\
 &= \{x \mid (x \in A \vee x \in B) \wedge (\text{true} \wedge (x \notin B \vee x \notin A))\} \\
 &\quad (\text{ } p \vee \neg p \text{ is always true}) \\
 &= \{x \mid (x \in A \vee x \in B) \wedge (x \notin B \vee x \notin A)\} \\
 &\quad (\text{true} \wedge p \text{ has the same truth value as } p) \\
 &= \{x \mid (x \in A \vee x \in B) \wedge \neg(x \in B \wedge x \in A)\} \\
 &\quad (\text{DeMorgan's Law}) \\
 &= \{x \mid (x \in A \vee x \in B) \wedge \neg(x \in A \cap B)\} \\
 &\quad (\text{definition of } \cap) \\
 &= \{x \mid (x \in A \cup B) \wedge \neg(x \in A \cap B)\} \\
 &\quad (\text{definition of } \cup) \\
 &= (A \cup B) - (A \cap B) \\
 &\quad (\text{definition of } -)
 \end{aligned}$$

### 5.2.1 Checking the Proofs Using Python

The proof given in §5.2 for  $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$  can be checked in Python as follows. While the checking is being done for specific input sets, it at least gives reassurance that no simple superficial mistakes have been made.

```

# set([2, 4, 6, 8, 10, 12, 14, 16])
A = { i for i in range(2,17) if (i%2 == 0) }

# set([3, 5, 7, 9, 11, 13, 15])
B = { i for i in range(2,17) if (i%2 == 1) }

# set([3, 4, 6, 8, 9, 12, 15, 16])
C = { i for i in range(2,17) if ((i%3 == 0) | (i%4 == 0)) }

# set([2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16])
S = { i for i in range(2,17) }

# ----- BEGIN -- A & (B | C) == (A & B) | (A & C)
#--- Using & and | for set operations ; and and or for logical operations

# LHS
T0 = A & (B | C)

# LHS, written in set comprehension form
T1 = { x for x in S if x in A & (B | C) }

# defn of & on sets : set to logic
T2 = { x for x in S if (x in A) and (x in B|C) }

# defn of | on sets : set to logic
T3 = { x for x in S if (x in A) and ((x in B) or (x in C)) }

# and distributes : in logic
T4 = { x for x in S if ((x in A) and (x in B)) or ((x in A) and (x in C)) }

# defn of & : logic to set
T5 = { x for x in S if (x in A & B) or (x in A & C) }

# defn of | : logic to set
T6 = { x for x in S if x in (A & B) | (A & C) }

# RHS
T7 = (A & B) | (A & C)

# One way to put in assertions
#
assert(T0 == T1 == T2 == T3 == T4 == T5 == T6 == T7), \
      "T0 == T1 == T2 == T3 == T4 == T5 == T6 == T7 VIOLATED!!"

```



# Chapter 6

## Review of Sets

In this chapter, we will go through the contents of Chapters 4 and 5 by posing problems and solving them.

### 6.1 Exercises Pertaining to Sets

1. Let

$$\mathbb{N} = \{0, 1, 2, \dots\}$$

$$P = \{1, 2, 4, 8, 16, \dots\}$$

$$E = \{0, 2, 4, 6, 8, \dots\}$$

$$O = \{1, 3, 5, 7, 9, \dots\}$$

$$T = \{\{1, 2\}, \{2, 4\}, \{3, 6\}, \{4, 8\}, \dots\}$$

For each of the following propositions, indicate whether it is True or False, giving reasons.

- (a)  $(E \cup O) \subset \mathbb{N}$
- (b)  $(E \cap O) \subseteq \{\}$
- (c) With respect to  $D = \mathbb{N}$ ,  $\overline{E} = O$  and  $\overline{O} = E$
- (d)  $\{\} \supseteq (P - E)$

(e)  $P \subseteq T$

2. Draw a Venn Diagram to represent the expression  $(A \cap C) \cup B$ .

## 6.2 Set Comprehension involving Booleans

### 6.2.1 Comprehension Example

List some of the elements and non-elements of this set:

$$\{(x, y) \mid (x > 3 \wedge y > 2) + (y > 3 \Rightarrow x = 5)\}$$

Answer:  $\{(4, 3), (5, 4), \dots\}$

Some pairs not in the set are  $\{(1, 100), (2, 100), (1, 1000), \dots\}$ . These are found out through inspection, and checked in the Python session below.

```
>>> {(x,y) for x in range(10) for y in range(10) if ((x>3) and (y>2)) or (not(y>3) or (x==5)) }
set([
(5, 9), (4, 7), (1, 3), (9, 1), (4, 8), (3, 0), (9, 8), (8, 0), (5, 4), (2, 1), (8, 9), (6, 2),
(9, 3), (9, 4), (5, 1), (0, 3), (8, 5), (5, 8), (4, 0), (1, 2), (9, 0), (9, 5), (4, 9), (3, 3),
(8, 1), (7, 6), (4, 4), (6, 3), (5, 6), (7, 2), (5, 0), (2, 2), (7, 7), (5, 7), (7, 3), (4, 1),
(1, 1), (9, 7), (6, 4), (3, 2), (0, 0), (6, 6), (8, 2), (7, 1), (4, 5), (7, 9), (8, 6), (5, 5),
(6, 0), (7, 5), (2, 3), (8, 7), (6, 8), (4, 2), (1, 0), (9, 6), (6, 5), (5, 3), (0, 1), (6, 9),
(7, 0), (4, 6), (6, 7), (9, 2), (7, 8), (6, 1), (3, 1), (9, 9), (7, 4), (2, 0), (8, 8), (4, 3),
(8, 3), (5, 2), (0, 2), (8, 4)])
```

```
>>> (1,100) in {(x,y) for x in range(2) for y in range(100) if
((x>3) and (y>2)) or (not(y>3) or (x==5)) }
False
>>> (2,100) in {(x,y) for x in range(2) for y in range(100)
if ((x>3) and (y>2)) or (not(y>3) or (x==5)) }
False
```

### 6.2.2 One More Comprehension Example

List some of the elements and non-elements of this set:

$$\{(x, y) \mid (x > 3 \Rightarrow y > 2) + (y > 3 \oplus x = 5)\}$$

Answer using Python, where  $\oplus$  is implemented by the infix operator  $\wedge$ :

```
>>> {(x,y) for x in range(10) for y in range(10) if (not(x>3) or (y>2)) or ((y > 3) ^ (x==5)) }
set([
(9, 3), (5, 9), (4, 7), (1, 3), (4, 8), (3, 0), (2, 8), (9, 8), (7, 8), (5, 4), (2, 1),
(8, 9), (5, 6), (2, 6), (1, 6), (9, 4), (5, 1), (3, 7), (0, 3), (8, 5), (2, 5), (5, 8),
(1, 2), (7, 4), (9, 5), (4, 9), (3, 3), (2, 9), (2, 0), (7, 6), (4, 4), (6, 3), (1, 5),
```

```
(8, 8), (8, 3), (3, 6), (0, 4), (8, 6), (5, 7), (5, 3), (1, 1), (9, 7), (2, 7), (3, 2),
(0, 0), (6, 6), (5, 0), (4, 5), (7, 9), (2, 2), (5, 5), (1, 4), (7, 7), (3, 9), (0, 5),
(0, 7), (8, 7), (6, 8), (6, 4), (6, 7), (1, 0), (0, 8), (9, 6), (6, 5), (3, 5), (0, 1),
(6, 9), (7, 3), (4, 6), (5, 2), (3, 1), (9, 9), (2, 4), (3, 8), (0, 6), (1, 8), (7, 5),
(4, 3), (1, 7), (0, 9), (2, 3), (8, 4), (3, 4), (0, 2), (1, 9]))
```

How to compute what is *not* in the set. First compute "U" which is like a "mini universal set." Then use set subtraction.

```
>>> U = {(x,y) for x in range(10) for y in range(10) }

>>> S = {(x,y) for x in range(10)
           for y in range(10) if (not(x>3) or (y>2)) or ((y > 3) ^ (x==5)) }

>>> U - S
set([(9, 0), (7, 0), (8, 2), (7, 1), (9, 2), (6, 1), (8, 0), (6, 0), (8, 1), (6, 2),
      (9, 1), (4, 2), (4, 1), (7, 2), (4, 0)])

>>> U
set([
(7, 3), (6, 9), (0, 7), (1, 6), (3, 7), (2, 5), (8, 5), (5, 8), (4, 0), (9, 0), (6, 7),
(5, 5), (7, 6), (0, 4), (1, 1), (3, 2), (2, 6), (8, 2), (4, 5), (9, 3), (6, 0), (7, 5),
(0, 1), (3, 1), (9, 9), (7, 8), (2, 1), (8, 9), (9, 4), (5, 1), (7, 2), (1, 5), (3, 6),
(2, 2), (8, 6), (4, 1), (9, 7), (6, 4), (5, 4), (7, 1), (0, 5), (1, 0), (0, 8), (3, 5),
(2, 7), (8, 3), (4, 6), (9, 2), (6, 1), (5, 7), (7, 4), (0, 2), (1, 3), (4, 8), (3, 0),
(2, 8), (9, 8), (8, 0), (6, 2), (5, 0), (1, 4), (3, 9), (2, 3), (1, 9), (8, 7), (4, 2),
(9, 6), (6, 5), (5, 3), (7, 0), (6, 8), (0, 6), (1, 7), (0, 9), (3, 4), (2, 4), (8, 4),
(5, 9), (4, 7), (9, 1), (6, 6), (5, 6), (7, 7), (0, 3), (1, 2), (4, 9), (3, 3), (2, 9),
(8, 1), (4, 4), (6, 3), (0, 0), (7, 9), (3, 8), (2, 0), (1, 8), (8, 8), (4, 3), (9, 5),
(5, 2)])
>>> len(U)
100

>>>
```

## 6.3 Defining Primes via Recursion

Consider these definitions:

- Primes are natural numbers other than 1 that are evenly divisible (without any remainder) exactly by themselves as well as 1.
- Composite numbers are natural numbers that are not prime.

Now, let us define the following set:

$$\{(x,y) \mid \text{odd}(x) \Rightarrow (\text{prime}(x) \wedge \text{composite}(y))\}$$

This set will have the following elements:  $\{(2,0),(2,1),(2,2),(3,4),(7,8),\dots\}$

Some pairs not in this set are  $\{(9,4),(9,9),(9,7),\dots\}$

**Let us have some fun generating these mechanically:**

```

#
# http://en.wikipedia.org/wiki/Primality_test provides far better methods
# for primality testing
#

import math

def primes(N):
    if (N <= 1):
        return []
    elif (N == 2):
        return [2]
    else:
        sq = int(math.ceil(math.sqrt(N)))
        p1 = primes(sq)
        p2 = sieve(range(2,sq+1), range(sq, N+1))
        return p1+p2

def sieve(divs, lst):
    if (divs == []):
        return lst
    else:
        knock1 = knock_off(divs[0], lst)
        return sieve(divs[1:], knock1)

def knock_off(d, lst):
    return filter(lambda x: x%d, lst)

def isPrime(N):
    if (N <= 1):
        return False
    elif (N == 2):
        return True
    else:
        sq = int(math.ceil(math.sqrt(N)))
        p2 = sieve(range(2,sq+1), [N])
        return (p2 != [])

def isComposite(N):
    return not(isPrime(N))

#--- http://primes.utm.edu/lists/small/1000.txt has the first 1000 primes
#--- for comparison

>>> primes(100)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
 79, 83, 89, 97]

>>> L = primes(100000)

>>> len(L)
9592

>>> isPrime(3)
True

>>> isPrime(8)

```

```

False

>>> isComposite(8)
True

>>> map(lambda x: isPrime(x), primes(30))
[True, True, True, True, True, True, True, True, True]

>>> map(lambda x: isComposite(x), primes(30))
[False, False, False, False, False, False, False, False, False]

>>> map(lambda x: isComposite(x), range(20))
[True, True, False, False, True, False, True, False, True, True, False,
 True, False, True, True, False, True, False, True, False]

>>> map(lambda x: isPrime(x), range(20))
[False, False, True, True, False, True, False, True, False, False, True,
 False, True, False, False, True, False, True, False, True]

#--- This is a very strong test because even if there is one mistake, it would reduce to false

>>> reduce(lambda x,y : x and y, map(lambda x: isPrime(x), primes(10000)))
True

#--- This is also a very strong test

>>> reduce(lambda x,y : x or y, map(lambda x: isComposite(x), primes(10000)))
False

#--- NOW WE CAN DO THE EXERCISE ---

[(x,y) for x in range(20) for y in range(20)
 if ((x%2 == 0) or (isPrime(x) and isComposite(y)))]
```

[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9), (0, 10), (0, 11), (0, 12), (0, 13), (0, 14), (0, 15), (0, 16), (0, 17), (0, 18), (0, 19), (2, 0), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (2, 6), (2, 7), (2, 8), (2, 9), (2, 10), (2, 11), (2, 12), (2, 13), (2, 14), (2, 15), (2, 16), (2, 17), (2, 18), (2, 19), (3, 0), (3, 1), (3, 4), (3, 6), (3, 8), (3, 9), (3, 10), (3, 12), (3, 14), (3, 15), (3, 16), (3, 18), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (4, 6), (4, 7), (4, 8), (4, 9), (4, 10), (4, 11), (4, 12), (4, 13), (4, 14), (4, 15), (4, 16), (4, 17), (4, 18), (4, 19), (5, 0), (5, 1), (5, 4), (5, 6), (5, 8), (5, 9), (5, 10), (5, 12), (5, 14), (5, 15), (5, 16), (5, 18), (6, 0), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6), (6, 7), (6, 8), (6, 9), (6, 10), (6, 11), (6, 12), (6, 13), (6, 14), (6, 15), (6, 16), (6, 17), (6, 18), (6, 19), (7, 0), (7, 1), (7, 4), (7, 6), (7, 8), (7, 9), (7, 10), (7, 12), (7, 14), (7, 15), (7, 16), (7, 18), (8, 0), (8, 1), (8, 2), (8, 3), (8, 4), (8, 5), (8, 6), (8, 7), (8, 8), (8, 9), (8, 10), (8, 11), (8, 12), (8, 13), (8, 14), (8, 15), (8, 16), (8, 17), (8, 18), (8, 19), (10, 0), (10, 1), (10, 2), (10, 3), (10, 4), (10, 5), (10, 6), (10, 7), (10, 8), (10, 9), (10, 10), (10, 11), (10, 12), (10, 13), (10, 14), (10, 15), (10, 16), (10, 17), (10, 18), (10, 19), (11, 0), (11, 1), (11, 4), (11, 6), (11, 8), (11, 9), (11, 10), (11, 12), (11, 14), (11, 15), (11, 16), (11, 18), (12, 0), (12, 1), (12, 2), (12, 3), (12, 4), (12, 5), (12, 6), (12, 7), (12, 8), (12, 9), (12, 10), (12, 11), (12, 12), (12, 13),

```
(12, 14), (12, 15), (12, 16), (12, 17), (12, 18), (12, 19), (13, 0),
(13, 1), (13, 4), (13, 6), (13, 8), (13, 9), (13, 10), (13, 12), (13,
14), (13, 15), (13, 16), (13, 18), (14, 0), (14, 1), (14, 2), (14, 3),
(14, 4), (14, 5), (14, 6), (14, 7), (14, 8), (14, 9), (14, 10), (14,
11), (14, 12), (14, 13), (14, 14), (14, 15), (14, 16), (14, 17), (14,
18), (14, 19), (16, 0), (16, 1), (16, 2), (16, 3), (16, 4), (16, 5),
(16, 6), (16, 7), (16, 8), (16, 9), (16, 10), (16, 11), (16, 12), (16,
13), (16, 14), (16, 15), (16, 16), (16, 17), (16, 18), (16, 19), (17,
0), (17, 1), (17, 4), (17, 6), (17, 8), (17, 9), (17, 10), (17, 12),
(17, 14), (17, 15), (17, 16), (17, 18), (18, 0), (18, 1), (18, 2),
(18, 3), (18, 4), (18, 5), (18, 6), (18, 7), (18, 8), (18, 9), (18,
10), (18, 11), (18, 12), (18, 13), (18, 14), (18, 15), (18, 16), (18,
17), (18, 18), (18, 19), (19, 0), (19, 1), (19, 4), (19, 6), (19, 8),
(19, 9), (19, 10), (19, 12), (19, 14), (19, 15), (19, 16), (19, 18])
284
>>> len([(x,y) for x in range(20) for y in range(20)
           if ((x%2 == 0) or (isPrime(x) and isComposite(y)))])

400
```

## 6.4 More Examples

1. List some of the elements and non-elements of this set:

$$\{(x,y) \mid (x > 3 \wedge y > 2) + (y > 3 \Rightarrow x = 5)\}$$

Answer:  $\{(4,3),(5,4),\dots\}$

Some pairs not in the set are  $\{(1,100),(2,100),(1,1000),\dots\}$ . These are found out through inspection, and checked in the Python session below.

```
>>> {(x,y) for x in range(10) for y in range(10) if ((x>3) and (y>2)) or (not(y>3) or (x==5)) }
set([
(5, 9), (4, 7), (1, 3), (9, 1), (4, 8), (3, 0), (9, 8), (8, 0), (5, 4), (2, 1), (8, 9), (6, 2),
(9, 3), (9, 4), (5, 1), (0, 3), (8, 5), (5, 8), (4, 0), (1, 2), (9, 0), (9, 5), (4, 9), (3, 3),
(8, 1), (7, 6), (4, 4), (6, 3), (5, 6), (7, 2), (5, 0), (2, 2), (7, 7), (5, 7), (7, 3), (4, 1),
(1, 1), (9, 7), (6, 4), (3, 2), (0, 0), (6, 6), (8, 2), (7, 1), (4, 5), (7, 9), (8, 6), (5, 5),
(6, 0), (7, 5), (2, 3), (8, 7), (6, 8), (4, 2), (1, 0), (9, 6), (6, 5), (5, 3), (0, 1), (6, 9),
(7, 0), (4, 6), (6, 7), (9, 2), (7, 8), (6, 1), (3, 1), (9, 9), (7, 4), (2, 0), (8, 8), (4, 3),
(8, 3), (5, 2), (0, 2), (8, 4)])
>>> (1,100) in {(x,y) for x in range(2) for y in range(100) if
                  ((x>3) and (y>2)) or (not(y>3) or (x==5)) }
False
```

```
>>> (2,100) in {(x,y) for x in range(2) for y in range(100)
                  if ((x>3) and (y>2)) or (not(y>3) or (x==5)) }
False
```

2. List some of the elements and non-elements of this set:

$$\{(x,y) \mid (x > 3 \Rightarrow y > 2) + (y > 3 \oplus x = 5)\}$$

Answer using Python, where  $\oplus$  is implemented by the infix operator  $\wedge$ :

```
>>> {(x,y) for x in range(10) for y in range(10) if (not(x>3) or (y>2)) or ((y > 3) ^ (x==5)) }
set([
(9, 3), (5, 9), (4, 7), (1, 3), (4, 8), (3, 0), (2, 8), (9, 8), (7, 8), (5, 4), (2, 1),
(8, 9), (5, 6), (2, 6), (1, 6), (9, 4), (5, 1), (3, 7), (0, 3), (8, 5), (2, 5), (5, 8),
(1, 2), (7, 4), (9, 5), (4, 9), (3, 3), (2, 9), (2, 0), (7, 6), (4, 4), (6, 3), (1, 5),
(8, 8), (8, 3), (3, 6), (0, 4), (8, 6), (5, 7), (5, 3), (1, 1), (9, 7), (2, 7), (3, 2),
(0, 0), (6, 6), (5, 0), (4, 5), (7, 9), (2, 2), (5, 5), (1, 4), (7, 7), (3, 9), (0, 5),
(0, 7), (8, 7), (6, 8), (6, 4), (6, 7), (1, 0), (0, 8), (9, 6), (6, 5), (3, 5), (0, 1),
(6, 9), (7, 3), (4, 6), (5, 2), (3, 1), (9, 9), (2, 4), (3, 8), (0, 6), (1, 8), (7, 5),
(4, 3), (1, 7), (0, 9), (2, 3), (8, 4), (3, 4), (0, 2), (1, 9)])
```

How to compute what is *not* in the set. First compute “U” which is like a “mini universal set.” Then use set subtraction.

```
>>> U = {(x,y) for x in range(10) for y in range(10) }

>>> S = {(x,y) for x in range(10)
            for y in range(10) if (not(x>3) or (y>2)) or ((y > 3) ^ (x==5)) }

>>> U - S
set([(9, 0), (7, 0), (8, 2), (7, 1), (9, 2), (6, 1), (8, 0), (6, 0), (8, 1), (6, 2),
      (9, 1), (4, 2), (4, 1), (7, 2), (4, 0)])

>>> U
set([
(7, 3), (6, 9), (0, 7), (1, 6), (3, 7), (2, 5), (8, 5), (5, 8), (4, 0), (9, 0), (6, 7),
(5, 5), (7, 6), (0, 4), (1, 1), (3, 2), (2, 6), (8, 2), (4, 5), (9, 3), (6, 0), (7, 5),
(0, 1), (3, 1), (9, 9), (7, 8), (2, 1), (8, 9), (9, 4), (5, 1), (7, 2), (1, 5), (3, 6),
(2, 2), (8, 6), (4, 1), (9, 7), (6, 4), (5, 4), (7, 1), (0, 5), (1, 0), (0, 8), (3, 5),
(2, 7), (8, 3), (4, 6), (9, 2), (6, 1), (5, 7), (7, 4), (0, 2), (1, 3), (4, 8), (3, 0),
(2, 8), (9, 8), (8, 0), (6, 2), (5, 0), (1, 4), (3, 9), (2, 3), (1, 9), (8, 7), (4, 2),
(9, 6), (6, 5), (5, 3), (7, 0), (6, 8), (0, 6), (1, 7), (0, 9), (3, 4), (2, 4), (8, 4),
(5, 9), (4, 7), (9, 1), (6, 6), (5, 6), (7, 7), (0, 3), (1, 2), (4, 9), (3, 3), (2, 9),
(8, 1), (4, 4), (6, 3), (0, 0), (7, 9), (3, 8), (2, 0), (1, 8), (8, 8), (4, 3), (9, 5),
(5, 2)])
```

>>> len(U)  
100

>>>

## 6.5 A Boolean, Set, and Lambda Wringer

This series of exercises illustrates how powerful the Lambda notation is, and how conveniently we can employ it to enumerate as well as count the number of Boolean functions over N inputs. Set comprehensions and cartesian product are also used in context. The reader is urged to type these definitions and exercise the various functions defined.

```
# First, let us define a 2-to-1 mux
#
def mux21a(s, i1, i0):
    """When called with s true, mux21a returns i1; else
       it returns i0.
    """
    return {(0,0,0):0,
             (0,0,1):1,
             (0,1,0):0,
             (0,1,1):1,
             (1,0,0):0,
             (1,0,1):0,
             (1,1,0):1,
             (1,1,1):1}[(s,i1,i0)]

def nor2(a, b):
    """ nor2 is realized via its truth-table.
    """
    return {(0,0):1,
             (0,1):0,
             (1,0):0,
             (1,1):0}[(a,b)]

# mux21 via nor2 gates.
# !s.i0 + s.i1 = !((s+i0) . (!s+i1)) = !(s+i0)+!(s+i1)
#
# p = !(s+i0)  q = !(s+i1)
#
def mux21b(s, i1, i0):
    """When called with s true, mux21a returns i1; else
       it returns i0. This is a mux realization using
       nor2 gates.
    """
    ni0 = nor2(i0,i0)
    ns = nor2(s,s)
    ni1 = nor2(i1,i1)
    p = nor2(s,ni0)
    q = nor2(ns, ni1)
    nrslt = nor2(p,q)
    return nor2(nrslt, nrslt)

def cart2(A, B):
    """Returns the cartesian product of sets A and B.
    """
    return {(a,b) for a in A for b in B}
```

```

def cart3(A, B, C):
    """Returns the cartesian product of sets A, B, and C.
    """
    return {(a,b,c) for a in A for b in B for c in C}

def test_mux21a_eq_mux21b():
    """Testing mux21a and mux21b being equal. Illustrates
       the use of cartesian product and higher-order functions.
    """
    testTriples = cart3(set({0,1}), set({0,1}), set({0,1}))
    testResults = map(lambda (s, i1, i0): mux21a(s, i1, i0) == mux21b(s, i1, i0), testTriples)
    bigAnd = reduce(lambda a, b: a and b, testResults)
    if bigAnd:
        print("Testing succeeded. mux21a and mux21b are the same function.")
    else:
        print("Testing FAILED. mux21a and mux21b are NOT the same function.")

def mux41(i0,i1,i2,i3):
    """A direct realization of mux41 - a 4-to-1 mux.
    """
    return lambda s1,s0:{(0,0):i0,(0,1):i1,(1,0):i2,(1,1):i3}[(s1,s0)]

def xor2(a,b):
    """An xor2 can be stamped out of a mux41 by invoking it with the
       desired truth-table output of XOR.
    """
    return mux41(0,1,1,0)(a,b)

def nand2(a,b):
    """A nand2 can be stamped out of a mux41 by invoking it with the
       desired truth-table output of XOR.
    """
    return mux41(1,1,1,0)(a,b)

def xor_via_nand(a,b):
    """This is how to realize XOR using four nands.
    """
    p = nand2(a,b)
    q = nand2(a,p)
    r = nand2(b,p)
    return nand2(q,r)

def test_xor2_eq_xor_via_nand():
    """We can now compare the XORS xor2 and xor_via_nand.
    """
    testPairs = cart2(set({0,1}), set({0,1}))
    print "testPairs =", testPairs
    testResults = map(lambda (a,b): xor2(a,b) == xor_via_nand(a,b), testPairs)
    bigAnd = reduce(lambda a, b: a and b, testResults)
    if bigAnd:
        print("Testing succeeded. xor2 and xor_via_nand are the same function.")
    else:
        print("Testing FAILED. xor2 and xor_via_nand are NOT the same function.")

# See how Python enumerates the coordinates
[(a,b,c,d) for a in {0,1} for b in {0,1} for c in {0,1} for d in {0,1} ]

```

```

def all_2input_fns():
    """We can create an enumeration of all 16 Boolean functions over 4 inputs.
    """
    return [ mux41(a,b,c,d) for a in {0,1} for b in {0,1} for c in {0,1} for d in {0,1} ]

all2inFns = all_2input_fns()

len(all2inFns)

# Nand2 happens to be the 15th function in the enumeration
#
nand2alt = all2inFns[14] # "14" obtained by how the (a,b,c,d) pairs gen.

```

## 6.6 A Useful Set Identity

### 6.6.1 Set Identity $A \cup (B - C)$

Given three sets  $A$ ,  $B$ , and  $C$ , where  $A$  and  $C$  are disjoint (meaning, no overlap), write a formal proof to show that  $A \cup (B - C) = (A \cup B) - C$ . Incorporate the disjointness assumption in your proof (for example, at some proof step, you may have to simplify a certain sub-expression using the identity  $x \in A \Rightarrow x \notin C$ ).

### 6.6.2 Checking the proof in §6.6.1

Check the proof you wrote in Question 6.6.1 for the following sets  $A$ ,  $B$ ,  $C$ , and universal set  $U$ .

- $A$  = The set of prime numbers in `range(3,21)`. You may define this set suitably using the function `primes` given in Section 6.3 on Page 87.
- $B$  = `{ i for i in range(2,17) if (i%2 == 0) }`
- $C$  = `{ i for i in range(2,17) if (i%6 == 0) }`
- $U$  = `range(2,21)`

Type in your proof as shown in §5.2.1 and check your proof.

# Chapter 7

## Systematic Design of Gate Networks

This chapter continues the discussions in earlier chapters and focuses on some new topics that either add details or lend cohesion to earlier discussed topics.

Suppose we have to design a staircase light switch which consists of two switches  $a$  and  $b$  at either end of a long staircase that control a single staircase light (lamp). Initially we have both switches in the “down” (say “off”) position, with the light being off.<sup>1</sup> When a person wishes to walk the stairs, he/she flicks the switch at that end (say  $a$ ), thus turning the light on. Then after passing the stairs, they flick the other switch (say  $b$ ), thus turning the light off. A person may then come from the opposite side, flick  $b$  (light becomes on), and exit at the other end and flick  $a$  (light becomes off). A person may flick  $a$  (light on), see a gigantic spider and run backwards scared, flicking  $a$  again (and turning the light off in the process). It is easy to see that the light would be on exactly when  $a \oplus b$ .

Suppose we have a master override  $m$  added. The desired logic is this: (1) if  $m$ , then the light is on (regardless of the state of  $a$  and  $b$ ); (2) if  $\neg m$ , the original logic takes effect. The resulting logic is captured through a truth-table in Figure 7.1. Notice that ‘-’ in the truth-table is a “don’t care,” standing for a 0 or a 1. Don’t cares are labor-saving devices used to populate truth-tables (else one is forced to spell out all the cases).

---

<sup>1</sup>Note: A conventional staircase light switch is not an on/off switch; it is a single-pole double throw, or SPDT. We won’t worry about this detail.

m	a	b	out
1	-	-	1
0	0	1	1
0	1	0	1
0	0	0	0
0	1	1	0

Figure 7.1: A Staircase Switch with Master Override

Let us try to obtain an equation that governs the truth of  $out$  (when  $out = 1$ ) by examining the truth-table entries:

- $out = 1$  when  $m = 1$ ; OR
- $out = 1$  when  $m = 0$  and  $a = 0, b = 1$ , or  $a = 1, b = 0$ .
- Thus, we can write the governing equation for  $out$  as

$$out = m \cdot !a \cdot !b + m \cdot !a \cdot b + m \cdot a \cdot !b + m \cdot a \cdot b + !m \cdot a \cdot !b + !m \cdot !a \cdot b$$

This is obtained by reading every row where  $out = 1$ .

- It is possible to synthesize a circuit directly out of this equation. However, we have the possibility of simplifying this equation (thus also simplifying the resulting circuit). This is what we proceed to do now.
- Taking  $m$  as a common factor, we can write

$$out = m \cdot (!a \cdot !b + !a \cdot b + a \cdot !b + a \cdot b) + !m \cdot a \cdot !b + !m \cdot !a \cdot b$$

- It can easily be verified that the following identity holds:  

$$(!a \cdot !b + !a \cdot b + a \cdot !b + a \cdot b) = true$$
- Using this identity we can obtain  $out = m + !m \cdot a \cdot !b + !m \cdot !a \cdot b$ .
- Now, using the propositional identity  $a + !a \cdot b = a + b$ , we finally obtain

$$out = m + (a \cdot !b + !a \cdot b).$$

This identity immediately makes sense: either  $m = 1$  in which case,  $out$  does not depend on  $a$  and  $b$ ; or  $m = 0$ , in which case  $out = a \oplus b$ .

- It is also possible to write this equation as

$out = m + (a.\bar{b} + \bar{a}.b)$ , which helps you get used to another popular way of writing negation.

## 7.1 The Disjunctive Normal Form

First, let us define a new term called a *literal*. A literal is either a variable or its negation. For example, in the equation

$$out = m + (a.\bar{b} + \bar{a}.b),$$

there are these literals:  $out$ ,  $m$ ,  $a$ ,  $\bar{a}$ ,  $b$ , and  $\bar{b}$ . Literals are “almost a raw variable,” but for a negation. Since it is “so easy” to negate a variable (we just need an inverter)—and in many circuit forms, the negated form is always readily available—we can view circuit synthesis in terms of literals, and at the last moment use a few inverters (if at all needed) to obtain negated literals. That is the overall plan; now let us return to synthesizing  $out$  using gates.

The equation  $out = m + (a.\bar{b} + \bar{a}.b)$  is in *disjunctive normal form* (DNF) or *sum of products* form. This means that the expression consists of the sum of product terms: in our example, the product terms are  $m$ ,  $a.\bar{b}$ , and  $\bar{a}.b$ . Single literals<sup>2</sup> such as  $m$  are also product terms in a trivial sense (one can always think of  $m$  as being equal to  $m.m$ , for instance). The other two product terms involved are, of course  $a.\bar{b}$  and  $\bar{a}.b$ , and they indeed are non-trivial product terms.

## 7.2 DNF Realizations using And/Or and Nand/Nand

All disjunctive normal form expressions can be realized in two ways:

- Directly in terms of Ands and Ors of literals (and to obtain negated literals such as  $\bar{x}$ , one can use an inverter to invert  $x$ ).
- Directly in terms of Nand gates.

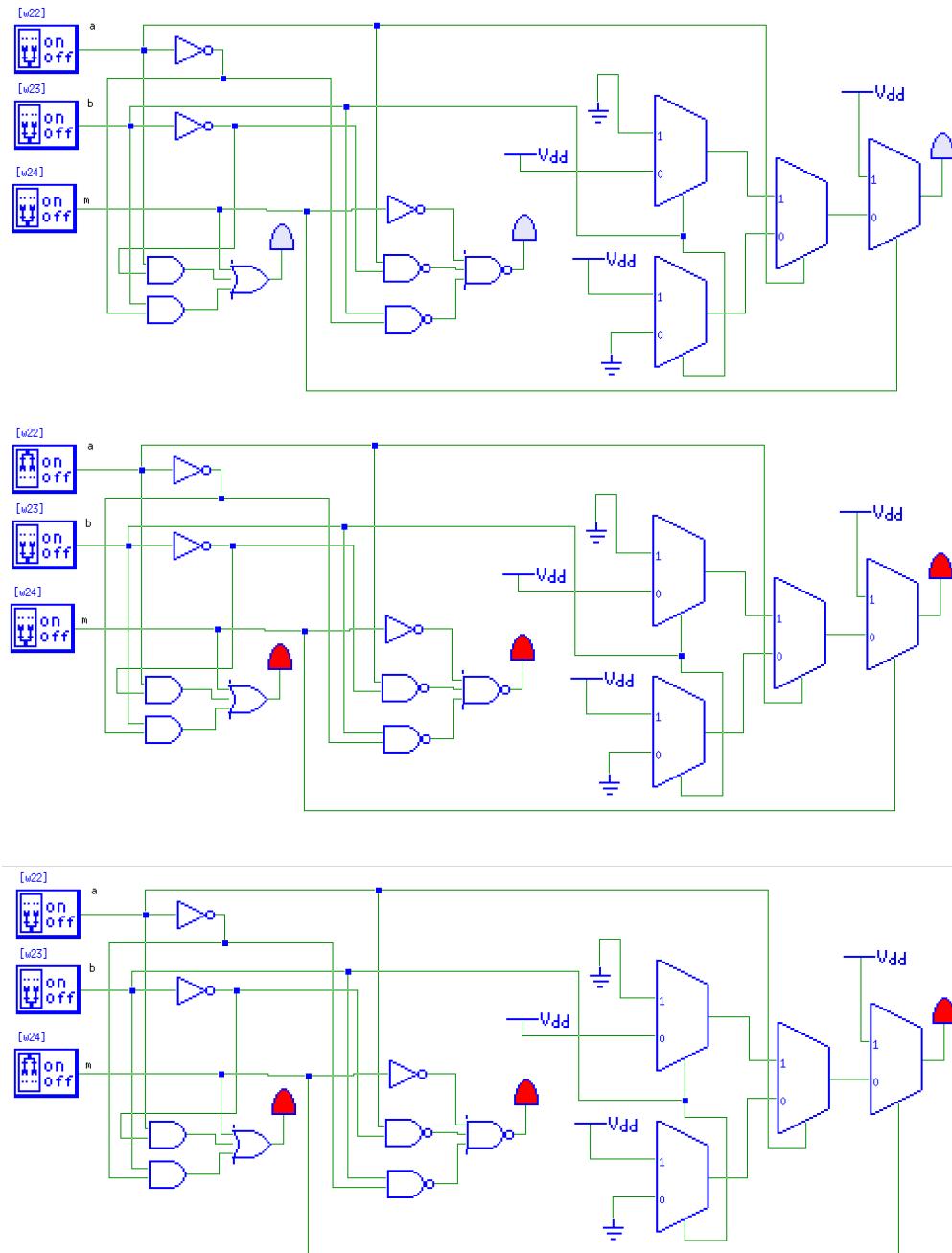


Figure 7.2: The Staircase Switch via DNF (And/Or and Nand/Nand), and ITENF. The cases are: master off, a off, b off (top); master off, a on, b off (middle); and master on (bottom)

### 7.2.1 The And/Or form of the DNF equation $out = m + (a.\!b + \!\!a.b)$ :

Please see Figure 7.2, focusing on the left-most circuits with its own LED driven by a three-input OR gate. The snapshots in the top, middle, and bottom show the response of this LED for various switch settings: This circuit is nothing but a direct rendering of this equation for  $out$ .

### 7.2.2 The Nand/Nand form of the DNF equation $out = m + (a.\!b + \!\!a.b)$ :

Please see Figure 7.2, focusing on the middle circuits with its own LED driven by a three-input NAND gate. The snapshots in the top, middle, and bottom show the response of this LED for various switch settings: This circuit is obtained as follows:

$$\begin{aligned} out &= m + (a.\!b + \!\!a.b) \\ &= \overline{\overline{m + (a.\!b + \!\!a.b)}} \\ &= \overline{\overline{m}} \cdot \overline{\overline{a.\!b}} \cdot \overline{\overline{\!\!a.b}} \end{aligned}$$

- This is the Nand/Nand realization, where  $\overline{m}$  can be viewed as a single-input Nand. For clarity one can also read this equation as

`nand(nand(m), nand(a, !b), nand(!a, b)),`

where `nand(m)` is equivalent to `!m`.

### 7.2.3 “Bubble Pushing” for Circuit Inter-conversion

One can also derive a Nand/Nand circuit from an And/Or circuit using “bubble-pushing” rules presented in Figure 7.3. This figure presents two identities based on De Morgan’s laws:

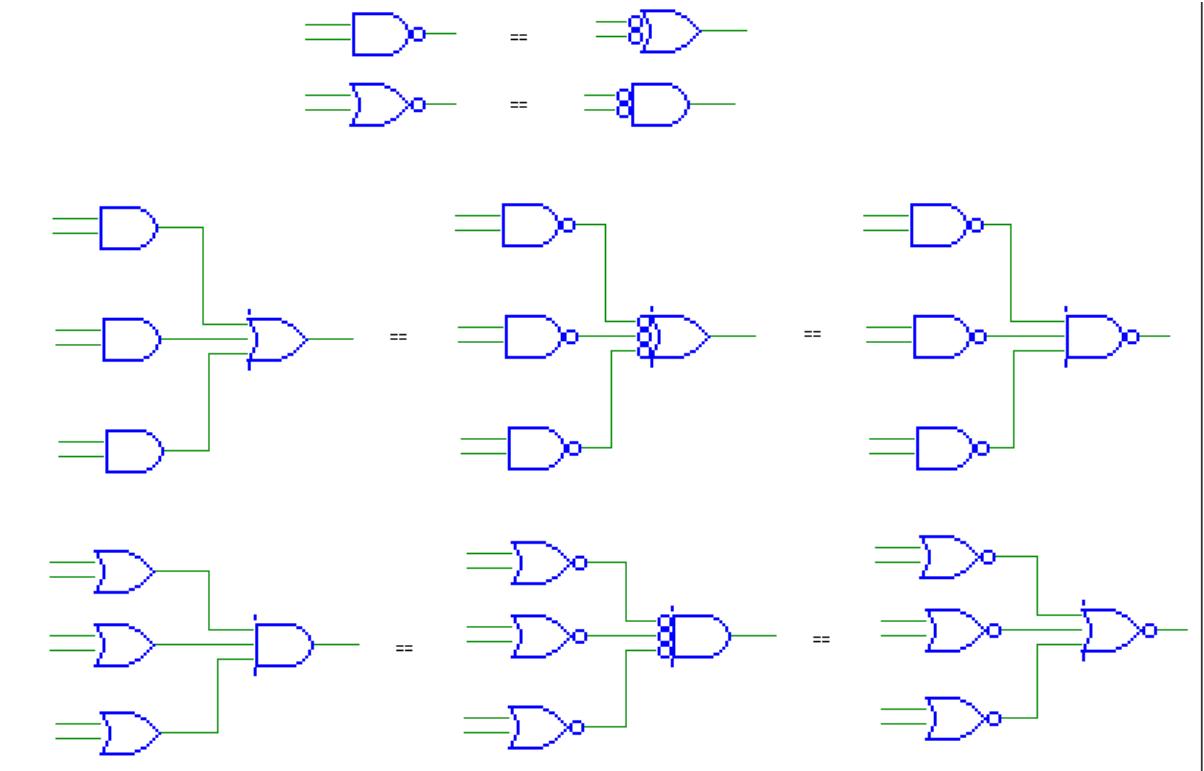


Figure 7.3: De Morgan's Laws Illustrated using “Bubble Pushing”

$$\begin{aligned}\overline{a \cdot b} &= \overline{a} + \overline{b} \\ a + b &= \overline{\overline{a} \cdot \overline{b}}\end{aligned}$$

Using these rules repeatedly, we can convert an And/Or circuit to a Nand/Nand Circuit. Similarly we can convert an Or/And circuit into a Nor/Nor circuit. Both conversions rely upon introducing a “bubble” at both ends of the middle wire, and then moving one set of bubbles to the output side of the final gate generating the output. In the rest of this section, we will be deriving circuits from first principles; once you understand these conversions well, you may use bubble-pushing safely.

### 7.3 The If-Then-Else Normal Form

The ITENF or If-Then-Else normal form comes under various names—for instance also the *Shannon Expansion*. Here is the idea. Suppose  $E$  is a

Boolean expression containing  $x$ . Let us use  $E \downarrow_{x=1}$  to denote  $E$  with every occurrence of  $x$  replaced with 1 (and likewise use  $E \downarrow_{x=0}$  to denote  $E$  with every occurrence of  $x$  replaced with 0). For example,

$$\begin{aligned} ab \downarrow_{a=1} \\ = 1.b \\ = b. \end{aligned}$$

Another example:

$$\begin{aligned} (a + b) \downarrow_{a=0} \\ = (0 + b) \\ = b. \end{aligned}$$

If the variable being substituted does not appear in the expression, then there is nothing to do; example:

$$\begin{aligned} (a + b) \downarrow_{c=0} \\ = (a + b). \end{aligned}$$

Another example:

$$\begin{aligned} (a + b) \downarrow_{d=1} \\ = (a + b). \end{aligned}$$

Given all this, the **Shannon Expansion** (or **ITENF**) of a given expression  $E$  is given by this equation:

$$E = x.E \downarrow_{x=1} + !x.E \downarrow_{x=0}$$

**Example:**  $E = ab$ .

$$\begin{aligned} E &= a.(ab) \downarrow_{a=1} + !a.(ab) \downarrow_{a=0} \\ &= a.(1.b) \downarrow_{a=1} + !a.(0.b) \downarrow_{a=0} \\ &= a.(1.b) \downarrow_{a=1} \\ &= ab \end{aligned}$$

Well, nothing significant was achieved by obtaining the ITENF for  $ab$ . However, in §7.4, we will use the ITENF to obtain a mux21 based circuit for the stair-case switch design. Also, in later chapters, we will relate the ITENF to two powerful ideas:

- Proof by cases, and
- Binary Decision Diagrams

## 7.4 The ITENF form of $out = m + (a \cdot !b + !a \cdot b)$

We now detail the steps involved in applying the ITENF rendering of the equation for  $out$ .

- Let us start with the value of  $out$ , which is given by:  

$$m + (a.!b + !a.b)$$
- Let us expand on  $m$ . This choice is by no means unique. In later chapters, we will learn that the order in which we select the variables around which the ITENF is constructed can make a significant difference. In our problem, we will choose  $m$  first, then  $a$ , and finally  $b$ . But we could easily have considered  $a$  first,  $b$  next, and finally  $m$  last. The order does affect the final circuit (how big or small it ends up being). However, all circuits obtained will be *correct!*
- Thus we proceed to do the Shannon expansion with respect to  $m$ .  

$$= m \cdot (m + (a.!b + !a.b)) \downarrow_{m=1} + !m \cdot (m + (a.!b + !a.b)) \downarrow_{m=0}$$
- Simplifying, we get  

$$= m \cdot (1 + (a.!b + !a.b)) + !m \cdot (0 + (a.!b + !a.b))$$
- which becomes  

$$= m \cdot (1) + !m \cdot ((a.!b + !a.b))$$
- The astute reader would have realized that we can indeed realize the above equation using a mux21! This mux21 would be fed 1 on the “1 input” and  $(a.!b + !a.b)$  on the “0 input,” with  $m$  acting as the selector for this multiplexor. Notice that these expressions are “ $m$ -free” (don’t contain  $m$ ) because we have essentially done a “case analysis” on  $m$  (we considered  $m$  in turn being a 0 or a 1).
- Now, let us expand the “ $m$ -free parts” with respect to  $a$ . We won’t bother expanding 1 with respect to  $a$ , as it will remain the same. So we focus on  $(a.!b + !a.b)$ .

$$\begin{aligned} & (a.!b + !a.b) \\ &= a \cdot (a.!b + !a.b) \downarrow_{a=1} + !a \cdot (a.!b + !a.b) \downarrow_{a=0} \\ &= a \cdot (1.!b + !1.b) + !a \cdot (0.!b + !0.b) \\ &= a \cdot (!b) + !a \cdot (b) \end{aligned}$$

- Now again, we get another mux21 with its inputs being  $b$  (going into the “0 input”) and  $!b$  (going into the “1” input).
- We could stop here, or simply do one more level of expansion of  $b$  and  $!b$  with respect to  $b$ . This yields two more mux21s.

- The resulting circuit is shown in the right-most column of the top, middle, and bottom of Figure 7.2.
- In case the reader is left wondering, the entire ITENF expansion for *out* can now be written as follows. However, instead of confusing you with a longer formula, let us resort to the “if-then-else” reading of mux21 written in pseudo-Python syntax. Using that reading, we can write

```

out = if m:
    return 1
else:
    if a:
        if b:
            return 0
        else:
            return 1
    else:
        if b:
            return 1
        else:
            return 0

```

## 7.5 The Conjunctive Normal Form

Figure 7.4 shows how we can obtain the governing equation for *out* from another perspective.

- Examine the truth-table in Figure 7.1 once again. But now focus on  $out = 0$ .
- We find that  $out = 0$  exactly when  $(!m \cdot !a \cdot !b + !m \cdot a \cdot b)$
- Thus,  $out = 1$  when the *negation* of  $(!m \cdot !a \cdot !b + !m \cdot a \cdot b)$  is true.
- Negating  $(!m \cdot !a \cdot !b + !m \cdot a \cdot b)$ , we obtain, using De Morgan’s law  $(m + a + b) \cdot (m + !a + !b)$ . This corresponds to the circuit in the left column of Figure 7.4, which directly realizes this Or/And equation.
- The form  $(m + a + b) \cdot (m + !a + !b)$  is called the *conjunctive normal form*

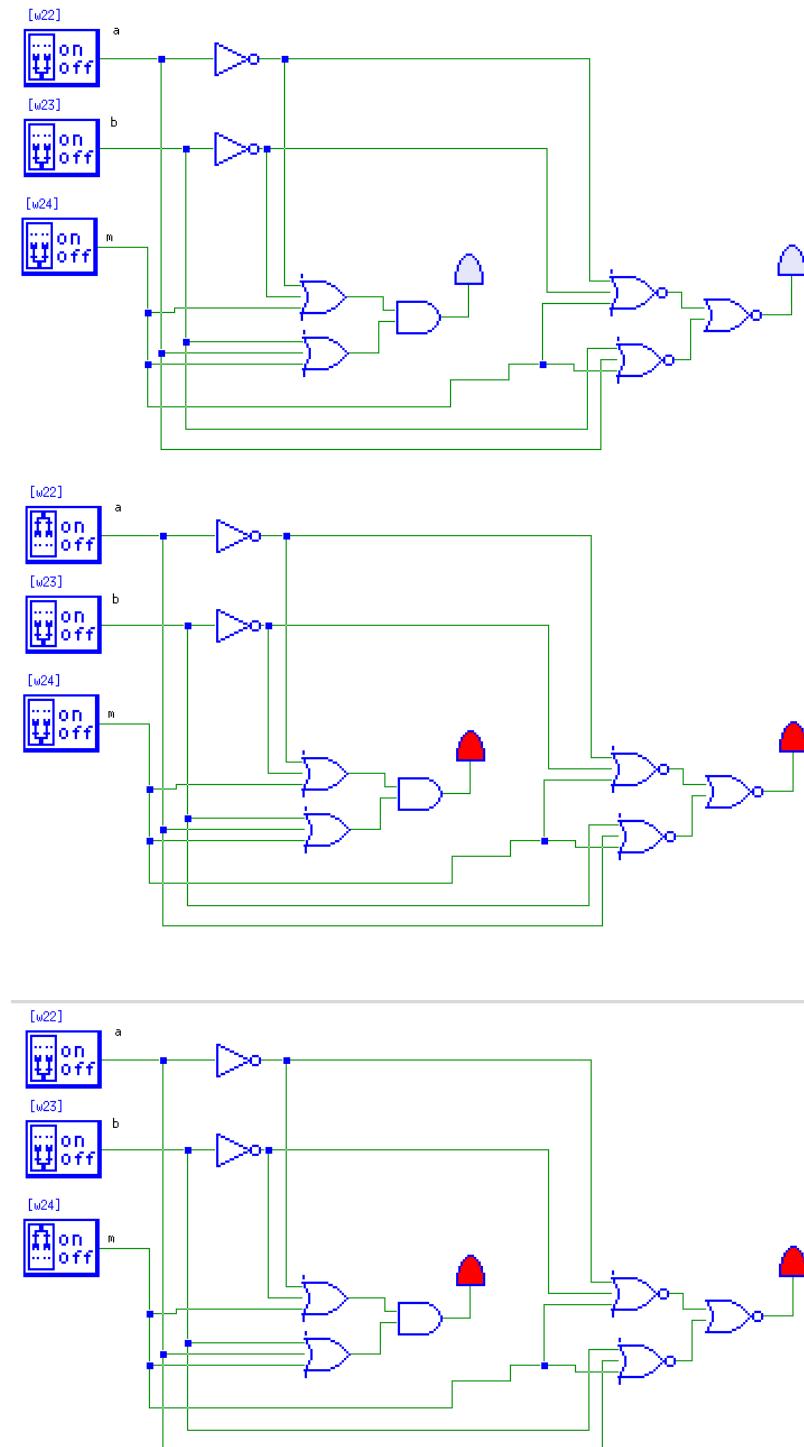


Figure 7.4: The Staircase Switch via CNF (Or/And and Nor/Nor). The cases are: master off, a off, b off (top); master off, a on, b off (middle); and master on (bottom)

(CNF) because it is a conjunct (product) of disjuncts (sums).

- Any expression in CNF can be directly realized using Nor gates. To see this, let us take double-negation as before.

$$\begin{aligned} & \overline{(m + a + b) \cdot (m + !a + !b)} \\ &= (\overline{m + a + b}) + (\overline{m + !a + !b}) \end{aligned}$$

- This is nothing but a Nor/Nor circuit. For clarity one can also read this equation as

`nor(nor(m, a, b), nor(m, !a, !b)).`

**Bubble Pushing:** It is clear that Bubble Pushing can be applied to the Or/And circuit of Figure 7.4, yielding the Nor/Nor circuit, as well. In particular, the Or/And to the left-hand side of Figure 7.4 was rendered into a Nor/Nor in the right-hand side of the same figure.



# Chapter 8

## Propositional Proofs

In this chapter, we will be going through three approaches to writing simple propositional proofs. All the Boolean algebra and propositional logic you learned so far will be put to use in carrying out these proofs.

We will first describe the notion of “proving something” in the context of propositional logic (§8.1). Then we will show you how to prove assertions already stated in logic (§8.2). Next, we will show you how to capture English sentences in propositional logic (§8.3). We discuss proof by contradiction—**a fundamentally important proof technique**—in §8.4. We will close the section by introducing you to Z3py, a proof assistant (§8.5).

### 8.1 The notion of Proving

**Example 1:**  $x \vee \neg x$  Suppose you are asked to prove  $x \vee \neg x$ . In Mathematical logic, proving this assertion simply means *showing it true for all x*. This is easily accomplished by building a truth-table for this entire formula viewed as a Boolean function. Obviously, this formula is true for  $x = 0$  and  $x = 1$ —in other words, it is “always true.”

**Example 2:**  $x \Rightarrow (y \Rightarrow x)$  Again, by going through the truth table, you will find that this formula is always true (true for all  $x$  and  $y$ ). One can also see it more clearly by expanding it to simpler Boolean propositions:

- $x \Rightarrow (y \Rightarrow x) = \neg x \vee (\neg y \vee x)$
- Since  $\vee$  is *commutative* (“order of application does not matter”, i.e., “ $a \vee b = b \vee a$ ”), we can write it as  $(\neg x \vee (x \vee \neg y))$

- Since  $\vee$  is *associative* (“bracketing order does not matter”), we can write it as  $((\neg x \vee x) \vee \neg y)$
- Since  $(\neg x \vee x)$  is true, we can simplify the above equation to  $True \vee \neg y$ )
- Since  $True \vee$  anything is  $True$ , the above reduces to  $True$ .

**Example 3:**  $x \vee y$  Clearly, this assertion cannot be “proven” because one can find a way to make this assertion false ( $x = y = 0$  does it). Now, if we substitute real-world facts in place of  $x$  and  $y$ , we will know that all this makes sense; e.g., let

- $x$  = “I took an Xray”
- $y$  = “I Yelled all along”

then,  $x \vee y$  need not always be true (I may or may not have told the truth when I made assertions  $x$  and  $y$ ; hence one cannot *prove* this as a fact regardless of whether I took an X-ray and-or whether I yelled all along).

**Example 4: Back to  $x \Rightarrow (y \Rightarrow x)$**  Now if we plug these real-world assertions into  $x \Rightarrow (y \Rightarrow x)$ , I get

- *I took an X-ray Implies (I yelled all along Implies I took an Xray).*
- Well, I may not have taken an X-ray. Then the whole assertion is vacuously true.
- Or I may have taken an X-ray; in that case, whether I yelled all along or not, it is true that  
*(I yelled all along Implies I took an Xray).*

**Example 5: Expressing  $a = b$  in Logic** How do we treat “=” in Logic? In Boolean logic,  $=$  is *XNOR*, and that is a fine way to proceed. But here is another insight that we will soon need, when we begin our proofs:

$$a = b \doteq (a \Rightarrow b) \wedge (b \Rightarrow a)$$

Here, read  $\doteq$  as “by definition equal to.” Is this true? You can check it out!

There is another way to write as well as read  $=$ . It is given through *bi-implication*:

$$a = b \doteq (a \Rightarrow b) \wedge (b \Rightarrow a)$$

which is usually written using the single bi-implication symbol  $\Leftrightarrow$  as

$$a = b \doteq (a \Leftrightarrow b).$$

**Example 6: Proving  $ab + bc + c\bar{a} = ab + c\bar{a}$**  Now, we can prove (attempt to prove)  $ab + bc + c\bar{a} = ab + c\bar{a}$  using the bi-implication expansion. We shall prove this in §8.2.3.

## 8.2 Proving Propositional Logic Formulae

When given a propositional formula such as in the examples so far, we use one of the approaches in the following sections, depending on the nature of the formula.

### 8.2.1 Truth-table method

$a$	$b$	$c$	$ab$	$bc$	$c\bar{a}$	$LHS = ab + bc + c\bar{a}$	$RHS = ab + c\bar{a}$
0	0	0	0	0	0	0	0
0	0	1	0	0	1	1	1
0	1	0	0	0	0	0	0
0	1	1	0	1	1	1	1
1	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	1	1	0	1	1

Figure 8.1: Conjecture  $ab + bc + c\bar{a} = ab + c\bar{a}$  shown through a Truth Table

As was already mentioned in the context of Example 1 above, one can build a truth-table in order to prove that the formula is true for all values of its variables (see Figure 10.2).

However, truth-tables are impractical for most purposes, since they are *guaranteed exponential* in size. We will therefore employ propositional identities and various other proof methods to avoid enumerating all cases. Proof assistants such as Z3Py help automate such proofs (see §8.5).

### 8.2.2 Propositional Identities

As shown already in the context of Example 2, we can use propositional identities to show that a formula is true. Basically, reduce the item to be proven to 1 or True.

### 8.2.3 Showing Bi-implication

We will now embark on proving the bi-implication in Example 6. In other words, when the principal operator of an assertion is  $\Leftrightarrow$ , the method described here will help approach the proof.

*Approach: Proof by bi-Implication:*

- Interpret = as  $\Rightarrow$  going both ways.
- Thus, we must have
  - Imp1:*  $ab + c\bar{a} \Rightarrow ab + bc + c\bar{a}$ , and
  - Imp2:*  $ab + bc + c\bar{a} \Rightarrow ab + c\bar{a}$ .
- *Imp1* is obvious because it is of the form

$$ab + c\bar{a} \Rightarrow ab + bc + c\bar{a}$$

which can be rewritten to

$$ab + c\bar{a} \Rightarrow ab + c\bar{a} + bc$$

or of the form

$$P \Rightarrow P + Q$$

which can be shown true through any number of propositional identities (show it).

- *Imp2* is not so obvious. However, if we could show  $bc \Rightarrow ab + c\bar{a}$ , then we will have the following results:

1.  $bc \Rightarrow ab + c\bar{a}$ , and
2.  $ab + c\bar{a} \Rightarrow ab + c\bar{a}$ .

Now, if  $P \Rightarrow Q$  and  $R \Rightarrow Q$ , then  $(P+R) \Rightarrow Q$ . (Try to prove this!) We can prove  $bc \Rightarrow ab + c\bar{a}$  either using truth-tables or through propositional identities. (Show it.)

## 8.3 Modeling and *then* Proving

We now turn our attention to real-world puzzles that can be solved using propositional logic. We consider a puzzle by Lewis Carroll, the famous author of “Alice in Wonderland.” Carroll was also a gifted logician (and actually worked as a professor of mathematics and logic). I am indebted to Prof. Gerald Hiles who has graciously allowed me to copy his website for classroom

use. This website is at: <http://tinyurl.com/Gerald-Hiles-Lewis-Carroll>; please browse it.

Consider the following puzzle from Hiles's compilation:

From the premises

1. Babies are illogical;
2. Nobody is despised who can manage a crocodile;
3. Illogical persons are despised.

Conclude that *Babies cannot manage crocodiles*.

We will use this as a running example in the rest of this subsection.

**Model and Solve the Propositions:** Propositions model real-world concepts that can be *True* or *False*. For example, in the Carroll puzzle, following Hiles's notation [2], we can encode the propositions of interest using single letters:

- $b$  : it is a baby
- $l$  : it is logical
- $m$  : it can manage a crocodile
- $d$  : it is despised

The Carroll puzzle asks a simple question: *If we assume that none of the premises are violated, can we prove the conclusion?*

- The premises guarantee that  $b$ ,  $l$ ,  $m$ , and  $d$  cannot randomly dangle about across the space of values  $\{\text{True}, \text{False}\}$ . For instance, the premis “Babies are illogical” asserts that  $b \Rightarrow \neg l$  is **always** true. This, in turn means that  $b \wedge l$  can never be satisfied. Similar constraints introduced by the other premises.

Under these constraints on the premises, we must conclude that *Babies cannot manage crocodiles*.

**Expressing the Premises:** We express the premises as follows.

- $b \Rightarrow \neg l$  (“Babies are illogical”)
- $m \Rightarrow \neg d$  (“Nobody is despised who can manage a crocodile”)
- $\neg l \Rightarrow d$  (“Illogical persons are despised”)

**Back to Babies and Crocodiles:** Consider the premises we have:

- The three premises of the puzzle:

$$P1: b \Rightarrow \neg l$$

$$P2: m \Rightarrow \neg d$$

$$P3: \neg l \Rightarrow d$$

- We have to now show G:  $b \Rightarrow \neg m$ :

The above discussion explains how you model problems. How do we prove after modeling? The first proof method we shall study is *proof by contradiction*.

## 8.4 Proof by Contradiction

Proof by contradiction is one of the most powerful items in our arsenal. The gist of this proof style is rather simple: If

$$P \Rightarrow False$$

is a theorem, then  $\neg P$ .

In actual use, suppose we have a proof goal  $G$ . Suppose we assert  $\neg G$ . After derivations, including chaining, we can write  $\neg G \Rightarrow false$ . Then, we can use the definition of  $\Rightarrow$  and write  $G$  as a theorem.

**Simple example of Proof by Contradiction:** Just for illustration, suppose we have to prove  $p \vee \neg p$  using proof by contradiction. The idea would be to write it as  $(p \vee \neg p) \Rightarrow False$  and show that this can be simplified to  $False$ . By applying the expansion for  $\Rightarrow$ , we have  $\neg(p \vee \neg p)$  which then amounts to  $(\neg p \wedge p)$  by DeMorgan's law—which is false.

While this was a simple illustration, the actual use of proof by contradiction in practice involves some more rules that we will now introduce.

**Three Crucial Propositional Theorems Pertaining to Implication:** The implication operator is one of the most important of operators when it comes to reasoning. This is hardly surprising because even in programming, virtually nothing useful can be expressed without an if-then-else. There are a few key propositional facts pertaining to implication that we will use repeatedly:

Modus Ponens: How to derive new facts using  $\Rightarrow$ : We have the theorem

$$(A \wedge (A \Rightarrow B)) \Rightarrow B$$

This theorem can be rigorously established as follows:

- Suppose  $A$  is false; then we have

$$\text{False} \Rightarrow B$$

Since  $P \Rightarrow Q$  stands for  $\neg P \vee Q$ , we can rewrite  $\text{False} \Rightarrow B$  as  $\neg \text{False} \vee B$ , which is true.

- Suppose  $A$  is true; then we have

$$(\text{True} \wedge (\neg \text{True} \vee B)) \Rightarrow B$$

which reduces to

$$B \Rightarrow B, \text{ or in fact } \neg B \vee B,$$

which is true.

An example of modus ponens is this:

- Consider the assertion “IF there is smoke, THEN there is fire”, and “There is smoke.”
- Modus ponens allows you to conclude “there is fire.”

Chaining  $\Rightarrow$ : We have the theorem

$$((A \Rightarrow B) \wedge (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)$$

This theorem can be rigorously established as follows:

- Suppose  $A$  is false; then  $(A \Rightarrow C)$  by itself is true.
- Then, the whole claim reduces to

$$((A \Rightarrow B) \wedge (B \Rightarrow C)) \Rightarrow \text{True}.$$

- Now,  $X \Rightarrow \text{True}$  is true, no matter what  $X$  is. Thus the theorem is established.
- Suppose  $A$  is true. Then using Modus Ponens, we can derive  $B$ . This allows us to derive  $C$ . Thus  $A \Rightarrow C$  is true as well. Again, we obtain

$$((A \Rightarrow B) \wedge (B \Rightarrow C)) \Rightarrow \text{True}$$

which is true.

An example of chaining is this:

- Consider the assertion “IF there is smoke, THEN there is fire.”
- Consider the assertion “IF there is fire, THEN the insurance business suffers.”
- We can conclude “IF there is smoke, THEN then insurance business suffers.”

Regardless of what exactly the first two assertions tried to model in English, the conclusion is inevitable.

Contrapositive Rule With Respect to  $\Rightarrow$ : We have the theorem

$$(A \Rightarrow B) \Rightarrow (\neg B \Rightarrow \neg A)$$

This theorem can be rigorously established as follows:

- $A \Rightarrow B$
- $= \neg A \vee B$
- $= (\neg A) \vee \neg(\neg B)$
- $= (\neg B) \Rightarrow (\neg A).$

An example of the contrapositive rule:

- Consider the assertion “IF there is smoke, THEN there is fire.”
- The contrapositive rule says that the above assertion is exactly equivalent to “IF there no fire, THEN there is no smoke.”

### Babies and Crocodiles through Proof by Contradiction:

- The three premises of the puzzle are:

$$P1: b \Rightarrow \neg l$$

$$P2: m \Rightarrow \neg d$$

$$P3: \neg l \Rightarrow d$$

- We have to now show G:  $b \Rightarrow \neg m$ :

• *Suppose not.* That is,  $\neg(b \Rightarrow \neg m)$ .

- From  $\neg(b \Rightarrow \neg m)$ , obtain  $b \wedge \neg m$  (meaning of  $\Rightarrow$ ).
- From  $b \wedge \neg m$  obtain  $b$  and  $\neg m$  (meaning of  $\wedge$ ).
- From  $m$  and P2, obtain  $\neg d$  (modus ponens).
- From P3, obtain P5:  $\neg d \Rightarrow l$  (contrapositive).
- From  $\neg d$  and P5, obtain  $l$  (modus ponens).
- From  $b$  and P1, obtain  $\neg l$  (modus ponens).

- From  $l$  and  $\neg l$ , obtain *False* (conjunction of  $l$  and  $\neg l$ ).
- By chaining, obtain  $\neg G \Rightarrow \text{False}$ .
- Thus, obtain  $G$  (proof by contradiction).

**Babies and Crocodiles through Inference:** While proof by contradiction is a handy approach for many problems, is not essential to apply it to this problem. We could also have proved the assertion  $b \Rightarrow \neg m$  through a series of inference steps. Here is how that proof would go:

- We can combine  $b \Rightarrow \neg l$  and  $\neg l \Rightarrow d$  to obtain  $b \Rightarrow d$ .
- From  $m \Rightarrow \neg d$ , we can obtain the contrapositive form  $d \Rightarrow \neg m$ .
- We can now combine  $b \Rightarrow d$  and  $d \Rightarrow \neg m$  to finally obtain  $b \Rightarrow \neg m$ . This is the conclusion sought.

**A Tip for Manual Proofs:** A useful tip when doing things manually is this: for every implication, write down its contrapositive also at the outset. This way, we will be able to see the chaining opportunities more clearly.

## 8.5 Z3Py: A Proof Assistant

We now present how we can perform mechanized reasoning within propositional logic. A tool called z3Py supports proofs of this nature. We go by example, first examining how we may verify propositional identities using z3Py. Next, we visit a larger problem due to Lewis Carroll, and make headway into solving it using z3Py.

**z3Py for proving propositional theorems:** z3Py provides Python bindings to Microsoft Research's flagship theorem prover  $\bar{Z}3$  (see [rise4fun.com](http://rise4fun.com) for details, including a web-interface to run these tools). The contents of file `propLogic.py` are now listed:

```
--- propLogic : propositional theorems using z3Py ---
from z3 import *

p, q, r = Bools('p q r')
demorgan = And(p, q) == Not(Or(Not(p), Not(q)))
```

```

contrapos= Implies(p, q) == Implies(Not(q), Not(p))
chaining = Implies(And(Implies(p,q), Implies(q,r)), Implies(p,r))
wrong_ch = And(Implies(p,q), Implies(q,r)) == Implies(p,r)
modus_ponens = Implies(And(p, Implies(p,q)), q)

def prove(f):
    s = Solver()
    s.add(Not(f))
    if s.check() == unsat:
        print "proved"
    else:
        print "failed to prove"
        print s.model()

print "Proving demorgan..."
prove(demorgan)

print "Proving contrapositive..."
prove(contrapos)

print "Proving chaining..."
prove(chaining)

print "Proving wrong_ch...won't be able to"
prove(wrong_ch)

print "Proving modus_ponens..."
prove(modus_ponens)

# execfile("propLogic.py")
# Proving demorgan...
# proved
# Proving contrapositive...
# proved
# Proving chaining...
# proved
# Proving wrong_ch...won't be able to
# failed to prove
# [q = False, r = True, p = True]
# Proving modus_ponens...
# proved
#--- end propLogic ---

```

- Fire up Python 2.7 in the class directory /home/student/z3/examples/python/
- Type `execfile("propLogic.py")` against the Python prompt.

- Line `p, q, r = Bools('p q r')` introduces three propositional variables.
  - Lines `demorgan` through `modus_ponens` introduces various propositional theorems. Notice that `wrong_ch` is a *non-theorem*.
  - Notice that function `prove` negates the given formula `f` and tries to check if it is `unsat` (or `False`). If so, by the principle of *proof by contradiction*, we would have proven `f`.
  - When the proof fails, a model is printed out — meaning, `f` is not `False`, but is somehow satisfiable. So `z3Py` tells you in what way it becomes satisfiable. By studying this model, we can often debug the situation.
  - In our example, all are theorems except `wrong_ch`. The model returned is `[q = False, r = True, p = True]`. For this assignment, we find that `And(Implies(p,q), Implies(q,r))` evaluates to `False` while `Implies(p,r)` evaluates to `True`. The correct chaining theorem is of course `Implies(And(Implies(p,q), Implies(q,r)), Implies(p,r))`
- In effect, the `Implies(p,r)` part is *implied*. However, `Implies(p,r)` does not imply `And(Implies(p,q), Implies(q,r))`, and hence they are not equivalent as wrongly claimed by `wrong_ch`.

## 8.6 Lewis Carroll : Young Wise Pigs

Let us solve another puzzle using Z3py. In this puzzle, we have these premises:

1. All who neither dance on tight ropes nor eat penny-buns are old.
2. Pigs, that are liable to giddiness, are treated with respect.
3. No one ought to lunch in public who looks ridiculous and eats penny-buns.
4. Young creatures, who go up in balloons, are liable to giddiness.
5. A wise balloonist takes an umbrella with him.
6. Fat creatures, who look ridiculous, may lunch in public, provided that they do not dance on tight ropes.
7. No wise creatures dance on tight ropes, if liable to giddiness.

8. A pig looks ridiculous carrying an umbrella.
9. All who do not dance on tight ropes and who are treated with respect are fat.

We must show that *no wise young pigs go up in balloons*.

```
from z3 import *

eatPennyBuns, old, young, danceTightRopes, pigs, respect, giddy, \
publicLunch, ridiculous, umbrella, fat, wise, balloon = \
Bools('eatPennyBuns old young danceTightRopes pigs respect \
giddy publicLunch ridiculous umbrella fat wise balloon')

# All who neither dance on tight ropes nor eat penny-buns are old.
P1 = Implies(And(Not(danceTightRopes), Not(eatPennyBuns)), old)

# Pigs, that are liable to giddiness, are treated with respect.
P2 = Implies(And(pigs,giddy), respect)

# No one ought to lunch in public who looks ridiculous and eats penny-buns.
P3 = Implies(And(ridiculous, eatPennyBuns), Not(publicLunch))

# Young creatures, who go up in balloons, are liable to giddiness.
P4 = Implies(And(young, balloon), giddy)

# A wise balloonist takes an umbrella with him.
P5 = Implies(And(wise, balloon), umbrella)

# Fat creatures, who look ridiculous, may lunch in
# public, provided that they do not dance on tight ropes.
P6 = Implies(And(fat, ridiculous, Not(danceTightRopes)), publicLunch)

# No wise creatures dance on tight ropes, if liable to giddiness.
P7 = Implies(And(wise, giddy), Not(danceTightRopes))

# A pig looks ridiculous carrying an umbrella.
P8 = Implies(And(pigs, umbrella), ridiculous)

# All who do not dance on tight ropes and who are treated
# with respect are fat.
P9 = Implies(And(Not(danceTightRopes), respect), fat)

# Frame = True
```

```

# Frame = (old == Not(young))

Conc = Implies(And(wise, young, pigs), Not(balloon))

Goal = Implies(And(P1,P2,P3,P4,P5,P6,P7,P8,P9,Frame), Conc)

def prove(f):
    s = Solver()
    s.add(Not(f))
    if s.check() == unsat:
        print "proved"
    else:
        print "failed to prove"
        print s.model()

print "Goal G..."
prove(Goal)

```

After encoding the puzzle, we found that we could not prove the goal. The model showed why:

```

# execfile("youngWisePigs.py")
# Goal G...
# failed to prove
# [old = True,
#  eatPennyBuns = False,
#  publicLunch = True,
#  fat = True,
#  balloon = True,
#  pigs = True,
#  young = True,
#  wise = True,
#  ridiculous = True,
#  danceTightRopes = False,
#  umbrella = True,
#  giddy = True,
#  respect = True]
# >>>

```

We find that both `old` and `young` are `True`! We add one more constraint (called the *Frame* constraint), resulting in a successful proof:

```

# Now add Frame = (old == Not(young))
#
# Then it proves it!

```

```

#
# execfile("youngWisePigs.py")
# Goal G...
# proved

```

## 8.7 A Circuit Interpretation of Babies and Crocodiles

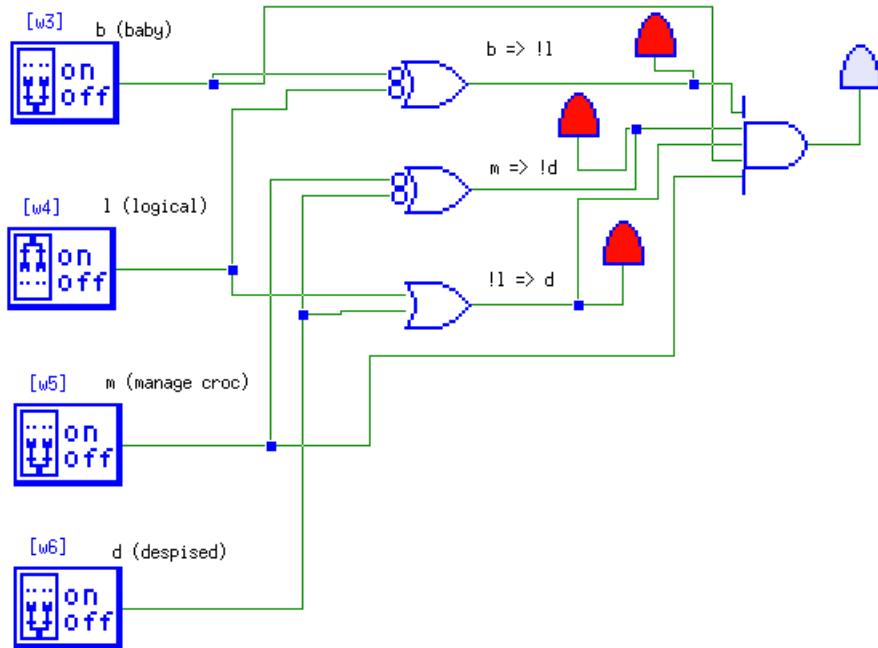


Figure 8.2: Babies and Crocodiles: Proof by Contradiction Shown via a Circuit. The final And gate's output is always off, no matter what switch setting, thus showing that a contradiction has been achieved!

Just to show that all these derivations are grounded in reality, we now present a circuit interpretation of proof by contradiction. The idea is simple:

- Build a circuit consisting of P1, P2, and P3 feeding into an And gate. Here, P1 =  $b \Rightarrow \neg l$ , P2 =  $m \Rightarrow \neg d$ , and P3 =  $\neg l \Rightarrow d$ .
- Feed the negated goal G into the same And gate. Since G =  $b \Rightarrow \neg m$ , the negated G is  $b \wedge m$ .

- Now, the output of this And gate is always “Off” (for all settings of the inputs).
- This demonstrates that the goal  $G$  is implied by the premises. See Figure 8.2 for details.

In general,

- If you have a collection of circuits  $C_1, \dots, C_n$  (modeling real-world assertions, say) over inputs  $I_1, \dots, I_m$  feeding into an And gate such that the gate output is always 0, then  $(C_1 \wedge \dots \wedge C_i) \Rightarrow (C_{i+1} \vee \dots \vee C_n)$  would be a theorem (or tautology) or “always true” assertion.
- In particular,  $(C_1 \wedge \dots \wedge C_{n-1}) \Rightarrow C_n$ .
- Since  $\wedge$  and  $\vee$  are commutative and associative, you can treat the circuit assertions in any order

## 8.8 Inference Rule Summary; More Examples

We shall emphasize proof by contradiction, given that it is highly goal-directed: *as soon as False has been inferred, one can stop the proof.*

### 8.8.1 Inference Rules

These are a collection of inference rules that we have used in our proofs. More rules can be introduced as needed. Recall that *you can make inferences using any tautology of Boolean algebra*.

$$\frac{P \wedge \neg P}{\text{False}} \text{ Contradiction}$$

This is how proofs terminate: as soon as you’ve proved some  $P$  and its negation  $\neg P$ , where  $P$  is any formula.

$$\frac{A \quad (A \Rightarrow B)}{B} \text{ Modus Ponens}$$

This is how from an assertion  $A$  and an implication  $A \Rightarrow B$  you make progress by deducing  $B$ .

$$\frac{A \Rightarrow B \quad B \Rightarrow C}{A \Rightarrow C} \text{ Chaining}$$

Chaining allows you to “transitively collapse” implications, obtaining “long reach” inference steps.

$$\frac{A \Rightarrow B}{\neg B \Rightarrow \neg A} \text{ Contrapositive}$$

Contrapositive allows you to “swing an implication the other way” making it amenable to more chaining steps. *Don’t forget to negate when you swing implications around!*

$$\frac{A \wedge B \wedge C \Rightarrow D}{\neg D \Rightarrow \neg A \vee \neg B \vee \neg C} \text{ Contrapositive Detail 1}$$

Contrapositive, in case you have a “stack” to the left.

$$\frac{(A \wedge B \wedge C) \Rightarrow (D \vee E \vee F)}{(\neg D \wedge \neg E \wedge \neg F) \Rightarrow (\neg A \vee \neg B \vee \neg C)} \text{ Contrapositive Detail 2}$$

Generalized contrapositive.

$$\frac{A \wedge B}{B \wedge A} \text{ And Commutativity}$$

This commutativity rule avoids having to state two And rules below; but good to have the separate rules anyhow.

$$\frac{A \wedge B}{A} \text{ And Rule 1}$$

You can’t have proven  $A \wedge B$  unless you have proven  $A$ .

$$\frac{A \wedge B}{B} \text{ And Rule 2}$$

You can’t have proven  $A \wedge B$  unless you have proven  $B$ .

$$\frac{A \Leftrightarrow B}{B \Leftrightarrow A} \text{ If and Only If}$$

This commutativity rule avoids having to state two  $\Leftrightarrow$  rules below; but good to have the separate rules anyhow.

$$\frac{A \Leftrightarrow B}{A \Rightarrow B} \text{ If and Only If 1}$$

“A If and only If B means” “If A then B” or “B If A”. Try applying contrapositive to  $A \Rightarrow B$  to know what else you can infer from  $A \Leftrightarrow B$ .

$$\frac{A \Leftrightarrow B}{B \Rightarrow A} \text{ If and Only If 2}$$

“A If and only If B means” “If B then A” or “A If B”. Try applying contrapositive to  $B \Rightarrow A$  to know what else you can infer from  $A \Leftrightarrow B$ .

$$\frac{A \quad A \wedge B \Rightarrow C}{B \Rightarrow C} \text{ Simplification of Implication}$$

When a rule has too many things “stacked up” before the  $\Rightarrow$ , you can get rid of some of them.

$$\frac{A \wedge B \Rightarrow C \vee D}{A \wedge B \neg C \Rightarrow D} \text{ Moving Around Implication}$$

You can move things around the implication by negating in the process. Imagine the  $\Rightarrow$  to have an  $\wedge$ -stack on the left and a  $\vee$ -stack on the right.

$$\frac{A \wedge B \Rightarrow C \vee D}{A \Rightarrow \neg B \vee C \vee D} \text{ Moving Around Implication}$$

You can move things around the implication by negating in the process. Imagine the  $\Rightarrow$  to have an  $\wedge$ -stack on the left and a  $\vee$ -stack on the right.

$$\frac{A \wedge B \wedge \neg C \Rightarrow \text{False}}{A \wedge B \Rightarrow C} \text{ Proof By Contradiction Again}$$

This sharply illustrates proof by contradiction.

## 8.8.2 More Examples

### A Made-up Example (shows steps for “Wise Young Pigs”)

This made-up example shows what skills you will need for the “wise young pigs” in your Assignment 2.

**NOTE:** For Assignment 2, you are encouraged to use single-letter variable names. Rename epb as a, old as b, etc. Else it gets quite tedious!

- From the premises

P1.  $a \cdot b \Rightarrow c$   
 P2.  $c \Rightarrow d$   
 P3.  $e \Rightarrow b$   
 P4.  $e \cdot f \Rightarrow \neg d$   
 P5.  $f \Rightarrow a$

Infer the goal C

C:  $e \Rightarrow \neg f$

- Proof:

1. Negate the desired conclusion C and add to the list of premises.

P1.  $a \cdot b \Rightarrow c$   
 P2.  $c \Rightarrow d$   
 P3.  $e \Rightarrow b$   
 P4.  $e \cdot f \Rightarrow \neg d$   
 P5.  $f \Rightarrow a$   
 P6.  $e$  -- from negated goal  
 P7.  $f$  -- from negated goal  
 P8.  $f \Rightarrow \neg d$  -- Simplify P4 using P6  
 P9.  $\neg d$  -- MP of P7 and P8  
 P10.  $a$  -- MP of P7 P5  
 P11.  $b \Rightarrow c$  -- Simplify P1 using P10  
 P12.  $b \Rightarrow d$  -- Chain P11 and P2  
 P13.  $b$  -- MP of P6 and P3  
 P14.  $d$  -- MP of P13 and P12  
 P15. False -- P9 and P14 contradict.

### Another Example from Carroll

Here is yet again from “the master,” courtesy of

[http://www.yesfine.com/carroll\\_symbolic\\_logic\\_simplified\\_statements.htm](http://www.yesfine.com/carroll_symbolic_logic_simplified_statements.htm).

**"All my dreams come true"** From

1. Every idea of mine, that cannot be expressed as a Syllogism, is really ridiculous;
2. None of my ideas about Bath-buns are worth writing down;
3. No idea of mine, that fails to come true, can be expressed as a Syllogism;
4. I never have any really ridiculous idea, that I do not at once refer to my solicitor;
5. My dreams are all about Bath-buns;
6. I never refer any idea of mine to my solicitor, unless it is worth writing down.

Modeling hints:

- Universe: "my idea";
- a = able to be expressed as a Syllogism;
- b = about Bath-buns;
- c = coming true;
- d = dreams;
- e = really ridiculous;
- h = referred to my solicitor;
- k = worth writing down.

**Proof of All My Dreams**

P1.  $\neg a \Rightarrow e$   
P2.  $b \Rightarrow \neg k$   
P3.  $\neg c \Rightarrow \neg a$   
P4.  $e \Rightarrow h$   
P5.  $d \Rightarrow b$   
P6.  $h \Rightarrow k$

Goal.  $d \Rightarrow c$

Negated goal gives rise to P7 and P8 below

P7.  $d$   
P8.  $\neg c$   
P9.  $b$  -- P7 and P5, MP  
P10.  $\neg k$  -- P9 and P2, MP  
P11.  $\neg h$  -- P10 and contrapositive of P6  
P12.  $\neg e$  -- P11 and contrapositive of P4  
P13.  $a$  -- P12 and contrapositive of P1  
P14.  $c$  -- P13 and contrapositive of P3  
P15. False -- P8 and P14

# Chapter 9

## Midterm Review

This chapter will try to take stock of all the material learned over the labs, lectures, and assignments. There are concepts we've seen, symbols and notations we've seen, and techniques we've seen. The test itself will remind you of symbols and notations as much as is reasonable. So you may focus on getting the concepts, techinques, and approaches right. The basic abilities to count, enumerate, reason, and know what is going on at any moment will be what we will test. With that, here we go. Our discussions will be organized in terms of main topic areas.

### 9.1 Sets

A set is an unordered collection or grouping of items. A circle you draw on paper (with a pencil) is a set (of graphite flakes that are a few hundreds of atoms thick). A mathematical circle is an infinite set (of points). The coordinates on a (finite piece of a) graph paper are a finite set of points.

#### 9.1.1 Operators

You should know how sets can be operated upon using the operators  $\cup$ ,  $\cap$ ,  $-$ ,  $\setminus$  (same as  $-$ ), complement (overbar, such as  $\bar{X}$ ), and  $\Delta$ .

Test your understanding:

1. Section 6.1, Page 85. Here,  $D$  is the domain over which the set is formed—in other words, the universal set.

2. Look at the set enumeration in Section 6.2.1 of the book.
3. Examples in Section 6.2.2 of the book.
4. Examples in Section 6.4 of the book.
5. How many distinct regions are there in a general Venn diagram involving 13 sets (see Section 5.1.2 for an answer)
6. Show that  $((A - C) \cup (B - C)) \subseteq \overline{C}$ . Illustrate a proof using a Venn diagram.
7. Show that  $((A - C) \cup (B - C)) \subseteq \overline{C}$ . Write a formal proof. This requires you to write a proof down to the level of set-theoretic arguments, defining each operator. (See proof below.)
8. Make sure you know how to do Problem 10 of Quiz 1.

### 9.1.2 Proof of $((A - C) \cup (B - C)) \subseteq \overline{C}$

$P \subseteq Q$  means for every  $x$ ,  $x \in P \Rightarrow x \in Q$ .

$$\text{LHS: } \{x \mid (x \in A \wedge x \notin C) \vee (x \in B \wedge x \notin C)\}$$

$$= \{x \mid (x \in A \wedge \neg x \in B) \wedge x \notin C\}$$

$$\text{RHS: } \{y \mid y \notin C\}$$

To show that LHS  $\subseteq$  RHS, show that

$$(x \in A \wedge \neg x \in B) \wedge x \notin C \Rightarrow x \notin C$$

Follows  $\Rightarrow$ 's definition.

## 9.2 Functions

We encountered functions right from Chapter 0. These were, however, “Python functions” that were supported by a computer program. These are also functions in the sense that given inputs are mapped to specific outputs. How-

ever, as soon as a global variable is manipulated inside a function in a programming language, the behavior of that function is no longer modeled by a mathematical function that works purely in terms of inputs and outputs.

### 9.2.1 Calculating the Number of Ternary Functions

Someone makes a sudden progress in semiconductor fabrication techniques by devising three voltages that can be reliably encoded. People now start making ternary gates that produce 0, 1, 2 given inputs that range over 0, 1, 2. The word **ternary** means “three values” (the values are {0, 1, 2}) similar to the word **binary** meaning “two values” (namely {0, 1}). Let us do a few exercises to drive home this thinking.

**$N$ -ary ternary gate types producing ternary outputs:** How many  $N$ -ary ternary gate types exist, assuming that the outputs are also ternary?

*Solution:* Imagine all possible  $N$ -ary ternary “truth tables.” There are  $3^N$  rows to the truth table. There are  $3^{3^N}$  personalities possible, because we can fill each output with one of three values..

**Repeat the calculation for ternary-to-binary gates:** That is, the signature of the functions being realized is

$$\{0, 1, 2\} \times \dots (N \text{ times}) \times \{0, 1, 2\} \rightarrow \{0, 1\}$$

In other words, these are functions that still produce a binary output from ternary inputs. Now the calculations change.

*Solution:* The number of rows to fill is still  $3^N$ ; however, the number of ways each row can be filled is 2; thus we have  $2 \times 2 \times \dots \times 2$  (repeated  $3^N$  times) for a total of  $2^{3^N}$ .

**Generalizing the calculation:** Let us assume that there are  $N$  inputs for a gate-type, and assume that each of the inputs are  $P$ -ary. Assume that the outputs are  $Q$ -ary. How many gate types of signature

$$\{0, 1, \dots, (P - 1)\} \times \dots (N \text{ times}) \times \{0, 1, \dots, (P - 1)\} \rightarrow \{0, 1, \dots, (Q - 1)\}$$

are there?

*Solution:* Then there are  $P^N$  rows in the “personality”, each of which can be filled in  $Q$  different ways. Thus we have a total of  $Q^{P^N}$  gate types of signature

### 9.2.2 Universal Ternary Gates?

Let's say that the world has completely switched over to Ternary gates! Again there are too many Ternary gates, and so people are now in search for a universal Ternary gate! What would it be? Describe its operation in sufficient detail (English would do) and show how it can serve as a universal gate?

*Solution:* We can devise a new MUX type and use it in a very similar fashion to build any ternary function, replicating the gate as a tree. Instead of a mux21, we will devise a TMux31, which is a ternary mux of three inputs and one ternary output. One possible Python description that realizes this mux would be as follows:

```
def TMux31(s,a,b,c):
    return a if (s == 0) \
        else b if (s == 1) \
        else c if (s == 2) \
        else "Erroneous s value"
```

*Exercise:* Try defining an arbitrary ternary function by hooking-up TMux31 gates into a ternary tree.

## 9.3 Boolean Algebra and Propositional Logic

### 9.3.1 Boolean Simplification 1

We will soon be learning how to simplify Boolean functions using Karnaugh maps. The main identities we shall use in this approach are listed below. Here,  $x$ ,  $y$ , and  $z$  can be any arbitrary Boolean expression. Prove that these identities are true.

- $x.y + !x.y = y$ .
- $x.y.z + !x.y.z + x.!y.z + !x.!y.z = z$

*Solution:* In the first identity, factor out  $y$ , giving us

$$(x + !x).y$$

which reduces to  $y$ .

For the second identity, factor out  $z$ , and then two expressions involving  $y$  and  $!y$ . In detail, we can write this expression as

$$z.(y.(x + !x) + !y.(x + !x))$$

which reduces to  $z$ .

### 9.3.2 Boolean Simplification 2

Prove that the following code actually swaps two bits  $x$  and  $y$ , where  $\wedge$  is the XOR operator:

```
x = x ^ y
y = x ^ y
x = x ^ y
```

### 9.3.3 If-Then-Else Normal Form

The If-then-else normal form, or *Shannon's Expansion*, is a direct way to implement a Boolean function. It breaks down a Boolean function by following two cases:

- Some variable  $x$  in the function is true
- The same variable is false

It specializes the function for each of these cases. Realize the staircase switch using the if-then-else normal form.

## 9.4 Proofs, and What They Mean

### 9.4.1 Validity (or Tautology) and Satisfiability

Given three variables  $x, y, z$ ,

- present a valid Boolean formula (or expression) over  $x, y, z$ , that involves all three variables. A valid formula is “always true”—true for all variable assignments. The complement of a valid formula is always false (\*: see below).
- present a satisfiable but not valid Boolean formula (or expression) over  $x, y, z$ , that involves all three variables. A valid formula is satisfiable. A satisfiable but non-valid formula is falsifiable. In other words, its complement is also satisfiable (compare with what we said against “\*” above).

### 9.4.2 Rules of Inference

Someone suggests a rule of inference  $\frac{A \Rightarrow B}{B \Rightarrow \neg A}$  Contraband

Is there anything wrong if ‘Contraband’ is allowed as a rule of inference?

*Solution:* This proof rule is unsound. If this were a sound rule, the following formula must be valid:

$$(A \Rightarrow B) \Rightarrow (B \Rightarrow \neg A)$$

Unfortunately, this is not valid: take  $B = 1$  and  $A = 1$ , which falsifies the whole formula.

## 9.5 Miscellaneous

One should be able to do any of the standard problems we discussed:

- Proof by contradiction. Also be able to explain how this proof works
- Given any rule of inference, be able to justify whether it can be included or not
- Apply a given set of proof-rules or Boolean simplification rules
- Defining a gate using Python definitions of simpler gates
- Defining a gate using a direct hash-table implementation

### 9.5.1 Connections to Practice

- One should be able to appreciate the importance of gate-based designs.
- One should be able to take an arbitrary truth-table and realize it using and/or or nand/nor or nor/nor logic
- One should be able to understand the rules of Boolean algebra involved in these realizations

### 9.5.2 For extra insight into the material

Be able to puzzle through the material in Section 6.5. Do you see how this Python code generates all functions over a given number of inputs, and allows you to pick and choose from these functions?

### 9.5.3 All the ways to implement 0 and 1

We often take 0 and 1 for granted; however, these primitive symbols often have physical realizations. Some of these are listed below:

- Relays: on or off.
- Vacuum tubes: on or off.

- Magnetic encodings on disk: magnetized or not.
- Transistors, flip-flops: on or off; flip-flop states.
- Fluidic gates: which side the air is stuck to in a fluidic cavity. See the Wikipedia for details.
- Mercury delay lines: the presence of a pressure point in sound circulating in a closed mercury column.
- Laser pulses circulating in a closed optical loop (periodically amplified); this was once a “crazy” proposal I heard (ISCA 1989) from a scientist at Bell Labs (similar to mercury delay lines).
- Carbon nanotubes that fly: Again, I’ve once heard of a scheme where one can bend and make a carbon nanotube to stick to the surface of a piece of silicon. The bent/unbent shape modulates charge apparently.
- Charge in a floating-gate: trapped charge (present or not).
- Amorphous/crystalline state changes: this reflects light in a polarized (or non-polarized) fashion. How writeable DVDs work.
- Fuses and Anti-fuses: used in FPGA to get the “write once” effect.

Can you add to this list? Can you find historical pictures and other details associated with each of the above methods?

#### 9.5.4 Historic Video from the MULTICS Era

REALLY inspiring video of how young computing is: Dr. Fernando Corbato explains time-sharing at MIT in 1963.

<http://www.youtube.com/watch?v=Q07PhW5sCEk>

I’ll also ask a few exam questions from this video!

## 9.6 Midterm-1 and Its Solution

Let’s now solve the midterm exam actually given.

1. **(5 points)** List five things you learned from Dr.Corbato’s lecture. Cover these points, plus two that you observed outside of this list, in one sentence each: (1) why timesharing was considered important; (2) what the input-output devices used were; (3) the programming language or the type of programs (what calculations) were run on these machines.

Answer as you best can.

2. **(15 points)** Consider this function which represents the Sum output of a half-adder. Derive a NOR-NOR realization using the techniques learned in class (obtain product-of-sums, then Or-And, then convert).

a	b	c	Sum
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Answer: The answer is obtainable from following the 0s. The function is

$$\text{!Sum} = \text{!a}.\text{!b}.\text{!c} + \text{!a.b.c} + \text{a}.\text{!b}.c + \text{a.b}.\text{!c}$$

and so the product of sums expression is:

$$\text{Sum} = (a + b + c).(a + \text{!b} + \text{!c}).(\text{!a} + b + \text{!c}).(\text{!a} + \text{!b} + c)$$

which turns directly into an OR/AND or NOR/NOR form directly as the above expression is structured.

3. **(10 points)** Using only 2-to-1 multiplexors, realize the *Sum* function described in the truth-table above. Use *any* method.

Answer: You know the straightforward method using personalities. Let me illustrate another method that uses the if-then-else normal form:

$$\begin{aligned} & (a + b + c).(a + \text{!b} + \text{!c}).(\text{!a} + b + \text{!c}).(\text{!a} + \text{!b} + c) \\ &= a.((b + \text{!c}).(\text{!b} + c)) + \text{!a}.((b + c).(\text{!b} + \text{!c})) \\ &= a.(b.(c) + \text{!b}.(\text{!c})) + \text{!a}.(b.(\text{!c}) + \text{!b}.(c)) \end{aligned}$$

This now can be rendered as a MUX-based circuit as described by this Python function (we supply a convenient `mux21` definition for local use):

```
def mux21(s,i1,i0):
    return i0 if (s == 0) else i1

def sum(a,b,c):
    mux21(a, mux21(b, c, !c), mux21(b, !c, c) )
```

4. **(5 points)** Is  $S_2 = \{AND, OR\}$  a universal set of gates? If so, show all the steps concisely. If not show why not.

Answer: They are not universal, as we can realize only six of the 16 Boolean functions, as we've already seen before ( $0, 1, a, b, a.b$ , and  $a + b$ ).

5. **(10 points)** How many 5-input Ternary to Binary functions are there. Derive the formula for this case. Ternary means  $\{0, 1, 2\}$  while Binary means  $\{0, 1\}$ . All five inputs are Ternary and there is a single Binary output.

Answer: Since there are five ternary inputs, there are  $3^5$  rows. Each row can be filled in 2 ways. The total is  $2^{3^5}$ .

6. **(10 points)** Draw a few Venn diagrams for sets  $A, B, C$  to check whether

$$(A \cap B) - C = (A - C) \cap (B - C)$$

is true. If you find a condition " $P$ " under which the above equation is **not** true, write down  $P$ . Then explain whether this equation is true if  $\neg P$ . Else write a formal proof (using the set-comprehension notation, or directly in terms of sets) that this equation is (always) true.

Answer: It is always true! Here is the proof using set-theory:

$$\begin{aligned} & (A \cap B) - C \\ &= (A \cap B) \cap \bar{C} \\ &= (A \cap B) \cap (\bar{C} \cap \bar{C}) \\ &= (A \cap \bar{C}) \cap (B \cap \bar{C}) \\ &= (A - C) \cap (B - C) \end{aligned}$$

7. **(10 points)** List **three elements** of the following set.

$$S = \{(x, y, z) \mid x, y, z \in range(10) \text{ and } odd(x + y) \Rightarrow even(x - z)\}$$

You must find each of the three members you list by making *odd* and *even* (above) true in all combinations. Your answer must be clear bulleted steps of the form "I found these x,y,z that made odd true/false and even true/false." (say which). Do this three times over, one for each possible way to admit an element.

8. (5 points) List **three non-elements** of the above set, giving explanations.

- Make odd and even false:  
Pick  $x = y = 1$  and  $z = 0$ .
- Make odd and even true:  
Pick  $x = 2$   $y = 1$  and  $z = 0$ .
- Make odd false and even true:  
Pick  $x = y = 2$  and  $z = 0$ .  
For a non-element, make odd true and even false:  
Pick  $x = 2$   $y = 1$  and  $z = 1$ .

9. (4 points) Is this a valid inference rule; why or why not?  

$$\frac{A \vee (B \wedge C)}{(A \wedge B) \vee C} \text{ Suspicious}$$

Answer: No it is not, because it says

$$(A + (B.C)) \Rightarrow ((A.B) + C)$$

is valid; and it is not (pick  $A = 1, B = 0, C = 0$ ).

10. (6 points) Identify which of these are valid and which are satisfiable but not valid, giving reasons:

(a)  $(a \Rightarrow b) \wedge (a \vee b) \wedge (b \Rightarrow a)$  — also written as  $(a \Rightarrow b).(a + b).(b \Rightarrow a)$

Answer: Not valid but satisfiable. It boils down to  $a$  or  $b$  because we have  $a = b$  here in disguise. We can set  $a = b = 0$ .

(b)  $((a \Rightarrow b) \wedge (a \vee b)) \Rightarrow (b \oplus a)$  — also written as  $((a \Rightarrow b).(a + b)) \Rightarrow (b \oplus a)$

Answer: Not valid but satisfiable. One can set  $a = b = 1$  to get the unsat.

(c)  $((a \wedge (a \Rightarrow b) \wedge \neg b)) \Rightarrow (a \oplus b)$  — also written as  $((a.(a \Rightarrow b).\neg b)) \Rightarrow (a \oplus b)$

Answer: Case analysis on antecedent: case antecedent false: true. Antecedent can't be made true because antecedent simplifies to false. So done. valid.

11. (5 points) Someone tries to infer  $b \Rightarrow a$  from  $a \Rightarrow b$  and  $a \vee b$  using proof by contradiction. Will this proof succeed? Explain why.

Answer: The proof will be stuck, as

$$((a \Rightarrow b) \cdot (a \vee b)) \Rightarrow (b \Rightarrow a)$$

is not valid. Pick  $b = 1$  and  $a = 0$ ; then the above formula is false. Hence this proof will be stuck.

12. (5 points) How will the person in Question 11 use z3Py to set up the above problem as a proof-by-contradiction, and what will Z3Py likely tell them, giving feedback on the situation?

Answer: Set up the problem by putting the facts and the negated goal into Z3py. Z3Py will produce the above satisfying assignment from which the user figures out that the contradiction wasn't achieved (what was supposed to be false was shown true by the above assignment  $b = 1$  and  $a = 0$ ).

13. (10 points) Write a formal proof by contradiction:

From premises

$$P1: \neg b \Rightarrow \neg a$$

$$P2: b \Rightarrow c$$

$$P3: (\neg d \vee \neg e) \Rightarrow \neg c$$

Prove Goal:

$$\text{Goal: } a \Rightarrow \neg(d \oplus e)$$

The proof-rules you may need are:

Chaining: From  $a \Rightarrow b$  and  $b \Rightarrow c$ , infer  $a \Rightarrow c$ .

Contrapositive: From  $a \Rightarrow b$ , infer  $\neg b \Rightarrow \neg a$ .

Answer: By taking the contrapositive of P1, chaining with P2 and then with the contrapositive of P3, we get  $a \Rightarrow (d \cdot e)$ . Now, since  $a$  is asserted (negated goal), we obtain  $(d \cdot e)$ . Now we have  $d \cdot e$  and  $d \oplus e$  which amounts to *False*. Hence proved.



# Chapter 10

## Karnaugh Maps

A **Karnaugh map** (or K-map) is a way to plot the truth-table of a Boolean function, *interpreted as a sum-of-products expression*, in a rectangular region. We aim to ensure that each adjacent pairs of cells in the horizontal or vertical direction differ by one bit. See Figure 10.1 for example of a 3-variable K-map.<sup>1</sup>

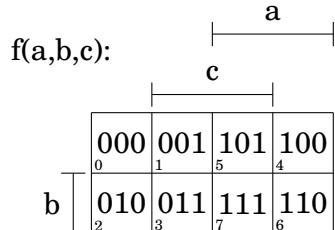


Figure 10.1: A 3-variable Karnaugh Map

This K-map's notion of “adjacency” is explained in more detail:

- K-maps are *wrapped around*. Thus, cells 000 and 100 are considered adjacent—again because they differ in one bit.
- Diagonal elements are *not* adjacent; thus, clearly 001 and 111 are not adjacent—again because they differ in more than one bit.

The purpose of plotting K-maps in this fashion is to be able to visualize opportunities to apply the following identities:

Identity-1:  $x.y + !x.y = y$ .

---

<sup>1</sup>We usually will depict K-maps of upto four variables; for more than four variables, the K-map method does get tedious.

$$\text{Identity-2: } x.y.z + !x.y.z + x.!y.z + !x.!y.z = z$$

### 10.0.1 Example of K-Map Usage

To see how K-maps help simplify expressions and see relationships between expressions, let us take an example. Suppose someone is interested in function  $LHS = ab + bc + c\bar{a}$ . Soon, suppose someone else claims that function  $LHS$  is equivalent to  $RHS = ab + c\bar{a}$ .

#### Portraying $LHS$ and $RHS$ via a Truth-table

Of course we can settle this claim by brute-force using truth-tables! In fact, Figure 10.2 shows that indeed both sides of the equation  $ab + bc + c\bar{a} = ab + c\bar{a}$  evaluate to the same truth value.

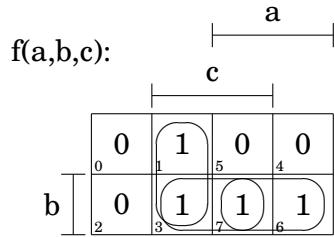
$a$	$b$	$c$	$ab$	$bc$	$c\bar{a}$	$LHS = ab + bc + c\bar{a}$	$RHS = ab + c\bar{a}$
0	0	0	0	0	0	0	0
0	0	1	0	0	1	1	1
0	1	0	0	0	0	0	0
0	1	1	0	1	1	1	1
1	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	1	1	0	1	1

Figure 10.2: Conjecture  $ab + bc + c\bar{a} = ab + c\bar{a}$  shown through a Truth Table

#### Settling $LHS = RHS$ using K-maps

We seek to settle this claim using K-maps. In a K-map, we depict function  $LHS$  as in Figure 10.3. *K-maps are nothing but handy ways of drawing Truth-tables so that we can use our visual intuitions to reason about the function in question.* They also help us visually identify opportunities for simplifying Boolean functions by identifying *covers* (explained more below; basically the “oval lassos” in our K-map drawings).

The reasoning for this portrayal in a K-map is as follows. First, we must show how truth-combinations for the function of interest are plotted on a rectangle with  $2^3 = 8$  cells. For each triple of inputs (000, 001, etc.), the truth-values of the function are entered into the cells. Each “1” that we enter

Figure 10.3: Karnaugh map for  $ab + bc + c\bar{a}$ 

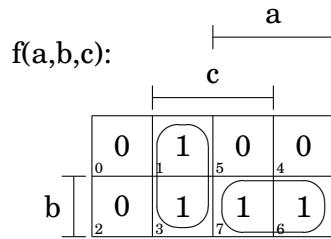
into the map corresponds to a **product term**. Each oval “lasso” around the 1s is known as a *cover*. Covers are **simplified product terms**.

- Thus, in Figure 10.3, there are 1s entries at four squares:
  - At 001, 011, 111, and 110.
  - These correspond to  $\bar{a}\bar{b}c$ ,  $\bar{a}bc$ ,  $abc$ , and  $a\bar{b}\bar{c}$ .
  - The cover (oval) encompassing  $\bar{a}\bar{b}c$  and  $\bar{a}bc$  represents the summation  $(\bar{a}\bar{b}c + \bar{a}bc)$ , which simplifies to  $\bar{a}c$  (since  $b + \bar{b}$  is 1), or equivalently  $c\bar{a}$ . Incidentally, this oval is the left-most oval “standing on its edge” in Figure 10.3.
  - Thus, we obtained one of the three disjuncts of *LHS* portrayed. In the process, notice that we applied Identity-1 mentioned above! **This is how K-maps help us see opportunities for simplifications.**
  - Likewise, the oval spanning cells 011 and 111 (the one “lying on its side” and is left-most in Figure 10.3) represents  $bc$  (another disjunct of *LHS*).
  - The oval spanning cells 111 and 110 (the one “lying on its side” and is right-most in Figure 10.3) represents  $ab$  (the third and final disjunct of *ab*).

To further see how K-maps help, suppose someone else claims that function *LHS* is equivalent to  $RHS = ab + c\bar{a}$ . Well, this is hard to see “at a glance.” But in a K-map, we depict *RHS* as in Figure 10.4, using the same conventions as explained above:

Now we see from the above K-map that **both Figure 10.3 and Figure 10.4 cover the same 1 spots**. The only thing that *LHS* has over and above *RHS* is the product term  $bc$ . Where does  $bc$  cover?

- It covers  $\bar{a}bc$ , which is already covered by  $\bar{a}c$  (vertical oval spanning 001 and 011).

Figure 10.4: Karnaugh map for  $ab + c\bar{a}$ 

- It covers  $abc$  term which is already covered by  $ab$  (horizontal oval spanning 111 and 110).

Thus, since  $LHS$  does not cover any more spots than  $RHS$  (and vice-versa), these are equivalent functions.

The Wikipedia article [http://en.wikipedia.org/wiki/Karnaugh\\_map](http://en.wikipedia.org/wiki/Karnaugh_map) has a very good description of K-maps. We will also study K-maps using the many references provided on our Moodle page.

### Some Terminology: *Minterms*

Refer to Figure 10.4 for the sake of an example. This also corresponds to the Truth Table of Figure 10.2 (right-most column). There is a standard way of naming truth-table rows using the terminology of *Minterms*.

Given a Boolean function of  $N$  variables, a *Minterm* is a product term that lists each of these  $N$  variables, either in *true* form or *complemented* form. In the truth-table of Figure 10.2—a truth-table over variables  $\{a, b, c\}$ , the minterms are:

- $m_0 = !a.!b.!c$  (equivalently,  $m_0 = \bar{a}.\bar{b}.\bar{c}$ )
- $m_1 = !a.!b.c$  (equivalently,  $m_1 = \bar{a}.\bar{b}.c$ )
- $m_2 = !a.b.!c$  (equivalently,  $m_2 = \bar{a}.b.\bar{c}$ )
- $m_3 = !a.b.c$  (equivalently,  $m_3 = \bar{a}.b.c$ )
- $m_4 = a.!b.!c$  (equivalently,  $m_4 = a.\bar{b}.\bar{c}$ )
- $m_5 = a.!b.c$  (equivalently,  $m_5 = a.\bar{b}.c$ )
- $m_6 = a.b.!c$  (equivalently,  $m_6 = a.b.\bar{c}$ )
- $m_7 = a.b.c$  (equivalently,  $m_7 = a.b.c$ )

It is easy to see that each “cell” of a K-map (each square) corresponds to one of these Minterms.

### 10.0.2 There are $2^{2^3}$ 3-variable K-maps!

Well, what else did you expect? Given that K-maps are “pretty truth-tables,” and there are  $2^2 = 256$  truth-tables of 3-variables, there must also be so many K-maps! Here are some of them (Figure 10.5):

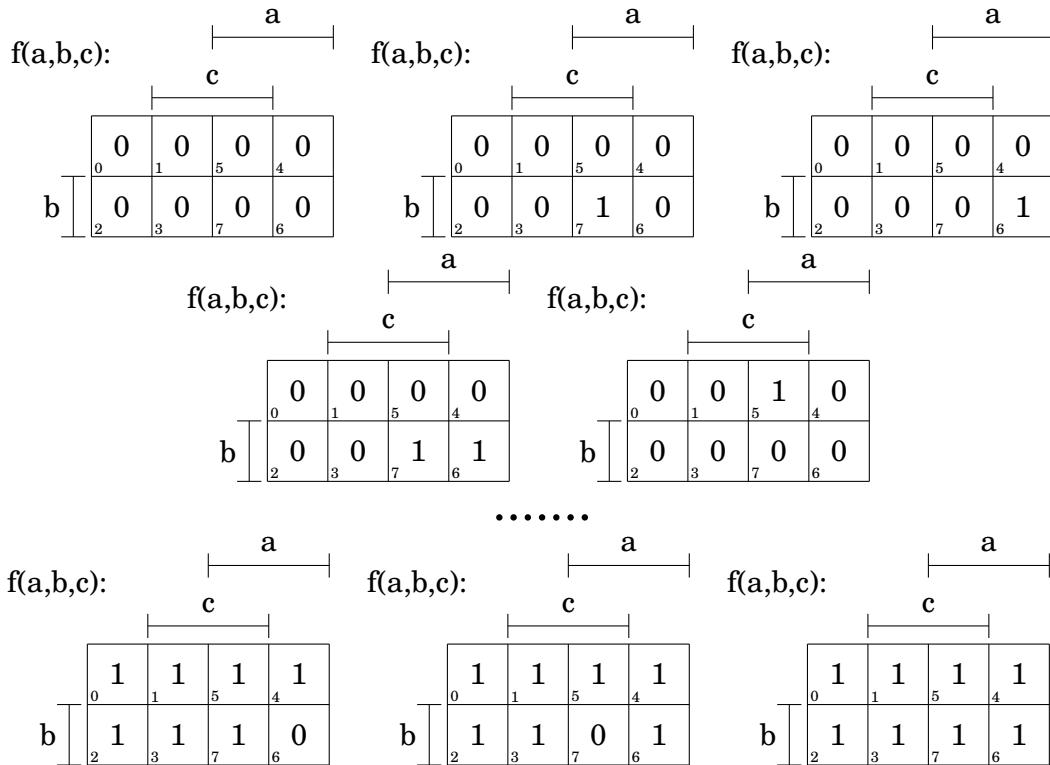


Figure 10.5: All possible 3-variable Karnaugh maps (256 of them!)

### 10.0.3 Generalized Simplification

Let us now see how Identity-2 can be used in a K-map setting (and in general, any such identity):

- If in a K-map we have the four squares 000, 001, 010, and 011 filled with a 1, we can simplify the expression to  $\bar{a} \cdot (\bar{b}\bar{c} + \bar{b}c + b\bar{c} + bc) = a$ .
- The above is true for any four adjacent squares where two of the bits go about through *all four combinations*.

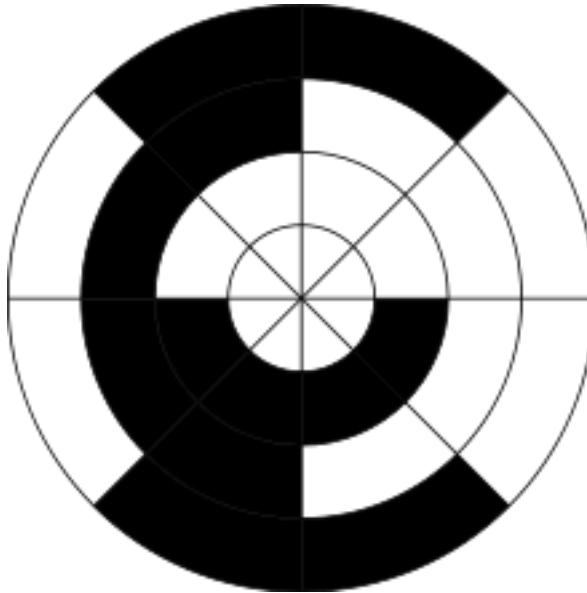


Figure 10.6: A Gray Code Wheel of 3 bits (courtesy of the Wikipedia article)

For example, if 000, 010, 100, and 110 are marked with a 1, we can simplify the expression to  $\bar{c}(\bar{a}\bar{b} + \bar{a}b + a\bar{b} + ab) = \bar{c}$ .

#### 10.0.4 K-maps and Gray Code Sequences

K-maps are closely related to Gray codes (described in [http://en.wikipedia.org/wiki/Gray\\_code](http://en.wikipedia.org/wiki/Gray_code)). A Gray code wheel for three bits is shown in Figure 10.6. Notice that this is a wheel which ensures that only one bit changes at a time (light gets interrupted when a black comes) and is useful to sense the rotation of this disk. **But there is never the case that a black turns white and a white turns black.** Such “double transitions” will be highly error-prone because we can’t make very precise sensors and measurement devices.

#### Undirected Graphs and Hamiltonian Cycles

An undirected graph consists of vertices (or nodes)  $V$  and edges  $E$ . The vertices (nodes)  $V$  are a set of names, and the edges  $E$  are a set of pairs of names. We will use “vertex” and “node” interchangeably, but denote it by  $V$ .

Since the graph is undirected, we don't care how we list the edges (in what order; thus  $(a, b)$  and  $(b, a)$  are equally valid listings of the same edges).

For example, consider the graph described by the state capitals of Utah (SLC), Nevada (CSN or Carson City), Arizona (Phoenix or PHX), and California (Sacramento, or SMF) going by the airport codes. Let every pair of capitals be connected. We obtain the graph shown in Figure 19.1. This graph, say " $G_{SC4}$ " standing for "the graph of four state capitals" is described as follows:

$$G_{SC4} = (V, E)$$

where

$$V = \{SLC, CSN, PHX, SMF\}$$

and

$$E = \{(SLC, CSN), (SLC, SMF), (SLC, PHX), (CSN, PHX), (CSN, SMF), (SMF, PHX)\}$$

There should be *six* edges, because we are choosing from *four* (4) capitals, taking *two* (2) at a time; or  $\binom{4}{2}$ , as we will soon study/recap. We can also present the graph as

$$\begin{aligned} G_{SC4} = & \\ & \{(SLC, CSN, PHX, SMF), \\ & \{(SLC, CSN), (SLC, SMF), (SLC, PHX), (CSN, PHX), (CSN, SMF), (SMF, PHX)\} \end{aligned}$$

Let us place a traveling salesperson in SLC and ask her to visit each state capital exactly once, returning to SLC (the return "touch" of SLC does not count as the second visit of SLC; the starting "touch" of SLC does count).

Clearly, the salesperson can visit in any one of these orders:

- SLC, PHX, SMF, CSN (and back to SLC, not explicitly listed).
- SLC, PHX, CSN, SMF (and back to SLC, not explicitly listed).
- SLC, CSN, SMF, PHX (and back to SLC, not explicitly listed).
- etc. etc. (many such – how many?)

Each of these is a Hamiltonian cycle.

CSN---SLC



SMF---PHX

Figure 10.7: An Undirected Graph of Four State Capitals of the Western US

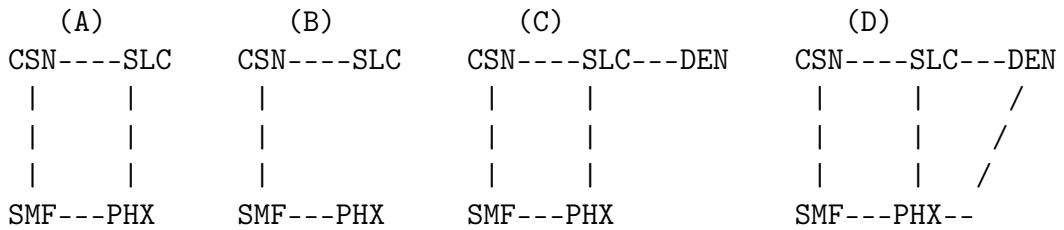


Figure 10.8: A Modified Graph of Four States

**Hamiltonian cycle:** Given an undirected graph  $G$ , a *Hamiltonian cycle* is a *sequence* listing the nodes of  $G$  exactly once, starting with an arbitrary node  $n \in V$  of  $G$ , such that each adjacent pair of nodes in  $G$  is connected by an edge that is in  $E$ . For our example graph, different Hamiltonian cycles were listed above.

**Hamiltonian Cycle for Modified Graphs** In Figure 19.2(A), there are fewer Hamiltonian cycles. In Figure 19.2(B), there *no* Hamiltonian cycles (for what reason?). In Figure 19.2(C) also, there *no* Hamiltonian cycles (for a different reason; what is it?). In Figure 19.2(D), there are many Hamiltonian cycles (how many?), but a tour that goes as DEN, PHX, SLC, . . . cannot be extended to become a Hamiltonian cycle (why?).

### Each Gray Code Sequence Describes a K-map Hamiltonian Cycle

There is an elegant way to generate a Gray Code sequence:

- Put a traveling salesperson in a K-map (say in cell 000).
- Tell the salesperson to visit all K-map squares, traveling in the sense of K-map adjacency.
- The salesperson may visit each city **exactly once**.
- The salesperson must return to cell 000.

**Exercise:** Look at Figure 10.6, and describe which Hamiltonian cycle is being described. Now generate three other Hamiltonian cycles. Draw the Gray-code sequences corresponding to these cycles.

**Solution:** This Gray code wheel goes as follows, starting from 000, where we read 0 as black and 1 as white, and read from the axis to the circumference: 000, 010, 011, 111, 110, 100, 101, 001 (and back to 000).

**Exercise:** Plot the above Gray-code sequence on the K-map of Figure 10.4.

**Solution:** The above Gray code sequence visits the cells of the K-map as follows:

000, 010, 011, 111, 110, 100, 101, 001 (and back to 000)

These translate to the following Minterms, referring to the variables in the order  $a$ , then  $b$ , then  $c$ :

$\neg a \cdot \neg b \cdot \neg c$ ,  $\neg a \cdot b \cdot \neg c$ ,  $\neg a \cdot b \cdot c$ ,  $a \cdot b \cdot c$ ,  $a \cdot b \cdot \neg c$ ,  $a \cdot \neg b \cdot \neg c$ ,  $a \cdot \neg b \cdot c$ ,  $\neg a \cdot \neg b \cdot c$  (and back to  $\neg a \cdot \neg b \cdot \neg c$ ).

**Exercise (Bieber):** Consider the following propositions:

- *does not go astray*:  $\neg a$
- *does not balk at complexity*:  $\neg b$
- *is creative*:  $c$

Suppose Justin Bieber tickets are given to those who simplify the following conditions the most:

- The person is  $\neg a \cdot \neg c \cdot \neg b$
- The person is  $\neg a \cdot \neg c \cdot b$
- The person is  $\neg a \cdot c \cdot \neg b$
- The person is  $a \cdot c \cdot b$
- The person is  $a \cdot \neg c \cdot \neg b$
- The person is  $a \cdot \neg c \cdot b$

**Solution:** Plotting on a K-map (say, of Figure 10.4), we obtain the following minterms:

- $\neg a \cdot \neg c \cdot \neg b$
- $\neg a \cdot \neg c \cdot b$
- $\neg a \cdot c \cdot \neg b$
- $a \cdot c \cdot b$
- $a \cdot \neg c \cdot \neg b$
- $a \cdot \neg c \cdot b$

Finding maximal covers, we get

- $\neg c + \neg a \cdot \neg b + a \cdot b$

Thus, one gets a Bieber ticket if one is “(non-creative) OR (does not go astray AND does not balk at complexity) OR (goes astray and balks at complexity).”



# Chapter 11

## Binary Decision Diagrams

Boolean functions are fundamentally important, both as conceptual tools and as concrete representations in several real-world modeling and analysis situations. Given the need to represent large Boolean functions (say, those involving dozens of Boolean variables), it is important to have practical (scalable) representations. Unfortunately, truth tables and Karnaugh maps are *not* scalable or practical for these sizes! While one may represent a Boolean function of a few inputs *e.g.*, And using a truth table, even something conceptually as simple as a magnitude comparator—comparing whether two bytes (8-bit words) are equal—requires us to employ a 16-input truth table. This truth-table will have 65,536 rows—somewhat like this:

b7	b6	b5	b4	b3	b2	b1	b0	a7	a6	a5	a4	a3	a2	a1	a0	f
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
...																
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0
...																
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Clearly, working with a truth-table of 65,536 rows (or a K-map with 65,536 cells) is not practical. Fortunately, there is an alternative representation of Boolean functions called a *Binary Decision Diagram* (BDD) that can, for many commonly occurring Boolean functions, be quite a bit more compact. It is BDDs that we shall now study systematically, beginning with some examples of Boolean functions.

Consider another example to motivate our discussions: the design of a 64-bit adder that adds two 64-bit integers producing a 65-bit result. As pointed out in the example of a comparator, truth-tables are poor representations for almost all functions, including for an adder. For instance, a truth-table for an adder with respect to each of the 65 bits of output will have size (number of rows) equaling  $2^{128}$ . It is clearly impossible to build such truth tables or verify such adders by going through every Boolean combination. We obviously need more efficient methods such as will be presented in this chapter. Specifically, we will introduce BDDs as a data structure conducive to representing Boolean functions compactly, *provided a good variable ordering can be selected*. While this method is not foolproof (*i.e.*, there are Boolean functions for which their BDDs are large), it often works surprisingly well in practice.

## 11.1 BDD Basics

BDDs are directed graphs. They have two types of nodes: ovals and rectangles. Ovals are interior nodes, representing variables and their decodings. One can in fact view the ovals as **2-to-1** muxes. The variable written inside the oval is connected to the “selector” of the mux. There are two leaf nodes, namely 0 and 1 written within rectangles. BDDs also have edges emanating from the ovals:

- red (dotted) edges are “0” edges. They are like the 0 input of the 2-to-1 muxes.
- blue (solid) edges are the “1” edges. They are like the 1 input of the 2-to-1 muxes.
- The output of each interior node (circle) represents a Boolean function realized using 2-to-1 muxes.

Figure 11.1 presents the BDDs for And, Or, and Xor.

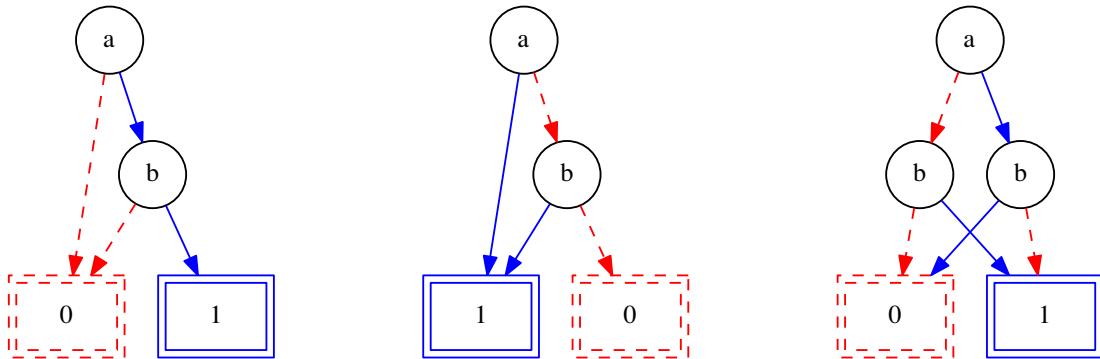


Figure 11.1: Some Common BDDs: And, Or, and Xor (from left to right). Blue is 1 and Red is 0. Memory aid: 0 is the most fundamental invention in math; and that goes with red (i.e., U's color :)

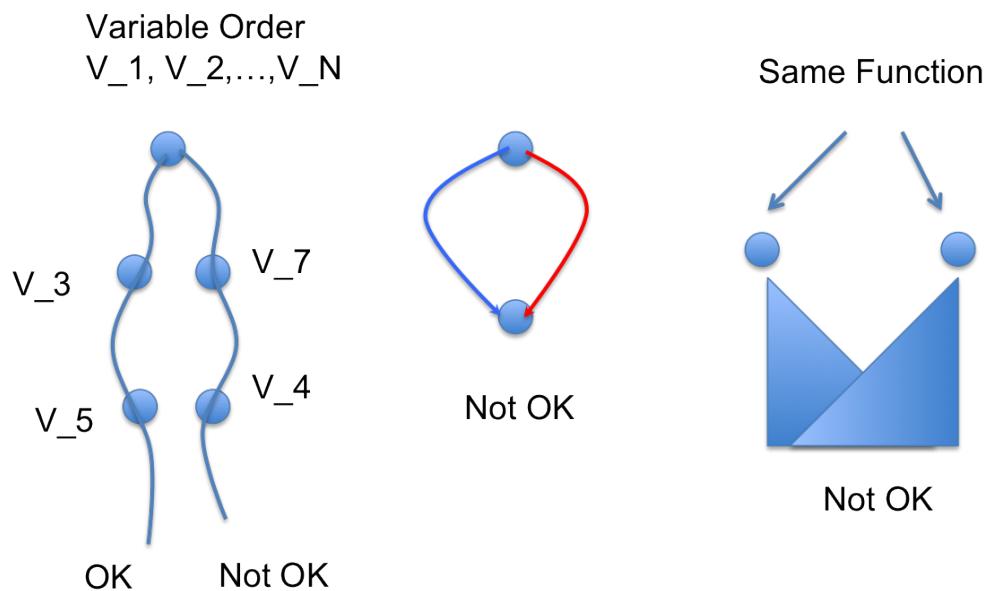


Figure 11.2: Situations to avoid in order to make BDDs Canonical Representations of Boolean functions

### 11.1.1 BDD Guarantees

BDDs that meet three conditions become *canonical* representations of Boolean functions:

- *Variable Ordering*: There is one fixed sequence  $v_1, v_2, \dots, v_N$  ordering the variables. In other words, in any path from the root of the BDD to a leaf (one of the squares), there is no  $v_k$  followed by a  $v_j$  for  $j < k$ . Note that it is okay for some variable  $v_j$  NOT to be on a path
- *No Redundant Decoding*: There is no circle whose outgoing red and blue edges go to the same “child” circle. They must go to different children.
- *No Duplicated Boolean Function*: There are no separately drawn circles representing the same Boolean function.

Figure 11.2 illustrates the situations to avoid so that BDDs may be canonical. Having a canonical representation allows us to compare equivalent BDDs through graph isomorphism. As implemented by most BDD packages, one does not have to carry out graph isomorphism, but rather compare the root node of the BDDs to be hashing into the same bucket (thus making function equality comparison a constant-time operation).

### 11.1.2 BDD-based Comparator for Different Variable Orderings

A comparator can have size *linear* in the number of bits being compared (for a favorable ordering of BDD variables). On the other hand, the BDD can also be exponentially large (for an unfavorable BDD variable ordering). These are illustrated in Figure 11.3.

### 11.1.3 BDDs for Common Circuits

Let us illustrate BDDs constructed through a simple Python script acting on a data file as shown:

```
---Mux41Good.txt begins here and ends where shown below---
Var_Order : s0 s1 i0 i1 i2 i3

Main_Exp : ~s0 & ~s1 & i0 | s0 & ~s1 & i1 | ~s0 & s1 & i2 | s0 & s1 & i3
---end of Mux41Good.txt---

---Mux41Bad.txt begins here and ends where shown below---
```

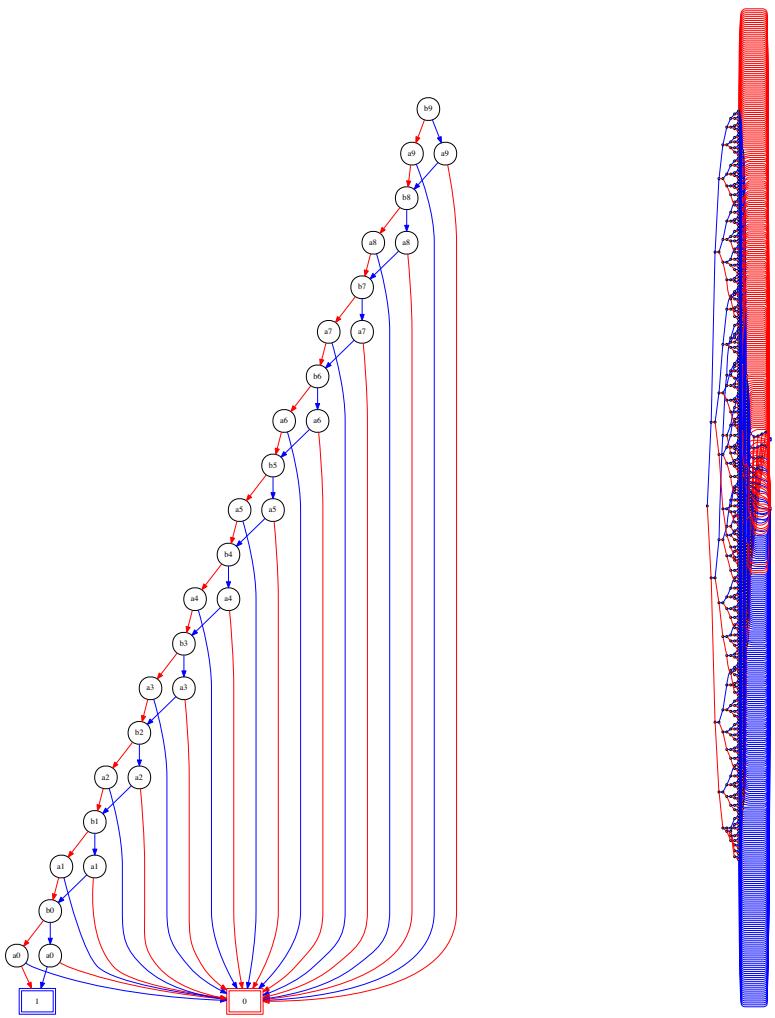


Figure 11.3: Comparator BDD for the Best Variable Ordering and the worst

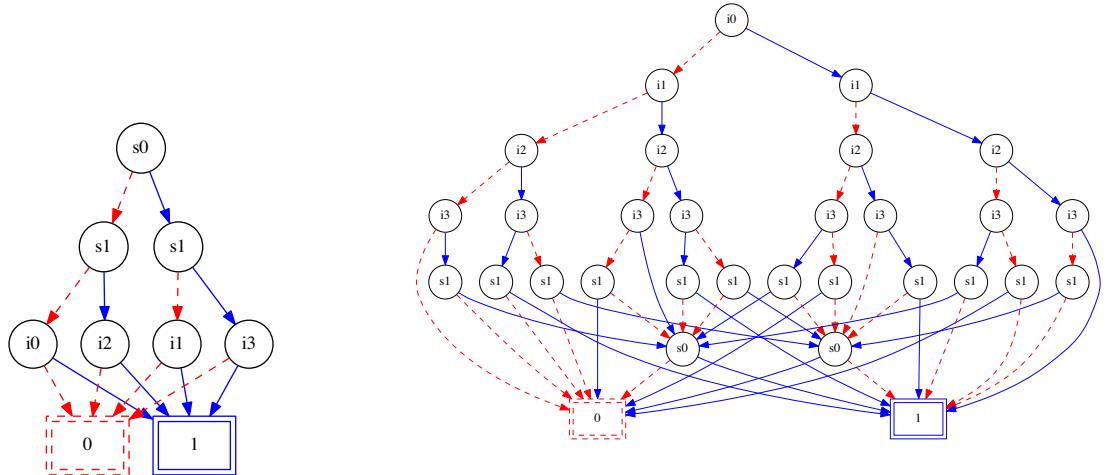


Figure 11.4: A 4-to-1 mux with good variable ordering (left) and a bad ordering (right)

```

Var_Order : i0 i1 i2 i3 s1 s0

Main_Exp : ~s0 & ~s1 & i0 | s0 & ~s1 & i1 | ~s0 & s1 & i2 | s0 & s1 & i3
#---end of Mux41Good.txt---

#---How to process BDD operations in Python---
#! /usr/bin/env python2.7

import sys
import os
sys.path.append("../include/")
sys.path.append("../.../PBL/include/") # Was PyBool, and not PBL
sys.path.append("../.../")
import BDD

if __name__ == "__main__":
    #creating and initializing the BDD
    x = BDD.bdd_init("Mux41Good.txt")
    BDD.build(x)
    dotf = "Mux41Good.dot"

```

```

pdff = "Mux41Good.pdf"
#---
BDD.dot_bdd(x, dotf)
os.system("dot -Tpdf " + dotf + " > " + pdff)
os.system("open -a Preview " + pdff)

#creating and initializing the BDD
x = BDD.bdd_init("Mux41Bad.txt")
BDD.build(x)
dotf = "Mux41Bad.dot"
pdff = "Mux41Bad.pdf"
#---
BDD.dot_bdd(x, dotf)
os.system("dot -Tpdf " + dotf + " > " + pdff)
os.system("open -a Preview " + pdff)

```

To summarize, a “good” variable ordering is one that minimizes the BDD size. It may not be unique (there could be two equally good orderings). Also it depends, in practice, on “how closely related” a collection of variables are in determining the truth value of the function. The “sooner” (after reading the fewest inputs) we can decide the function output, the better.

By studying BDDs in CS 2100, we will have several gains:

- Learn another representation (a canonical representation) for Boolean functions.
- A representation that makes sense to use in practice (exponentially better than truth tables in many important cases)
  - Knuth’s observation: There are  $2^{2^N}$  Boolean functions over  $N$  inputs
  - Most are uninteresting in practice
  - Therefore, there must be a “compressed” representation for those that matter in practice
  - Much like compression of images etc. (many pixels that really don’t matter that much..)
- Will learn how to obtain mux-based circuits straight out of BDDs
- Learn how to read out CNF and DNF representations out of BDDs
- Will be able to do combinatorics pertaining to “unstructured information” with respect to BDDs

### 11.1.4 A Little Bit of History

BDDs are the culmination of a gradual evolution of ideas (1970s, notably Sheldon Akers). In 1986, Randy Bryant introduced the concept of *reduced ordered BDDs* or ROBDDs (this is what we call “BDD”). He invented it in the context of electronic digital circuit simulation/analysis (1986). Since Bryant’s invention, BDDs took off like “wildfire.” They are the basis of many tools. Knuth’s Volume 4a (<http://www-cs-faculty.stanford.edu/~knuth/>) covers BDDs and their use in combinatorics and other applications quite extensively. Knuth calls BDDs “one of the most important of data structures to be introduced in the last 25 years.”

#### Example: Design and debugging of a comparator BDD

Suppose we are given a bit-vector  $[a_2, a_1, a_0]$  of three bits, where  $a_2$  is the MSB and  $a_0$  is the LSB. Similarly, suppose  $[b_2, b_1, b_0]$  is another bit vector. Suppose we want to define the  $<$  relation between these bit vectors. One definition that was attempted recently proved to be **incorrect**; it is:

```
# A < B
# i.e. a2,a1,a0 < b2,b1,b0

Var_Order : a2, b2, a1, b1, a0, b0

Main_Exp : ~a2 & b2 | ~a1 & b1 | ~a0 & b0
```

From Figure 11.5 (left), we can see that this BDD is not correct. Go through all possible paths and see if you can spot errors. One clue: what happens when  $a_2$  is 1 and  $b_2$  is 0? What should it be? (In a correct comparator, the answer must be 0.)

The corrected comparator’s description is below, and its BDD is in Figure 11.5 (right). Notice that we do a full case analysis of how the comparison must go.

```
# A < B
# i.e. a2,a1,a0 < b2,b1,b0

Var_Order : a2, b2, a1, b1, a0, b0

Main_Exp : ~a2 & b2 | (a2 <= b2) & (~a1 & b1 | (a1 <= b1) & ~a0 & b0)
```

In the next section, we shall also see how the CNF and DNF formulae can be obtained from BDDs.

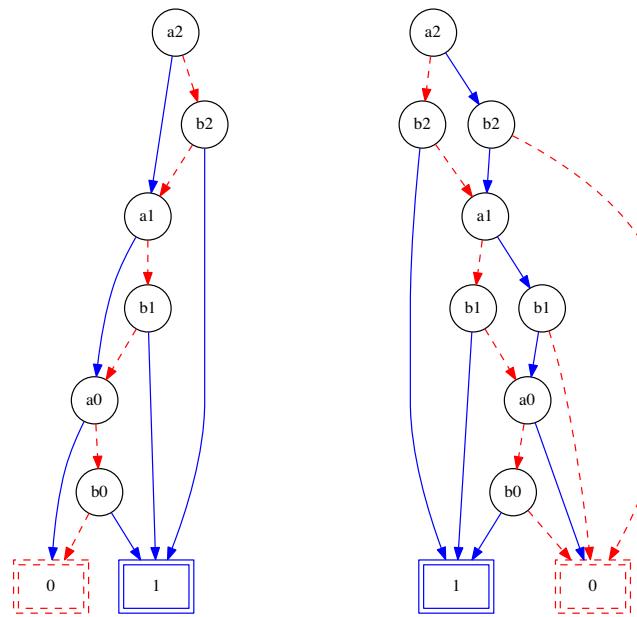


Figure 11.5: Incorrect (left) and Corrected (right) magnitude comparator for the Less-than relation  $<$ . The mistake is for instance in not completely specifying the decodings.

## 11.2 Reading CNF and DNF Off BDDs

In this section, we discuss how we may read conjunctive normal form (CNF; also the same as product of sums, or POS) and disjunctive normal form (DNF; also sum of products, or SOP) expressions equivalent to a BDD-expressed expression (which, by the way, is in an optimized Shannon expansion format). The procedure is as follows:

Algorithm for CNF Derivation from BDD:

- Traverse each path from the root node to the “0” node.
- For each path, take a disjunction (Or) of the **negation of the** literals encountered (negated or un-negated variables are called literals), thus

creating a *clause*. Thus, if you encounter  $\sim b$ , write down  $b$ , and if you encounter  $c$ , write down  $\sim c$ . This extra negation is important, as we are traversing paths to the **0** node, and so when DeMorgan's law is applied for the function output being **1**, these literals will complement again. A clause is a summation of literals, and looks like this:

$$(a \mid \sim b \mid c)$$

- Conjoin (And) the clauses, resulting in a CNF formula that looks like this:

$$(a \mid \sim b \mid c) \ \& \ (e \mid b \mid \sim c)$$

**Illustration of CNF Derivation:** Consider the corrected version of the  $<$  from Figure 11.5. Reading through all the paths leading to 0, we can form a CNF as follows:

```
#-- CNF
Var_Order : a2, b2, a1, b1, a0, b0

Main_Exp : (~a2 | b2) &
           (~a2 | ~b2 | ~a1 | b1) &
           (~a2 | ~b2 | ~a1 | ~b1 | ~a0) &
           (~a2 | ~b2 | ~a1 | ~b1 | a0 | b0) &
           (~a2 | ~b2 | a1 | b1 | ~a0) &
           (~a2 | ~b2 | a1 | b1 | a0 | b0) &
           (~a2 | ~b2 | a1 | b1 | a0 | b0) &
           (a2 | b2 | ~a1 | b1) &
           (a2 | b2 | ~a1 | ~b1 | ~a0) &
           (a2 | b2 | ~a1 | ~b1 | a0 | b0) &
           (a2 | b2 | a1 | b1 | ~a0) &
           (a2 | b2 | a1 | b1 | a0 | b0)
```

Two observations:

- When this CNF is fed through our BDD generator, the result is *the same BDD as in Figure 11.5 (right, corrected)*, thus once again proving that BDDs are canonical representations of Boolean functions. That is, even if we obtained a very different looking Boolean expression and synthesized a BDD, we obtained *the same BDD*. This is one property of BDDs that will prove to be handy time and again.

- In effect, BDDs give us a way to quickly tell whether two Boolean expressions are equivalent (they are equivalent if they result in the same BDD under the same variable ordering).
- It is easily seen that the number of paths—hence the number of clauses—can become exponential. In general, if we have a series of “diamond structures” as in this BDD (branch, reconverge, branch, reconverge, keep doing this in sequence, ...), one can take either the left or the right path of each diamond. This decision can be taken *independently* at every diamond, thus giving us  $2 \times 2 \times \dots \times 2$  paths, thus potentially leading to an exponential number of clauses.

Algorithm for DNF Derivation from BDD:

- Traverse each path from the root node to the “1” node.
- For each path, take a conjunction (And) of the literals encountered, creating a *product term*. Notice that we do not negate the literals: we are taking paths to 1. A product term is a product of literals, and looks like this:

$$(a \ \& \ \sim b \ \& \ c)$$

- Disjoin (Or) the product terms, resulting in a DNF formula that looks like this:

$$(a \ \& \ \sim b \ \& \ c) \ \mid \ (\text{e} \ \& \ \sim b \ \mid \ c)$$

**Illustration of DNF Derivation:** Consider the corrected version of the < from Figure 11.5. Reading through all the paths leading to 1, we can form a DNF as follows:

```
#-- DNF
Var_Order : a2, b2, a1, b1, a0, b0

Main_Exp : (a2 & b2 & a1 & b1 & ~a0 & b0)
           | (a2 & b2 & ~a1 & ~b1 & ~a0 & b0)
           | (a2 & b2 & ~a1 & b1)
           | (~a2 & ~b2 & a1 & b1 & ~a0 & b0)
           | (~a2 & ~b2 & ~a1 & ~b1 & ~a0 & b0)
           | (~a2 & ~b2 & ~a1 & b1)
           | (~a2 & b2)
```

Two observations:

- When this DNF is fed through our BDD generator, the result is *the same BDD as in Figure 11.5 (right, corrected)*, thus once again proving that BDDs are canonical representations of Boolean functions.
- For the same reasons as discussed before, it is easily seen that the number of paths—hence the number of clauses—can become exponential.

## 11.3 Designing using BDD

In this section, we will illustrate the construction and manipulation of BDDs using two examples, one involving map coloring of a simple map, and the other pertaining to traversal in a square grid from coordinate  $(0,0)$  to coordinate  $(3,3)$ .

### 11.3.1 Map Coloring: Canadian Map



Figure 11.6: A Political Map of Canada (courtesy [map-canada.blogspot.com](http://map-canada.blogspot.com))

Consider coloring the provinces of British Columbia (BC) through Quebec (QB)—all lying in a straightline as per Figure 11.7. A BDD for this coloring relation can be generated as per the file below:

File "GoodConstrCanada.txt":

```
Var_Order : aBC, bBC, cBC, aAB, bAB, cAB, aSK, bSK, cSK, \
            aMA, bMA, cMA, aON, bON, cON, aQB, bQB, cQB

BCAB = ~((aBC <=> aAB) & (bBC <=> bAB) & (cBC <=> cAB))

ABSK = ~((aAB <=> aSK) & (bAB <=> bSK) & (cAB <=> cSK))

SKMA = ~((aSK <=> aMA) & (bSK <=> bMA) & (cSK <=> cMA))

MAON = ~((aMA <=> aON) & (bMA <=> bON) & (cMA <=> cON))

ONQB = ~((aON <=> aQB) & (bON <=> bQB) & (cON <=> cQB))

Main_Exp : BCAB & ABSK & SKMA & MAON & ONQB
```

There isn't a very obvious way to pick “good” and “bad” variable orderings for this problem. But again, going by the usual rule of thumb, in the *good ordering*, we put all the variables of one state first; then follow it by the variables of the next state; and so on. In the *bad ordering*, we put all the a variables of all the states, then bring on the b variables of all the states, etc. Here, for instance, is the “bad variable order” (the rest of the details of how the constraints are encoded remains the same).

```
Var_Order : aBC, aAB, aSK, aMA, aON, aQB, bBC, bAB, bSK, \
            bMA, bON, bQB, cBC, cAB, cSK, cMA, cON, cQB
```

In a sense, this results in *closely variables* being situated *far apart* in the variable order. This is when BDDs blow up.

### Coloring along a straightline graph

Why are there 134456 colorings?

- Each province of Canada modeled using 3 bits
- Thus there are 8 colors permissible
- The shape of coloring graph is a straightline:

BC----AB----SK----MA----ON----QB

$$8 * 7 * 7 * 7 * 7 * 7 = 134456$$

- This combinatorics is justified as follows:
  - The first node (British Columbia) can be colored 8 ways
  - The next state (Alberta) can be one of 7 colors
  - The next state (Saskatchewan) can be one of 7 colors
  - Going this way, each of the subsequent states (Manitoba, Ontario, and Quebec) can be one of 7 colors.

### 11.3.2 Walking a Grid using BDDs

Imagine specifying all paths from  $T(0,0)$  to  $T(3,3)$  by taking a right turn or an upward turn, as shown in Figure 11.8. We can express these moves by encoding constraints to go from every  $T(i,j)$  to  $T(i+1,j)$  or  $T(i,j+1)$ . The idea is quite simple, taking  $T_{22}$  as an example:

$$T_{22} = ((a_{22} \& \sim b_{22}) \& T_{32}) \mid ((\sim a_{22} \& b_{22}) \& T_{23})$$

Thus, by expressing a series of equations connecting  $T_{00}$  to  $T_{33}$ , we can generate a BDD that spells out these paths. The constraints are as follows.<sup>1</sup>

```
Var_Order : a00, b00, a10, b10, a20, b20, b30,
           a01, b01, a11, b11, a21, b21, b31,
           a02, b02, a12, b12, a22, b22, b32,
           a03, a13, a23
```

$$T_{33} = 1$$

$$T_{32} = (b_{32} \& T_{33})$$

$$T_{31} = (b_{31} \& T_{32})$$

$$T_{30} = (b_{30} \& T_{31})$$

$$T_{23} = (a_{23} \& T_{33})$$

$$T_{22} = ((a_{22} \& \sim b_{22}) \& T_{32}) \mid ((\sim a_{22} \& b_{22}) \& T_{23})$$

---

<sup>1</sup>We list  $T_{33}$  first and  $T_{00}$  last because of our current tool limitations, to be overcome in our release.

```

T21 = ((a21 & ~b21) & T31) | ((~a21 & b21) & T22)

T20 = ((a20 & ~b20) & T30) | ((~a20 & b20) & T21)

T13 = (a13 & T23)

T12 = ((a12 & ~b12) & T22) | ((~a12 & b12) & T13)

T11 = ((a11 & ~b11) & T21) | ((~a11 & b11) & T12)

T10 = ((a10 & ~b10) & T20) | ((~a10 & b10) & T11)

T03 = (a03 & T13)

T02 = ((a02 & ~b02) & T12) | ((~a02 & b02) & T03)

T01 = ((a01 & ~b01) & T11) | ((~a01 & b01) & T02)

T00 = ((a00 & ~b00) & T10) | ((~a00 & b00) & T01)

Main_Exp : T00

```

Figure 11.9 shows that all the 20 paths are represented in the BDD. However, one mysterious piece of statistics printed by the BDD generator is this:

There are 262144 satisfying instances.

It may be puzzling that 20 BDD paths are reported as 262,144 satisfying instances. The reason is that there are many *don't cares* in the BDD. For instance, each path from (0,0) to (3,3) depends only on six edges or 12 variables, when in fact there are a total of 24 variables. The variables not participating in the path description are essentially being set through all their combinations, giving us the high SAT count.

## 11.4 Solving Puzzles using BDDs

Let us gain an appreciation of how BDDs help us manipulate sentences in mathematical logic. Here is a puzzle by Lewis Carroll.

From the premises

1. Babies are illogical;

2. Nobody is despised who can manage a crocodile;
3. Illogical persons are despised.

Conclude that *Babies cannot manage crocodiles*.

We leave this puzzle for you to solve, and now move on to explaining another puzzle (also by Lewis Carroll). I've shown below how to encode and solve it. From the premises

1. All who neither dance on tight ropes nor eat penny-buns are old.
2. Pigs, that are liable to giddiness, are treated with respect.
3. A wise balloonist takes an umbrella with him.
4. No one ought to lunch in public who looks ridiculous and eats penny-buns.
5. Young creatures, who go up in balloons, are liable to giddiness.
6. Fat creatures, who look ridiculous, may lunch in public, provided that they do not dance on tight ropes.
7. No wise creatures dance on tight ropes, if liable to giddiness.
8. A pig looks ridiculous carrying an umbrella.
9. All who do not dance on tight ropes and who are treated with respect are fat.

Show that *no wise young pigs go up in balloons*.

```
# A puzzle by Lewis Carroll :
#
#Tyler Sorensen
#December 5, 2011

#Input for the second Lewis Carroll problem
#Written in the mark up language for the python BDD manager

#Declare all the variables. Notice how you can use
#more than one line.
Var_Order : eatPennyBuns old young danceTightRopes
Var_Order : pigs respect giddy publicLunch ridiculous
```

```

Var_Order : umbrella fat wise balloon

#These are the Premises
P1 = (~danceTightRopes & ~eatPennyBuns) => old
P2 = (pigs & giddy) => respect
P3 = (ridiculous & eatPennyBuns) => ~publicLunch
P4 = (young & balloon) => giddy
P5 = (wise & balloon) => umbrella
P6 = (fat & ridiculous & ~danceTightRopes) => publicLunch
P7 = (wise & giddy) => ~danceTightRopes
P8 = (pigs & umbrella) => ridiculous
P9 = (~danceTightRopes & respect) => fat

#This is the conclusion
C = (wise & young & pigs) => ~balloon

#This is all the premises conjoined
P_All = P1 & P2 & P3 & P4 & P5 & P6 & P7 & P8 & P9

#The main expression to be considered.
#Main_Exp : P_All & ~C

#To fix, comment out the first Main_Exp and uncomment the
#following lines:

#This is the frame axiom (default condition) needed
Frame = old <=> ~young

Main_Exp : P_All & Frame & ~C

```

**English is Fallible!** We pretty much encoded each English assertion and put it in. This is conjoined with the negated proof goal G. Notice how the “frame axiom” is needed to obtain a correct proof by contradiction (the tool “does not know that” old means not young!) Figure 11.10 (left) shows that we have old and young in the same satisfying paths to the 1 node! Compare this discussion with that in §8.6 on the *frame* axiom.

**Frame Axiom to the Rescue!** To fix the above mistake, we conjoin Frame with P\_All. Figure 11.10 (right) shows that we indeed have a successful proof by contradiction—showing that the proof goal G is proven!

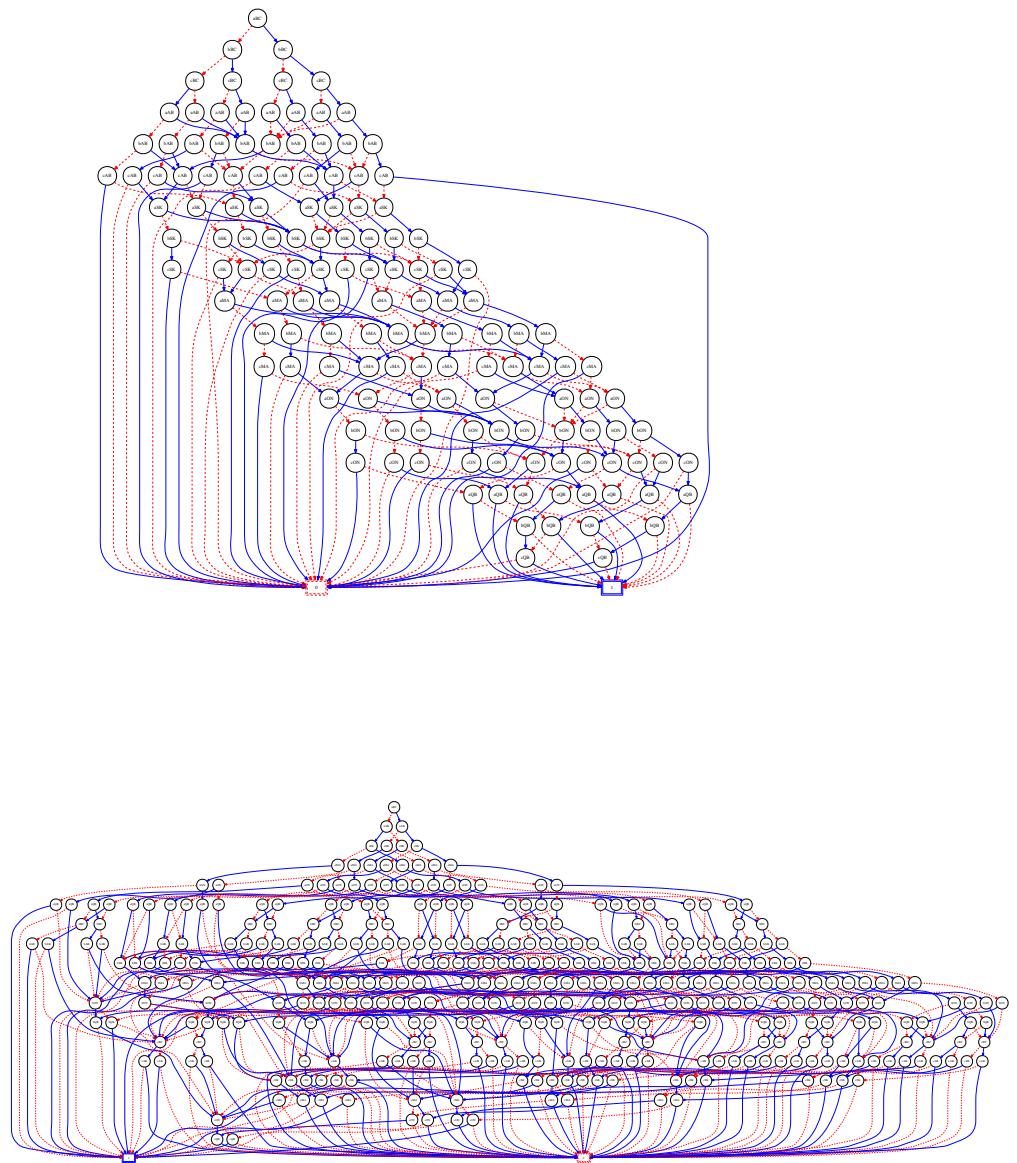


Figure 11.7: Canadian Coloring BDD: Good (top) and Bad (bottom) Variable Orderings. The fact that the BDD has 134,456 satisfying instances (four colorings) is reassuring (possibly no bugs in our BDD package, and  $134456 = 8 \times 7^5$ —also highly reassuring!)

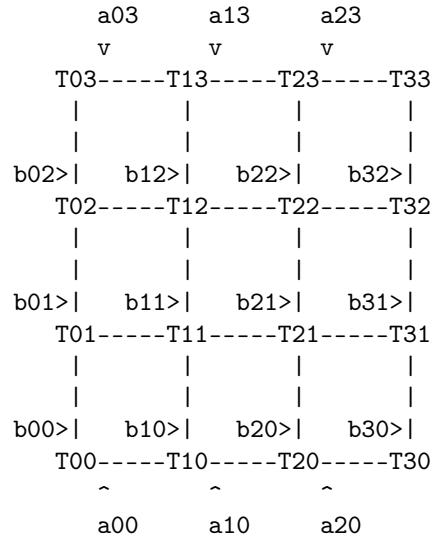


Figure 11.8: Paths in a 3x3 Grid

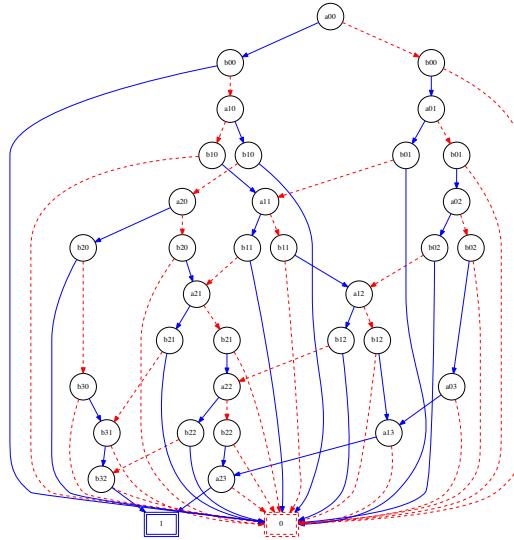


Figure 11.9: Paths in a Square Grid

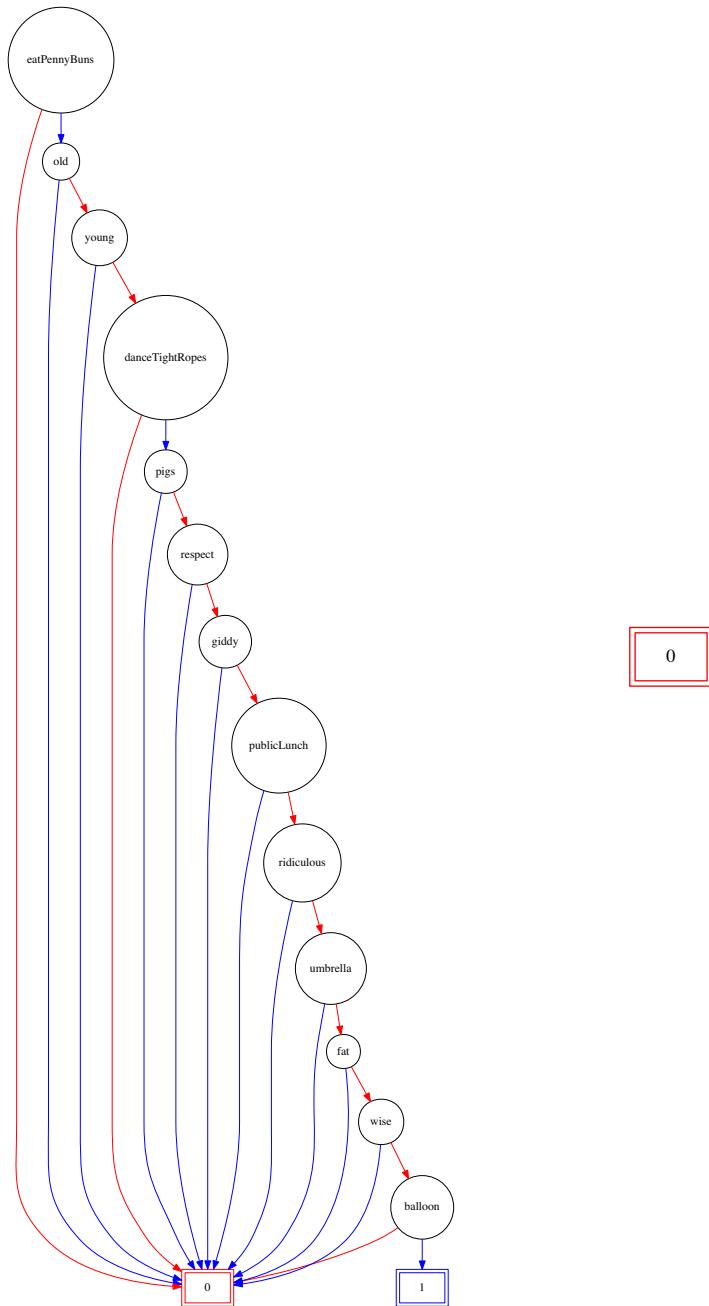


Figure 11.10: Lewis Carroll Puzzle 2's Encoding: Buggy, without the Frame condition (left), and with the Frame condition (and hence, contradiction attained; right)

## 11.5 Review and Labs

You may use the Python BDD package to solve any of these problems.

1. Draw a BDD for the most significant bit (MSB) of the sum output of a bit-serial adder that adds the following bit vectors:

```

[a2 a1 a0] +
cout <-           <- cin
[b2 b1 b0]
-----
[s2 s1 s0]

```

As you can see, the MSB is  $s_2$ . The adder adds bit-vectors  $[a_2 \ a_1 \ a_0]$  and  $[b_2 \ b_1 \ b_0]$ , along with the carry input  $cin$  to produce the sum output  $[s_2 \ s_1 \ s_0]$  and the carry output  $cout$ . Pick the *best variable ordering* where closely related bits are kept together (if there are multiple ‘nearly best’ choices, pick one that gives the most compact BDD (Hint: you may pick variables in some order, but please consider going MSB toward LSB, and also the other direction.)

2. Generate a BDD, picking the *worst* variable order (pick the nearly worst, in case there are several that suggest themselves (again consider going MSB toward LSB, and also the other direction in scanning for variables). How much bigger is the BDD for the worst variable ordering compared to the one for the best ordering?
3. Develop a BDD for the Boolean function  $x$  over input variables  $a, b, c$  such that  $x$  is true when exactly two of the three variables is true (*i.e.*,  $ab$ ,  $bc$ , or  $ac$ ). Obtain a mux-based circuit based on this BDD.
4. Obtain a disjunctive normal form (DNF) formula for the BDD in Question 3. Show that the function you extract is correct, by generating a BDD for it and comparing it with the BDD in Question 3.
5. Obtain a conjunctive normal form (CNF) formula for the BDD in Question 3. Show that the function you extract is correct, by generating a BDD for it and comparing it with the BDD in Question 3.
6. Develop a BDD to node-color a four clique. Allocate the required number of variables for the nodes. Choose a suitable variable order. Now,

find out how many ways there are to color this clique, and compare your answer with that returned by the SAT-count result obtained from the BDD.

# Chapter 12

## Graphs

Graphs are one of the most important of data structures in computer science. From a discrete mathematics point of view, graphs are embodiments of mathematical relations. They allow us to model and solve many real-world problems conveniently. In this chapter, we will take up an important problem that is easy to state and solve it both using brute-force search, and using logical constraint solving. This problem is that of *graph coloring*.

### 12.1 Undirected Graphs, and Graph Coloring

Undirected graphs are pairs  $(\text{Nodes}, \text{Edges})$  where Nodes are a set of atoms (numbers, strings, etc.) and Edges are a set of pairs of atoms (pairs of numbers, strings, etc.). Figure 12.1 presents a graph of the western states of the USA in the mathematically oriented tuple style of  $(\text{Nodes}, \text{Edges})$  as well as in a data structure called an *adjacency list*.

Suppose we are given an undirected graph  $G = (N, E)$ . In case we have two nodes  $x, y \in N$  such that  $E$  contains  $(x, y)$  but not  $(y, x)$ , we will pretend that  $E$  contains  $(y, x)$  also. More formally, we will assume that given any undirected graph  $G$ , its edge relation has been subject to the *symmetric closure* step.

#### 12.1.1 Map Coloring

This question pertaining to map drawing has puzzled humans for a very long time: *how many colors are necessary to draw a political map (say states)*

```

USWest = (USWestNodes, USWestEdges)

USWestNodes = {WA, OR, ID, CA, NV, AZ, MT, WY, UT, CO, NM}

USWestEdges = {(ID, WA), (OR, WA), (OR, ID), (CA, OR), (NV, OR), (CA, NV), (AZ, NV),
                (CA, AZ), (MT, ID), (WY, ID), (WY, MT), (UT, ID), (UT, NV), (ID, NV),
                (UT, WY), (CO, WY), (UT, CO), (AZ, UT), (NM, AZ), (CO, NM)}

#--- An Adjacency List representation of the same.
#--- The function ChkGraphIsSymmetric explains how this representation
#--- is supposed to be used
#
#
from z3 import *

grWest = { 'WA' : ['ID', 'OR'],
           'OR' : ['WA', 'ID', 'CA', 'NV'],
           'CA' : ['OR', 'NV', 'AZ'],
           'ID' : ['WA', 'OR', 'NV', 'MT', 'WY', 'UT'],
           'NV' : ['OR', 'CA', 'AZ', 'UT', 'ID'],
           'AZ' : ['CA', 'NV', 'UT', 'NM'],
           'MT' : ['ID', 'WY', 'ND', 'SD'],
           'WY' : ['MT', 'ID', 'UT', 'CO', 'SD', 'NE'],
           'UT' : ['ID', 'NV', 'AZ', 'CO', 'WY'],
           'CO' : ['WY', 'UT', 'NM', 'NE', 'KS', 'OK'],
           'NM' : ['AZ', 'CO', 'OK', 'TX'],
           'ND' : ['MT', 'SD'],
           'SD' : ['ND', 'MT', 'WY', 'NE'],
           'NE' : ['SD', 'WY', 'CO', 'KS'],
           'KS' : ['NE', 'CO', 'OK'],
           'OK' : ['KS', 'CO', 'NM', 'TX'],
           'TX' : ['OK', 'NM']
       }

def ChkGraphIsSymmetric(gr):
    """ Checks if gr is symmetric; i.e. for all S1 -> S2, i.e. connected to S2,
    we also have S2 -> S1. """
    for k in gr.keys():
        for j_adj_to_k in gr[k]:
            # print "State " + k + " -> " + j_adj_to_k
            assert(k in gr[j_adj_to_k]), (
                """Error: node """ + str(k) + """ connected to """ + str(j_adj_to_k) +
                """ but not the other way.""" )

```

ChkGraphIsSymmetric(grWest)

Figure 12.1: Western States of the USA: As Classical Undirected Graphs (N, E), and as an Adjacency List. We check for symmetry to catch typos (the user is supposed to provide an adjacency list that is symmetric).

such that no two adjacent political regions (states) are colored the same? The history of the problem is well described in Wikipedia, including how the problem was finally solved in 1976 by Appel and Haken through a laborious computer-assisted proof [17]. Recently, the last bit of doubts were removed from this proof by Georges Gonthier [5] who works for Microsoft Research.

The problem is rather easy to state precisely:

- We are given an arbitrary map
- No two adjacent states/countries ought to have the same color
- “Adjacent” is defined (and illustrated) as follows:
  - If two regions are adjacent, they must share a point that is not shared by a third region.
  - Thus, considering the famous “Four Corners” region of Utah, Utah is not adjacent to New Mexico, and also Arizona is not adjacent to Colorado [17]. Therefore, in the four corners region, there are only four adjacencies: (UT, CO), (CO, NM), (NM, AZ), and (AZ, UT).
  - If there is a point shared by Arizona and Colorado other than at the four corner’s region, then (and then only would) AZ and CO be considered adjacent. Such a point can be arranged say by giving Colorado a strip of land through Utah, directly going through and touching Arizona.

## 12.2 Brute-force Solving of Graph Coloring

What are we waiting for? Let us roll our own brute-force solution to the map coloring, as given in Figure 12.2.

This solution is fine up to a certain number of nodes. If you add some more states (e.g., ND, SD, KS, NE, OK, and TX), the explicit approach grinds to a halt. Luckily for us, Z3py’s constraint-based approach can handle significantly larger problems much more quickly. In later chapters, we will introduce other methods for solving this problem, including using *Binary Decision Diagrams*.

## 12.3 Constraint-based Solving

Ah-ha, now you are talking! Clearly, it is far more elegant to be solving the map coloring problem using constraints. The constraint based scheduling is straightforward, thanks to the versatility of z3py. We first for a list of keys

```

Ncol = 4

for WA in range(Ncol):
    for OR in range(Ncol):
        for CA in range(Ncol):
            for ID in range(Ncol):
                for NV in range(Ncol):
                    for AZ in range(Ncol):
                        for MT in range(Ncol):
                            for WY in range(Ncol):
                                for UT in range(Ncol):
                                    for CO in range(Ncol):
                                        for NM in range(Ncol):
                                            if (
                                                (ID != WA) & (OR != WA) & (OR != ID) &
                                                (CA != OR) & (NV != OR) & (CA != NV) &
                                                (AZ != NV) & (CA != AZ) & (MT != ID) &
                                                (WY != ID) & (WY != MT) & (UT != ID) &
                                                (UT != NV) & (ID != NV) & (UT != WY) &
                                                (CO != WY) & (UT != CO) & (AZ != UT) &
                                                (NM != AZ) & (CO != NM)
                                            ):
                                                print (WA, OR, CA, ID, NV, AZ, MT, WY, UT,
                                                       CO, NM)
                                            exit()

--- In case all solutions are of interest
Ncol = 4
USWest = [ (WA, OR, CA, ID, NV, AZ, MT, WY, UT, CO, NM)
           for WA in range(Ncol) for OR in range(Ncol)
           for CA in range(Ncol) for ID in range(Ncol)
           for NV in range(Ncol) for AZ in range(Ncol)
           for MT in range(Ncol) for WY in range(Ncol)
           for UT in range(Ncol) for CO in range(Ncol)
           for NM in range(Ncol)
           if
               (ID != WA) & (OR != WA) & (OR != ID) &
               (CA != OR) & (NV != OR) & (CA != NV) &
               (AZ != NV) & (CA != AZ) & (MT != ID) &
               (WY != ID) & (WY != MT) & (UT != ID) &
               (UT != NV) & (ID != NV) & (UT != WY) &
               (CO != WY) & (UT != CO) & (AZ != UT) &
               (NM != AZ) & (CO != NM)
           ]
print "Length of USWest, i.e. # colorings = " + str(len(USWest))
print "One coloring is " + str(USWest[0])

```

Figure 12.2: Brute-force calculation of *one* solution to the map coloring problem of the western states of the US. A list comprehension in case *all* solutions are of interest, as well as the *number* of such solutions.

```

from z3 import *
grWest = { 'AZ' : ['CA', 'NV', 'UT', 'NM'],
           'MT' : ['ID', 'WY', 'ND', 'SD'],
           'WY' : ['MT', 'ID', 'UT', 'CO', 'SD', 'NE'],
           'UT' : ['ID', 'NV', 'AZ', 'CO', 'WY'],
           'ID' : ['WA', 'OR', 'NV', 'MT', 'WY', 'UT'],
           'CO' : ['WY', 'UT', 'NM', 'NE', 'KS', 'OK'],
           'WA' : ['ID', 'OR']           'OR' : ['WA', 'ID', 'CA', 'NV'],
           'CA' : ['OR', 'NV', 'AZ'],     'NV' : ['OR', 'CA', 'AZ', 'UT', 'ID'],
           'NM' : ['AZ', 'CO', 'OK', 'TX'], 'ND' : ['MT', 'SD'],
           'SD' : ['ND', 'MT', 'WY', 'NE'], 'NE' : ['SD', 'WY', 'CO', 'KS'],
           'KS' : ['NE', 'CO', 'OK'],      'OK' : ['KS', 'CO', 'NM', 'TX'],
           'TX' : ['OK', 'NM']
}

ChkGraphIsSymmetric(grWest)

# ['CO', 'WA', 'CA', 'AZ', 'ID', 'WY', 'NM', 'UT', 'MT', 'OR', 'NV']
stLstWest = grWest.keys() # [ 'WA', 'OR', ...]

NStates = len(stLstWest)

# [CO, WA, CA, AZ, ID, WY, NM, UT, MT, OR, NV]
LInts = Ints(stLstWest)    # Creates constraint vars ; L[0] is int var WA, etc.

# {'CO': CO, 'NM': NM, 'WA': WA, 'UT': UT, 'CA': CA, 'OR': OR, 'NV': NV, ...}
stNamIntDict = { stLstWest[i] : LInts[i] for i in range(NStates) }

CMax = 3

s = Solver()

for state in stLstWest:
    # Default color range constraints
    s.add(stNamIntDict[state] >= 0)
    s.add(stNamIntDict[state] <= CMax)
    # These are the graph coloring constraints
    s.add(And([stNamIntDict[state] != stNamIntDict[adjState]
              for adjState in grWest[state]]))

rslt = s.check()
if (rslt == sat):
    print "The coloring of states is"
    print(s.model())
else:
    print "There is no " + str(CMax) + " coloring for the given graph."

```

Figure 12.3: A Constraint-based Solution of Map Coloring using Z3py

`stLstWest`, and also determine the number of states `NStates`. We then form a list of constraint integers in `LInts`. We form a dictionary `stNamIntDict` that associates the string name of a state to its integer constraint variable. We then seek a coloring in the range `0..CMax`. The crucial loop that adds constraints is

```
for state in stLstWest:
    # Default color range constraints
    s.add(stNamIntDict[state] >= 0)
    s.add(stNamIntDict[state] <= CMax)
    # These are the graph coloring constraints
    s.add(And([stNamIntDict[state] != stNamIntDict[adjState]
              for adjState in grWest[state]]))
```

This loop begins by adding constraints for the integer variables: each is constrained to be in the range (*closed interval*)  $[0, CMax]$ . Thereafter, we add an inequality constraint relating every pair of adjacent nodes.

```
s.add(And([stNamIntDict[state] != stNamIntDict[adjState]
          for adjState in grWest[state]]))
```

The brute-force method of §12.2 performs nearly the same as the constraint-based version for small graphs. However, for larger graphs, there is no comparison: the constraint-based version significantly outperforms the brute-force version.

## 12.4 Applications: Scheduling and Coloring

### 12.4.1 Application of Coloring: Committee Scheduling

This is an example from [14]. Consider  $N$  committees with some committees having the same member. Suppose we have to schedule hour-long meetings for all the committees such that two committees are allowed to concurrently meet only if they have no common person. What is the minimum number of hours necessary to hold all the committee meetings one after the other?

This problem can be solved by modeling each committee as a graph node. Two nodes (committees) have an edge connecting them if they have a committee member in common. Therefore by modeling the “committee graph” and finding its chromatic number—minimum number of colors necessary to color the graph.<sup>1</sup> The minimum number of crayons necessary to color the nodes such that two connected nodes do not have the same color.

**Example 1:** Consider the following committees that are to schedule meetings of one-hour duration such that no committee member is asked to be in two meetings during the same hour. What is the minimum number of hours necessary to hold all the meetings?

- C1: A, B, C, D
- C2: B
- C3: C, E, A
- C4: F, H
- In this example, there will be the following edges: (C1, C2) (this is because member B cannot be in two meetings at the same time) and (C1, C3) (this is because members A and C cannot be in two meetings at the same time).
- There are no other committee member conflicts (being asked to be in two separate meetings during the same hour). Thus, the coloring graph we have is:

$$(N = \{C1, C2, C3, C4\}, E = \{(C1, C2), (C1, C3)\})$$

- The chromatic number of this graph is 2. This means that the committees can finish meeting in 2 hours.
- We can schedule as follows, thus assigning C1 the time period of 0, and all other nodes the time period if 1.

C1 ; (C2 || C3 || C4)

**Example 2:** Consider the committees

- C1: TimbaktuTrekking (TT), SleepResearch (SR), TofuSnarfing (TS)
- C2: SleepResearch (SR), MohawkGrooming (MG)
- C3: MohawkGrooming (MG), UnicycleRiding (UR)
- C4: UnicycleRiding (UR), SomnambulismMarathon (SM), BuffetCrashing (BC)
- C5: BuffetCrashing (BC), BungeeJumping (BJ), TofuSnarfing (TS)

The coloring graph is

$$(N = \{C1, C2, C3, C4, C5\}, E = \{(C1, C2), (C2, C3), (C3, C4), (C4, C5), (C5, C1)\})$$

This graph of an *odd cycle length* needs at least three colors. This is true of all graphs that are odd-length cycles.

**Example 3:** The makeup of C5 was changed, and one more committee (C6) was formed, so that those who snarf tofus do so after a bungee-jump session:

- C5: BuffetCrashing (BC), BungeeJumping (BJ)
- C6: BungeeJumping (BJ), TofuSnarfing (TS)

*Exercise:* What is the graph and its chromatic number now?

### 12.4.2 Application of Coloring: Register Allocation

Time flows downwards.

A    B    C

```

|           Use R1
|   |       Use R1, R2
|   |   |   Use R1, R2, R3
|       |   Use R1, R2, R3
|   |       Use R2, R3
|           Use R2

```

```

|   |       Use R1, R2
|   |       Use R1, R2
|           Use R1
|       |   Use R1, R3
|       |   Use R1, R3
|           Use R1

```

The overlap graph as an adjacency list:

```
{
A1: [B1, C1],
B1: [A1, C1],
C1: [A1, B1, B2],
B2: [C1],
A2: [B3, C2],
B3: [A2],
C2: [A2]
}
```

Figure 12.4: Register coloring and the overlap graph

This is also an example from [14] and further explained in [12]. Consider a program where multiple variables are used, and the value of each variable is to be held in registers as far as possible, for more efficient processing.

- that distinct registers are available for each time slice over which multiple values are being computed.
- only the smallest number of registers are employed.

For example, see Figure 12.4 where we have variable A being subject to updates for a while, then after a brief pause, we have A being subject to updates again. The same is true of B and C. A typical algorithm is to load A into a register, operate on the register, and then spill the register at the end of the first period of use of A. Then we repeat this again for the next period of use of A, and so on. The same is true of variables B and C also.

If we want to minimize the number of registers used, we can begin with a program graph and model each variable using multiple intervals (capturing where all the variable is being employed). One can now employ a register to hold the value of each variable over each interval of time. By computing the “overlap graph” of such interval diagrams, we can ensure these points:

For the graph model of Figure 12.4, we will employ 7 nodes and we will employ five bidirectional edges between these nodes. The reasoning is as follows:

- We model entire intervals as graph nodes.
- We look for points of *maximal* overlaps between collections of intervals. This maximal degree of overlap must exist at some point in time. For the given example, this occurs at (A1, B1), (B1, C1), and (A1, C1). This immediately forces us to use three registers R1, R2, and R3.
- The remaining edges introduced are (B2, C1), (A2, B3), and (A2, C2).
- The resulting graph is also shown (as an adjacency list), and has a chromatic number of 3.
- Initially, register R1 is active.
- Because of the edges (A1, B1) and (B1, C1), three colors (registers) are essential during the time period when variables A, B, and C are active. (B2, C1), (A2, B3), and (A2, C2).



# Chapter 13

## Basic First Order Logic

Suppose someone asks you to define an infinite set of integers using only mathematical logic. You might think of the following definition:

- A set  $S$  is infinite exactly when it has no maximal element.
- That is, for every element  $x \in S$ , there is a  $y \in S$  such that  $y > x$ .

You can see that we have already begun using a different “language” to describe these ideas:

- We have started using constructions such as: “For Every” and “There Exists.” These are called *quantifiers*.
- We have started talking about things such as  $y > x$ . These are *predicates*.

**First order Logic or Predicate Logic** The ideas above are what *first order predicate logic* introduces. We sometimes use “FOL” for first order logic. Let us now gain more familiarity with these ideas.

### 13.1 Universe, Individuals, and Predicates

Look at the book **Logic Programming with Prolog** by **Max Bramer** made available to you. You should be reading the Introduction beginning Page 5 and also Chapters 1, 2, and 8 of this book. We are using Prolog to illustrate predicates and first order logic (FOL) which may seem strange at first.

**Universe and Individuals** A *universe* is assumed in any discussion involving first order logic. A universe is a collection of values called *individuals*.

als. Take a look at the discussion on Page 6 of the introduction of Bramer's book. It talks about fido, rover, henry, felix, michael and jane. These are the individuals under discussion.

**Predicates** Take a look at the same example. There are three predicates defined:

- dog is a predicate defined to be true of fido, rover and henry.
- cat is a predicate defined to be true of felix, michael and jane.
- animal is defined in terms of dog. (Even if cats are animals, we are at liberty to define predicates the way we want.) The definition must be read “animal(X)” (or X is an animal) **if** dog(X) (or X is a dog). The “:-” is to be read as **if**.

**Quantifiers** Prolog is a practical programming language, and so it only makes a restricted use of quantifiers. When fully expanded, the statement `animal(X) :- dog(X)` must be read as:

For all  $X$ ,  $\text{dog}(X)$  implies  $\text{animal}(X)$

or in mathematical notation

$$\forall x : \text{dog}(x) \Rightarrow \text{animal}(x).$$

**Specializing the domain of Quantifications** When we do not explicitly specify the domain over which the quantified variables such as  $\forall x :$  (forall  $x$ ) and  $\exists x :$  (there exists  $x$  or for some  $x$ ) range, it is tacitly assumed that they range over the *whole universe*.

However, often, we will write  $\forall x \in D : \dots$  and  $\exists x \in D : \dots$  where the domain  $D$  is specified. In this case,  $D$  is a subset of the universe. We will clarify these usages in the sections below.

## 13.2 Individuals, Quantifiers over them

The term *First order logic* exists because propositional logic (Boolean logic) is sometimes called *zeroth order* logic. There are also *Second* and *Higher Order* logics. The main difference between first and higher (“higher than first”) order logics is this:

- In First order logic, the universe is made of simple kinds of individuals. They are only “countably infinite” at most. Predicates in FOL are relations over these “simple individuals.”
- In higher order logics, the universe as well as the predicates can be over other functions and predicates (equivalently, the universe can be uncountably infinite).

We now set aside all higher order logics, and focus on FOL in the rest of this chapter.

**FOL** In first-order logic, we have not only Boolean (or propositional) variables, but also *individual* variables. The term “individuals” is used to denote numbers, strings, trees, doctors, barbers, etc. (whatever be the objects in the domain of interest be). In this setting, we can define the notion of *quantifiers*. There are two main quantifiers of interest:

- Forall
- There exists

Here are usages of these quantifiers:

- “All men are mortal”: Forall  $x$  that are men,  $x$  is mortal.
- “All squares are rectangles”: For all  $s$  that are squares, they are always rectangles.
- “Some rectangles are squares”: There exist rectangles  $r$  that are squares also.

- “Some rectangles are triangles”: Well, this can be *said* in first-order logic, but when it comes to evaluating the truth, these sentences will be deemed to be **false**.
- “Forall  $x$ ,  $x$  equals 0”: Again, it can be said, but is false.
- “All rectangles are squares”: False again, because while *some* rectangles are squares, not all of them are.

Let the *individuals* of interest be  $\{i_0, i_1, i_2, \dots\}$  (say,  $i_0, i_1$  are numbers or strings). Then, when one writes forall and exists using standard mathematical notation, they have the following meanings:

- $\forall x : P(x) =$
- $\wedge P(i_0)$
- $\wedge P(i_1)$
- $\wedge P(i_2)$

$\wedge P(i3)$ 

...

Similarly,

- $\exists x : P(x) =$
- $\vee P(i0)$
- $\vee P(i1)$
- $\vee P(i2)$
- $\vee P(i3)$
- ...

There is no need that the set of individuals is finite, although they often are. One can notice that quantifiers are nothing but repeated conjunction (written as “forall” or  $\forall$ ) and repeated disjunction (written as “exists” or  $\exists$ ). In fact, there is a very strong analogy:

- $\forall x : P(x)$  is very similar to  $\prod_{i=0}^N s(i)$ . Forall is “repeated and”—much like  $\Pi$  is “repeated multiplication.”
- $\exists x : P(x)$  is very similar to  $\sum_{i=0}^N s(i)$ . Exists is “repeated or”—much like  $\Sigma$  is “repeated addition.”

### Being explicit about the individuals

Having the notion of individuals floating around in the background makes everyone nervous, confused, or at best uncomfortable. To avoid this uncertainty, in the next section (and in much of the rest of the chapter), we will explicitly introduce the individuals by introducing two handier notations:

$$\forall x \in D : P(x), \text{ and } \exists x \in D : P(x)$$

where the *domain of quantification*,  $D$ , is introduced explicitly. Often  $D$  is the set of individuals themselves, and for example is *Nat*, *Rectangles*, *People*, etc.

### 13.3 Quantifiers over a Domain

Here is how these operators are defined, taking *Nat*, the set of natural numbers (*i.e.*,  $\{0, 1, 2, \dots\}$  as the domain of quantification):

- $\forall x \in \text{Nat} : P(x) =$

$\wedge P(0)$   
 $\wedge P(1)$   
 $\wedge P(2)$   
 $\wedge P(3)$   
... (to infinity)

Similarly,

- $\exists x \in Nat : P(x) =$
- $\vee P(0)$   
 $\vee P(1)$   
 $\vee P(2)$   
 $\vee P(3)$   
... (to infinity)

Let us revisit the assertion that  $S$ , a subset of natural numbers, is infinite. We can write this assertion using the standard notation for quantification as

$$\exists S \subseteq Nat : \forall x \in S : \exists y \in S : y > x$$

We first employ two abbreviations:

$\forall x \in S : p(x)$  is an abbrev. for  $\forall x : (x \in S \Rightarrow p(x))$ .

$\exists y \in S : q(y)$  is an abbrev. for  $\exists y : (y \in S \wedge q(y))$ .

Using these abbreviations, we can write:

$$\exists S \subseteq Nat : \forall x \in S : \exists y \in S : y > x \equiv$$

$$\exists S : (S \subseteq Nat) \wedge (\forall x : (x \in S \Rightarrow (\exists y : (y \in S \wedge (y > x))))).$$

We can write the main part of the formula as follows:

$$(\forall x : (x \in S \Rightarrow (\exists y : (y \in S \wedge (y > x))))).$$

This formula is clearly true.

- Expanding on  $\forall x$ , we have:

$$\wedge (0 \in S \Rightarrow (\exists y : (y \in S \wedge (y > 0))))$$

$$\wedge (1 \in S \Rightarrow (\exists y : (y \in S \wedge (y > 1))))$$

$$\wedge (2 \in S \Rightarrow (\exists y : (y \in S \wedge (y > 2))))$$

....

- We can also expand the inner  $\exists$ , resulting in an overall expansion of this form:

$$\bullet \quad \wedge (x : 0 \in S \Rightarrow$$

$$\vee (y : 0 \in S \wedge y : 0 > x : 0)$$

$$\vee (y : 1 \in S \wedge y : 1 > x : 0)$$

...

$$\wedge (x : 1 \in S \Rightarrow$$

$$\vee (y : 0 \in S \wedge y : 0 > x : 1)$$

$$\vee (y : 1 \in S \wedge y : 1 > x : 1)$$

$$\vee (y : 2 \in S \wedge y : 2 > x : 1)$$

...

$\wedge$  ...

In the above expansion, we annotate the values with  $x :$  or  $y :$  to indicate which quantifier these values come from. As can be seen, this expression is **true** because for any choice of  $x$  (written before the implication operator  $\Rightarrow$ ), there is a choice of  $y$  (coming in the disjunctive chain) which exceeds this  $x$ .

- To see that all this is working out even more clearly, we can negate the very first translation:

$$\neg(\exists S : (S \subseteq Nat) \wedge (\forall x : (x \in S \Rightarrow (\exists y : (y \in S \wedge (y > x)))))) \equiv$$

$$(\forall S : \neg(S \subseteq Nat) \vee \neg(\forall x : (x \in S \Rightarrow (\exists y : (y \in S \wedge (y > x)))))) \equiv$$

$$(\forall S : (S \subseteq Nat) \Rightarrow \neg(\forall x : (x \in S \Rightarrow (\exists y : (y \in S \wedge (y > x)))))) \equiv$$

$$(\forall S : (S \subseteq Nat) \Rightarrow (\exists x : (x \in S \wedge \neg(\exists y : (y \in S \wedge (y > x)))))) \equiv$$

$(\forall S : (S \subseteq Nat) \Rightarrow (\exists x : (x \in S \wedge (\forall y : (\neg(y \in S) \vee \neg(y > x)))))) \equiv$   
 $(\forall S : (S \subseteq Nat) \Rightarrow (\exists x : (x \in S \wedge (\forall y : (y \in S \Rightarrow (y \leq x))))))$ , which can be abbrev. as  
 $\forall S \subseteq Nat : \exists x \in S : \forall y \in S : y \leq x$   
which says “every  $S \subseteq Nat$  is finite”—clearly **false**.

### 13.3.1 A Helpful Example to Understand Quantifiers

We now present the following example to explain how to negate quantified formulae in practice. Consider this assertion (likely a rumor): A Reticulated Giraffe  $G$  is a male if for all animals weighing more than  $G$  kilograms one can find a cluster of skin patches with  $a$  patches below 10 cm in perimeter,  $b$  with perimeter 20, and  $c$  above 30 on one of its sides, such that any cluster with a different perimeter has no more hair than these standard patches.

How do we disprove this conjecture? In other words, how do we show that the original formula is true?

- Find *a giraffe* weighing more than  $G$  kilos
- Look at *both* its sides
- Look at *all* the patches matching the stated criteria
- Find *one* cluster with a different perimeter than the standard patches
- Show that this cluster has more hair than the standard patches

Notice how we flipped the quantifiers around:

- “on one of its sides” became “look at both its sides”
- “find a cluster of patches” became “look at all patches”
- “any cluster with a different perimeter” became “find one with a different perimeter”
- “has no more hair” became “has more hair.”

This again shows how one can employ quantifiers quite meaningfully in everyday contexts.<sup>1</sup>

### 13.3.2 Illustration on Fermat’s Last Theorem

To obtain some practice on negating quantified formulae, let us In number theory, Fermat’s Last Theorem (sometimes called Fermat’s conjecture, especially in older texts) states that no three positive integers  $a$ ,  $b$ , and  $c$  can

---

<sup>1</sup>Keep your hair on; the Giraffe thing was, after all, a rumor!

satisfy the equation  $a^n + b^n = c^n$  for any integer value of  $n$  greater than two; see [http://en.wikipedia.org/wiki/Fermat's\\_Last\\_Theorem](http://en.wikipedia.org/wiki/Fermat's_Last_Theorem).

$$\forall a, b, c, n : (((a, b, c > 0) \wedge (n \geq 3)) \Rightarrow (a^n + b^n) \neq c^n)$$

Here are literal quotes from [http://en.wikipedia.org/wiki/Fermat's\\_Last\\_Theorem](http://en.wikipedia.org/wiki/Fermat's_Last_Theorem), focusing on the human element of scientific research: This theorem was first conjectured by Pierre de Fermat in 1637, famously in the margin of a copy of Arithmetica where he claimed he had a proof that was too large to fit in the margin. No successful proof was published until 1995 despite the efforts of countless mathematicians during the 358 intervening years. ... It is among the most famous theorems in the history of mathematics and prior to its 1995 proof was in the Guinness Book of World Records for "most difficult mathematical problems."

... Upon hearing of Ribet's proof, Andrew Wiles decided to commit himself to ... proving a special case of the modularity theorem (then known as the Taniyama-Shimura conjecture) for semistable elliptic curves. (It was Wiles's childhood dream to solve the Fermat's Last Theorem! See <http://www.pbs.org/wgbh/nova/physics/andrew-wiles-fermat.html> for details.) Wiles worked on that task for six years in almost complete secrecy. ... By mid-1993, Wiles was sufficiently confident of his results that he presented them in three lectures delivered on June 21, 1993 at the Isaac Newton Institute for Mathematical Sciences. ... However, it soon became apparent that Wiles's initial proof was incorrect. ... Wiles and his former student Richard Taylor spent almost a year trying to repair the proof, without success. On 19 September 1994, Wiles had a flash of insight that the proof could be saved by returning to his original Horizontal Iwasawa theory approach, ... The two papers were vetted and published as the entirety of the May 1995 issue of the Annals of Mathematics. ... last step in proving Fermat's Last Theorem, 358 years after it was conjectured.

Suppose Fermat's Last Theorem were false; then, the negation of

$$\forall a, b, c, n : (((a, b, c > 0) \wedge (n \geq 3)) \Rightarrow (a^n + b^n) \neq c^n)$$

would have been true; i.e.,

$$\exists a, b, c, n : ((a, b, c > 0) \wedge (n \geq 3) \wedge ((a^n + b^n) = c^n))$$

Unfortunately, try and try again as much as you wish, you will *never* find such a set of numbers  $(a, b, c, n)$  such that this equation holds.

## 13.4 Nonplussed Barbers, Non-quack Doctors

### 13.4.1 Keep your hair on!

Consider the assertions

*Every barber shaves everyone who does not shave himself. No barber shaves someone who shaves himself. Prove that there exists no barber.*

We can understand how to tease out the underlying first-order assertions as follows. First of all, fix on *individuals* being *people* (set  $P$ ). Barbers are people, and shavers are people<sup>2</sup> So when we use  $\forall x : P(x)$  or  $\exists x : P(x)$  in this domain of discourse,  $x$  ranges over *people*  $P$ .

- Suppose there is a set of barbers  $b = \{b_0, b_1, b_2, \dots\}$ .
- Suppose there is a set  $sh$  of  $\langle x, y \rangle$  such that  $x$  shaves  $y$ . Notice that  $x = y$  is allowed<sup>3</sup>

Then, we can render the first assertion as follows:

- Focus on the part ...everyone who does not shave himself:

$$\forall p : \neg sh(p, p)$$

- Focus on the part every barber...:

$$\forall q : b(q)$$

- Put it all together:

$$\forall q : (b(q) \Rightarrow (\forall p : \neg sh(p, p) \Rightarrow sh(q, p)))$$

Please try piecing together the second assertion in the same manner, and then try to prove that *there exists no barber*. That is, if we make the  $b$  set non-empty, **we will immediately run into a contradiction**. This is part of your next assignment.

### 13.4.2 If it quacks like a duck, it ain't a doctor

Consider the assertions:

Some patients like every doctor. No patient likes a quack. Therefore, prove that no doctor is a quack.

Please prove it along the lines above!

---

<sup>2</sup>Even non-shavers are people, of course!

<sup>3</sup>Not everyone needs to go to a barber; auto-shavers are often the dominant kind.

## 13.5 Prolog, Canadian Map!

We've learned many of our tools with the help of some "animatronics"—*i.e.*, animating our ideas using programmatic tools. In a sense, we've already done this a lot using z3py, although we did not show you the use of quantifiers, and in many cases, we were using first order logic with extra "theories" (data types) included (*e.g.* reasoning about the properties of integers in the *Send More Money* problem, etc.). In general, first-order logic query solving cannot be automated. More formally, only *semi-algorithms* can be built: solver algorithms, say  $S$ , where:

- If a first-order logic assertion  $A$  is indeed true,  $S$  will say "yes," and halt.
- If  $A$  is false,  $S$  may never find this out, and go into a possible infinite loop.

The fact that *no single algorithm exists to cover all possible assertions A* is one of the fundamental results of computer science. Typically, proofs of this theorem are taught in graduate classes (they are not that hard or long; ask me if you want to study this on your own). This is the reason why z3py restricted first order logic to be *quantifier-free*:

- Validity queries are taken to have universal quantification
- Satisfiability queries are taken to have existential quantification

In the remainder of this section, we will illustrated first-order logic through simple examples written in the programming language *Prolog*. Prolog belongs to the family of logic-programming languages. It implements a fragment of first-order logic, and efficiently answers queries. We will illustrate the use of Prolog in simple examples, and also provide the approximate semantics of these examples in first-order logic.

### 13.5.1 Prolog Basics

To a high degree of simplification, a Prolog program consists of a collection of *facts* and a collection of *rules*. Facts are also special cases of rules. A rule is of the form:

$p(X) :- q(X, Y), r(Y, Z).$

This can be read as

- $p(X)$  is true **if**  $q(X, Y)$  is true **and**  $r(Y, Z)$  is true.
- Here,  $p$  is a *predicate*, and so is  $q$  and  $r$ .

- p is one-ary while q, r are two-ary.
- All predicates are *predicate constants*. They are written in lower-case.
- All variables are written in upper-case.
- One can also interpret this situation as

$$\forall X, Y, Z : (q(X, Y) \wedge r(Y, Z) \Rightarrow p(X))$$

Now let us come to *facts*, which are written as in these examples:

- p(a).
- q(b,c).

These facts simply assert that p is true of *constant* a, q is true of constants b, c, etc. One can also view facts as:

- p(a) :- TRUE.
- q(b,c) :- TRUE.

That is, they are true if TRUE is true—which is always the case.

### 13.5.2 Maps of Canada

Let us take the Canadian map problem and express its details in Prolog. We express the *East-Of* relation. We say that east of British Columbia (b) is Alberta (a), east of Alberta is Saskatchewan (s), then east if it is Manitoba (m), then comes Ontario (o), and finally Quebec (q). These facts are easily seen below.

Now we express the rules?

- `east_of(X, Z)` if `e(X, Z)`. Get it straight from the facts.
- `east_of(X, Z)` if `e(X, Y)` and `east_of(Y, Z)`. *Transitively* follow the chain of reachability! You are learning three concepts here:
  - Logic
  - What is a *transitive* relation
  - Graph traversal
- Finally, `west_of(X, Y)` if `east_of(Y, X)`. This is because neither `east_of()` nor `west_of()` are *symmetric* relations.
- Notice that `e(X, X)` is not true for any X. Thus, e is *not a reflexive relation*.

```
e(b, a).
e(a, s).
e(s, m).
e(m, o).
```

```
e(o,q).

east_of(X,Z) :- e(X,Z).
east_of(X,Z) :- e(X,Y), east_of(Y,Z).
west_of(X,Y) :- east_of(Y,X).
```

### Milking all answers

We can milk all answers one by one, by hitting ;:

```
[ganesh@128-110-83-28 cpsc433]$ gprolog
GNU Prolog 1.4.2
By Daniel Diaz
Copyright (C) 1999-2012 Daniel Diaz
| ?- [map].
compiling /Users/ganesh/Documents/svn/Classes/cs2100/cs2100/f88-s89/cpsc433/map.pl
for byte code...
/Users/ganesh/Documents/svn/Classes/cs2100/cs2100/f88-s89/cpsc433/map.pl compiled,
10 lines read - 1166 bytes written, 7 ms

(1 ms) yes
| ?- east_of(A,B).

A = b
B = a ? ;

A = a
B = s ? ;

A = s
B = m ? ;

A = m
B = o ? ;

A = o
B = q ? ;

A = b
B = s ? ;

A = b
B = m ? ;

A = b
```

```
B = o ? ;
```

```
A = b
```

```
B = q ? ;
```

```
A = a
```

```
B = m ? ;
```

```
A = a
```

```
B = o ? ;
```

```
A = a
```

```
B = q ? ;
```

```
A = s
```

```
B = o ? ;
```

```
A = s
```

```
B = q ? ;
```

```
A = m
```

```
B = q ? ;
```

(2 ms) no

It ends with a no saying “failing to find more answers.”

### Milking Specific Answers

We can milk *specific* answers by proving a match pattern. Here, we are extracting who is to the east of m, then a, and then asking what is the X to whose east lies m:

```
yes
| ?- east_of(m,X).
```

```
X = o ? ;
```

```
X = q ? ;
```

```
no
```

```
| ?- east_of(a,X).
```

```
X = s ? ;
```

```

X = m ? ;
X = o ? ;
X = q ? ;
(1 ms) no
| ?- east_of(X,m).

X = s ? ;
X = b ? ;
X = a ? ;
(1 ms) no
| ?-

```

### 13.5.3 Logical Basics of Prolog

#### Query Semantics

Refer to Question 10 of §13.6. The query you issue is also a special case of the rule syntax. In particular:

```
?- eats_pizza("Bruce").
```

is equivalent to

```
FALSE IF eats_pizza("Bruce").
```

That is, we are trying to prevent (poor) Bruce from eating pizza (because “Bruce” is lying as the antecedent of an implication). Bruce is however clever; he ends up invoking proof by contradiction, and ends up eating.

#### Rule Semantics

Even the rules have multiple readings.

- $\forall X, Y, Z : (q(X, Y) \wedge r(Y, Z) \Rightarrow p(X))$

rewrite the above as

$$\forall X, Y, Z : (\neg q(X, Y) \vee \neg r(Y, Z) \vee p(X))$$

Reduce the scope of the “forall  $Y$ ”:

$$\forall X, Z : (\forall Y : (\neg q(X, Y) \vee \neg r(Y, Z)) \vee p(X))$$

Move the forall  $Y$  into the antecedent of an implication that you create now:

$$\forall X, Z : (\exists Y : q(X, Y) \wedge r(Y, Z)) \Rightarrow p(X)$$

This is why one can often read a rule as

“ $p(X)$  if EXISTS  $Y$  such that  $q(X, Y)$  and  $r(Y, Z)$ .

## 13.6 Review and Labs

1. Negate the formula  $\exists S \subseteq Nat : \forall x \in S : \exists y \in S : x < y$ . Describe the original and negated formulae in English. Are these formulae true or false? In about half a page, write down a description that reveals your thinking.
2. Repeat the previous exercise for the formula (here  $x \mid y$  means “ $x$  evenly divides  $y$ ” – i.e., divides without a remainder.

$$\exists S \subseteq Nat : (\forall y \in S : (\forall x \in Nat : (x \mid y \Rightarrow (x = y) \vee (x = 1))))$$

Answer these questions:

- (a) write down the original and negated formulae and their truth-values.
- (b) write down the meaning of the formulas in English, and justify your answers in about half a page.
- (c) Is the set  $S$  uniquely defined? How many  $S$  sets satisfy the  $\exists S$  part of this formula?
- (d) Can you alter this formula such that the new formula is only true of  $S$  being the set of all Prime numbers?

**Hint:** The following formula is nearly it, but **needs a crucial modification**. Please think through and repair it to be exactly what you need ( $S$  must be all the Primes and only the Primes).

$$\exists S \subseteq Nat :$$

$$\forall x \in Nat :$$

$$\forall y \in Nat : (y \mid x \Rightarrow y = x \vee y = 1)$$

$$\Rightarrow$$

$$x \in S$$

3. Express in first-order logic the proposition that there is a subset of  $Nat$  which contains the set of even numbers  $0, 2, 4, \dots$
4. Express in first-order logic the proposition that corresponding to any natural number  $z$ , there exist two natural numbers  $x$  and  $y$  such that  $x + y = z$ .
5. Express in first-order logic the proposition that for every infinite subset  $A \subseteq Nat$ , there is an infinite subset  $B \subseteq Nat$  such that  $A \subseteq B$ .

6. Consider the Python program run by typing `test()`:

```
def tep1(x):
    print(x),
    return 1 if (x==1) else tep1(x/2) if (x%2==0) else tep1(3 * x + 1)

def test():
    x = input('> ')
    while(x >= 1):
        print tep1(x)
        x = input('> ')
```

- Express the behavior you have observed for this program for several test inputs (specifically, what is the final value returned by `tep1(x)`? Please only state the behavior to the extent you have tested the program on various inputs.
  - It is *conjectured* that this program returns 1 always. Express the conjecture accurately in first-order logic: “either it returns 1 always or for some input  $x$ , it returns something other than a 1.”
7. In CS 3100, you will encounter the following formula, and will be required to negate it. Please do that now, writing the resulting formula neatly, using similar formatting as the formula below. Here,  $\Sigma^*$  stands for “the set of all strings”,  $|y|$  means “the length of string  $y$ ”,  $\epsilon$  means “the empty string”, and  $xuv^iwz$  is a concatenation of strings  $x$ ,  $u$ ,  $v$  (concatenated  $i$  times),  $w$ , and  $z$ .

$$\begin{aligned} \exists N \in \text{Nat} : \\ \forall x, y, z \in \Sigma^* : xyz \in L \wedge |y| \geq N \\ \Rightarrow \\ \exists u, v, w \in \Sigma^* : \\ \wedge \quad y = uvw \\ \wedge \quad |uv| \leq N \\ \wedge \quad v \neq \epsilon \\ \wedge \quad \forall i \geq 0 : xuv^iwz \in L. \end{aligned}$$

8. Write a clear definition of these two propositions in first-order logic:

- There is no infinite subset of any finite subset.

- There is an infinite superset to any set (finite or infinite).

Now, negate the propositions you write, and compute the truth value of the resulting propositions, and justify your answer in about a page.

9. An actual solution posted by me (your instructor) in the Fall 2012 of CS 3100 pertaining to the assertion “there exists an infinite subset of  $\text{Nat}$ ” was wrong!<sup>4</sup> But what I loved about the mistake was the dialog that ensued with a student. Please understand this dialog and answer the embedded questions.

Now, here is how the dialog went:

- I'm a bit confused on your sample solutions for problem 8 on the exam. The logical statement  

$$\exists S \subseteq \text{Nat} : \forall i, j \in S : ((i \in S) \wedge (j > i) \Rightarrow (j \in S))$$
is always true. Question for you: explain why.
- A better statement (sez the student):

$$\exists S \subseteq \text{Nat} : \forall i \in S : \exists j \in S : (i < j)$$

Question for you: Is this true? If so, provide evidence; if not, provide a counterexample.

- Alternatively:

$$\exists S \subseteq \text{Nat} : \forall i : ((i \in S) \Rightarrow \exists j : (j \in S \wedge i < j))$$

Question for you: Is this true? If so, provide evidence; if not, provide a counterexample.

- Now study

$$\exists S \subseteq \text{Nat} : \forall i : ((i \in \text{Nat}) \Rightarrow \exists j : (j \in S \wedge i < j))$$

Question for you: Is this true? If so, provide evidence; if not, provide a counterexample.

- Now study

$$\exists S \subseteq \text{Nat} : S \neq \emptyset \wedge \forall i : ((i \in \text{Nat}) \Rightarrow \exists j : (j \in S \wedge i < j))$$

Question for you: Is this true? If so, provide evidence; if not, provide a counterexample.

---

<sup>4</sup>Egg-on-my-face-wrong!

(f) Is there any difference between the assertions in 9b through 9e?

#### 10. (To Eat or Not To Eat... Pizza)

Consider the following Prolog program:

```

room(3105).
room(102).

does_extra_credit("Bruce").

in_class("Charlie", 2100).
in_class("Tony", 2100).
in_class("Bruce", 2100).

proof_reads_book("Charlie").
proof_reads_book("Tony").

attends_talk("Charlie").
attends_talk("Tony").

goes_to("Bruce", 102).

goes_to("Bruce", 3105).

eats_pizza(S) :- in_class(S, 2100), goes_to(S, R), room(R), R > 102,
               proof_reads_book(S).

eats_pizza(S) :- does_extra_credit(S), attends_talk(S).
```

#### **Exercise:**

- Alter the rules so that Bruce end up eating Pizza today.
- Alter the rules so that Charlie and Tony end up eating Pizza today.
- Enter yourselves into the Prolog program such that you end up eating pizza without proof-reading the book.
- Enter yourselves into the Prolog program such that you end up eating pizza without attending the talk.

11. Please write the score formulation methods of Chapter 6, Sections 15.1.1 and 15.1.2 in first order logic using the  $\exists W$  form such that there is only one witness  $W$  satisfying the formula (representing the white score).



# Chapter 14

## Relations

In previous chapters, we have studied functions. Relations are closely related to functions; in fact, *every function can be viewed as a relation*. Here an example of how it works out. In general, one can view a function such as  $+$  as an infinite set of tuples of the form

$$\{\langle 0, 0, 0 \rangle, \langle 0, 1, 1 \rangle, \langle 1, 0, 1 \rangle, \langle 1, 1, 2 \rangle, \langle 2, 3, 5 \rangle, \langle 30, 2, 32 \rangle, \dots\}$$

Such sets are relations. In this example, the relation is *ternary*. It is *functional relation* because for the same “input” (the first two positions of these triples), there is only one “output” (the last position). In other words, we won’t have the situation

$$\{\langle 0, 0, 0 \rangle, \langle 0, 0, 3 \rangle, \dots\}$$

in a functional relation—although in a general ternary relation, this is perfectly fine. In the rest of the chapter, we shall be mainly concerned with *binary* relations.

### 14.1 Binary Relations

Binary relations help impart structure to sets of related elements. They help form various meaningful *orders* as well as *equivalences*, and hence are central to mathematical reasoning. Our definitions in this chapter follow [6, 10, 7] closely.

A binary relation  $R$  on  $S$  is a subset of  $S \times S$ . It is a relation that can be expressed by a 2-place predicate. Examples: (i)  $x$  loves  $y$ , (ii)  $x > y$ .

Set  $S$  is the *domain* of the relation. It is possible that the domain  $S$  is empty (in which case  $R$  will be empty). In all instances that we consider, the domain  $S$  will be non-empty. However, it is also possible that  $S$  is non-empty while  $R$  is empty (in which case, none of the pairs of elements happen to be related—the situation of an *empty* relation<sup>1</sup>).

We now proceed to examine various types of binary relations. In all these definitions, we assume that the binary relation  $R$  in question is *on*  $S$ , i.e., a subset of  $S \times S$ . For a relation  $R$ , two standard prefixes are employed: *irr-* and *non-*. Their usages will be clarified in the sequel.

Relations can be depicted as graphs. Here are conventions attributed to Andrew Hodges in [7]. The domain is represented by a closed curve (e.g., circle, square, etc) and the individuals in the domain by dots labeled, perhaps,  $a$ ,  $b$ ,  $c$ , and so on. The fact that  $\langle a, b \rangle \in R$  will be depicted by drawing a *single arrow* (or equivalently *one-way arrow*) from dot  $a$  to dot  $b$ . We represent the fact that both  $\langle a, b \rangle \in R$  and  $\langle b, a \rangle \in R$  by drawing a double arrow between  $a$  and  $b$ . We represent the fact that  $\langle a, a \rangle \in R$  by drawing a double arrow from  $a$  back to itself (this is called a *loop*). We shall present examples of these drawings in the sequel.

### 14.1.1 Types of binary relations

We shall use the following examples. Let  $S = \{1, 2, 3\}$ ,  $R_1 = \{\langle x, x \rangle \mid x \in S\}$ ,  $R_2 = S \times S$ , and

$$R_3 = \{\langle 1, 1 \rangle, \langle 2, 2 \rangle, \langle 3, 3 \rangle, \langle 1, 2 \rangle, \langle 2, 1 \rangle, \langle 2, 3 \rangle, \langle 3, 2 \rangle\}.$$

All these (and three more) relations are depicted in Figure 14.1.

#### Reflexive, and Related Notions

$R$  is **reflexive**, if for all  $x \in S$ ,  $\langle x, x \rangle \in R$ . Equivalently,

In  $R$ 's graph, there is no dot without a loop.

Informally, “every element is related to itself.”

A relation  $R$  is **irreflexive** if there are *no* reflexive elements; i.e., for no  $x \in S$  is it the case that  $\langle x, x \rangle \in R$ . Equivalently,

---

<sup>1</sup>A situation where nobody loves anybody else (including themselves!) is an example of  $S \neq \emptyset$  and  $R = \emptyset$ .

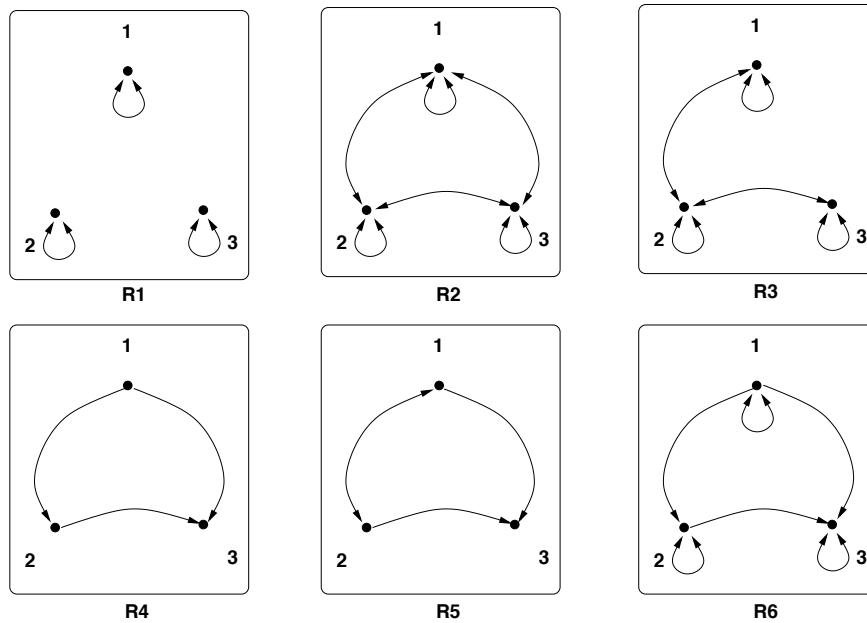


Figure 14.1: Some example binary relations

In  $R$ 's graph, no dot has a loop.

Note that irreflexive is not the negation (complement) of reflexive. This is because the logical negation of the definition of reflexive would be, “there exists  $x \in S$  such that  $\langle x, x \rangle \notin R$ . This is not the same as irreflexive because *all* such pairs must be absent in an irreflexive relation.

A relation  $R$  is **non-reflexive** if it is neither reflexive nor irreflexive. Equivalently,

In  $R$ 's graph, at least one dot has a loop and at least one dot does not.

*Examples:*

- $R_1, R_2, R_3$  are all reflexive.
- If  $S = \emptyset$  (in the empty domain), then  $R = \emptyset$  is reflexive and irreflexive. It is not non-reflexive.
- For  $x, y \in \text{Nat}$ ,  $x = y^2$  is non-reflexive (true for  $x = y = 1$ , false for  $x = y = 2$ ).

### Symmetric, and Related Notions

$R$  is *symmetric* if for all  $x, y \in S$ ,  $\langle x, y \rangle \in R \Rightarrow \langle y, x \rangle \in R$ . Here,  $x$  and  $y$  need not be distinct. Equivalently,

In  $R$ 's graph, there are no single arrows. If the relation holds one way, it also holds the other way.

Examples:  $R_1, R_2$ , and  $R_3$  are symmetric relations. Also note that  $\emptyset$  is a symmetric relation.

$R$  is **asymmetric** if for  $x, y \in S$ , **not necessarily distinct**, if  $\langle x, y \rangle \in R$ , then  $\langle y, x \rangle \notin R$ . Example: “elder brother” is an asymmetric relation, and so is  $<$  over  $\text{Nat}$ . Asymmetric relations need not be total; that is, it is not required that for two arbitrary  $x, y$ , we have  $\text{elderbrother}(x, y)$  or  $\text{elderbrother}(y, x)$ . But if it holds one way, it does not hold the other way. Equivalently,

There are no double arrows in its graph; if the relation holds one way, it does not hold the other.

Curiously, this rules out  $\leq$ . We have  $0 \leq 0$ . But it does not follow that  $\neg(0 \leq 0)$  because of the *not necessarily distinct* aspect.

Again, note that *asymmetric* is *not* the same as the negation of (the definition of) symmetric. The negation of the definition of symmetric would be that *there exists* distinct  $x$  and  $y$  such that  $\langle x, y \rangle \in R$ , but  $\langle y, x \rangle \notin R$ .

$R$  is **non-symmetric** if it is neither symmetric nor asymmetric (there is at least one single arrow and at least one double arrow).

Example:  $\emptyset$  is symmetric and asymmetric, but not non-symmetric.

$R$  is **antisymmetric** if for all  $x, y \in S$ ,  $\langle x, y \rangle \in R \wedge \langle y, x \rangle \in R \Rightarrow x = y$  (they are the same element). Equivalently,

There is no double arrow unless it is a loop.

Antisymmetry is a powerful notion that, unfortunately, is too strong for many purposes. Consider the elements of  $2^S$ , the powerset of  $S$ , as an example. If, for any two elements  $x$  and  $y$  in  $S$ , we have  $x \subseteq y$  and  $y \subseteq x$ , then we can conclude that  $x = y$ . Therefore, the set containment relation  $\subseteq$  is *antisymmetric*; and hence, antisymmetry is appropriate for comparing two sets in the “less than or equals” sense.

Consider, on the other hand, two basketball players,  $A$  and  $B$ . Suppose the coach of their team defines the relation  $\leq_{BB}$  as follows:  $A \leq_{BB} B$  if and only if  $B$  has more abilities or has the same abilities as  $A$ . Now, if we have two players  $x$  and  $y$  such that  $x \leq_{BB} y$  and  $y \leq_{BB} x$ , we can conclude that they have identical abilities - they don't end up becoming the very same person, however! Hence,  $\leq_{BB}$  must not be antisymmetric. Therefore, depending on what we are comparing, antisymmetry may or may not be appropriate.

### Transitive, and Related Notions

To define transitivity in terms of graphs, we need the notions of a *broken journey* and a *short cut*. There is a broken journey from dot  $x$  to dot  $z$  via dot  $y$ , if there is an arrow from  $x$  to  $y$  and an arrow from  $y$  to  $z$ . Note that dot  $x$  might be the same as dot  $y$ , and dot  $y$  might be the same as dot  $z$ . Therefore if  $\langle a, a \rangle \in R$  and  $\langle a, b \rangle \in R$ , there is a broken journey from  $a$  to  $b$  via  $a$ . Example: there is a broken journey from Utah to Nevada via Arizona. There is also a broken journey from Utah to Nevada via Utah.

There is a short cut just if there is an arrow direct from  $x$  to  $z$ . So if  $\langle a, b \rangle \in R$  and  $\langle b, c \rangle \in R$  and also  $\langle a, c \rangle \in R$ , we have a broken journey from  $a$  to  $c$  via  $b$ , together with a short cut. Also if  $\langle a, a \rangle \in R$  and  $\langle a, b \rangle \in R$ , there is a broken journey from  $a$  to  $b$  via  $a$ , together with a short cut.

*Example:* There is a broken journey from Utah to Nevada via Arizona, and a short cut from Utah to Nevada.

$R$  is **transitive** if for all  $x, y, z \in S$ ,  $\langle x, y \rangle \in R \wedge \langle y, z \rangle \in R \Rightarrow \langle x, z \rangle \in R$ . Equivalently,

There is no broken journey without a short cut.

$R$  is **intransitive** if, for all  $x, y, z \in S$ ,  $\langle x, y \rangle \in R \wedge \langle y, z \rangle \in R \Rightarrow \langle x, z \rangle \notin R$ . Equivalently,

There is no broken journey *with* a short cut.

$R$  is **non-transitive** if and only if it is neither transitive nor intransitive. Equivalently,

There is at least one broken journey with a short cut and at least one without.

*Examples:*

- Relations  $R_1$  and  $R_2$  above are transitive.
- $R_3$  is non-transitive, since it is lacking the pair  $\langle 1, 3 \rangle$ .
- Another non-transitive relation is  $\neq$  over  $Nat$ , because from  $a \neq b$  and  $b \neq c$ , we cannot always conclude that  $a \neq c$ .
- $R_4$  is irreflexive, transitive, and asymmetric.
- $R_5$  is still irreflexive. It is not transitive, as there is no loop at 1. It is not intransitive because there is a broken journey (2 to 3 via 1) with a short cut (2 to 1). It is non-transitive because there is one broken journey without a short cut and one without.
- $R_5$  is not symmetric because there are single arrows.
- $R_5$  is not asymmetric because there are double arrows.
- From the above, it follows that  $R_5$  is non-symmetric.
- $R_5$  is not antisymmetric because there is a double arrow that is not a loop.

### 14.1.2 Preorder (reflexive plus transitive)

If  $R$  is reflexive and transitive, then it is known as a *preorder*. Continuing with the example of basketball players, let the  $\leq_{BB}$  relation for three members  $A$ ,  $B$ , and  $C$  of the team be

$$\{\langle A, A \rangle, \langle A, B \rangle, \langle B, A \rangle, \langle B, B \rangle, \langle A, C \rangle, \langle B, C \rangle, \langle C, C \rangle\}.$$

This relation is a *preorder* because it is reflexive and transitive. It helps compare three players  $A$ ,  $B$ , and  $C$ , treating  $A$  and  $B$  to be equivalent in abilities, and  $C$  to be superior in abilities to both.

In Section 14.4, we present a more elaborate example of a preorder.

### 14.1.3 Partial order (preorder plus antisymmetric)

If  $R$  is reflexive, antisymmetric, and transitive, then it is known as a *partial order*. As shown in Section 14.1.1 under the heading of *antisymmetry*, the subset or equals relation  $\subseteq$  is a partial order.

### 14.1.4 Total order, and related notions

A *total order* is a special case of a partial order.  $R$  is a total order if for all  $x, y \in S$ , either  $\langle x, y \rangle \in R$  or  $\langle y, x \rangle \in R$ . Here,  $x$  and  $y$  need not be distinct (this is consistent with the fact that total orders are reflexive).

The  $\leq$  relation on  $Nat$  is a total order. Note that ' $<$ ' is not a total order, because it is not reflexive.<sup>2</sup> However, ' $<$ ' is transitive. Curiously, ' $<$ ' is antisymmetric.

A relation  $R$  is said to be **total** if for all  $x \in S$ , there exists  $y \in S$  such that  $\langle x, y \rangle \in R$ . In other words, a “total” relation is one in which every element  $x$  is related to at least one other element  $y$ . If we consider  $y$  to be the image (mapping) of  $x$  under  $R$ , this definition is akin to the definition of a *total* function.

Note again that  $R$  being a *total order* is **not** the same as  $R$  being a partial order and a total relation. For example, consider the following relation  $R$  over set  $S = \{a, b, c, d\}$ :

$$R = \{\langle a, a \rangle, \langle b, b \rangle, \langle c, c \rangle, \langle d, d \rangle, \langle a, b \rangle, \langle c, d \rangle\}$$

$R$  is a partial order.  $R$  is also a total relation. However,  $R$  is *not* a total order, because there is *no* relationship between  $b$  and  $c$  (neither  $\langle b, c \rangle$  nor  $\langle c, b \rangle$  is in  $R$ ).

### 14.1.5 Relational Inverse

The inverse of a relation  $R$  can be defined as follows:

$$R^{-1}(y, x) \text{ if and only if } R(x, y).$$

Thus, if

$$R = \{\langle x, y \rangle \mid p(x, y)\}$$

for some characteristic predicate  $p$ , then  $R^{-1}$  is as follows:

$$R^{-1} = \{\langle y, x \rangle \mid p(x, y)\}.$$

- Example: The inverse of the  $<$  relation over natural numbers  $Nat$  is the relation  $>$  over  $Nat$ . It is *not* the same as  $\geq$ . (Note that if we

---

<sup>2</sup>Some authors are known to abuse these definitions, and consider  $<$  to be a total order. It is better referred to as *strict* total order or *irreflexive* total order.

negate the characteristic predicate defining  $<$ , we will have obtained  $\geq$ . This is however not how you obtain relational inverses. Relational inverses are obtained by “flipping” the tuples around.)

- Example: The inverse of the  $<$  relation over Integers  $Int$  (positive and negative whole numbers) is the relation  $>$  over  $Int$ .
- Observation: If we take every edge in the graph of relation  $R$  and reverse the edges, we obtain the edges in the graph of relation  $R^{-1}$ .

## 14.2 Equivalence (Preorder plus Symmetry)

An equivalence relation is reflexive, symmetric, and transitive. Consider the  $\leq_{BB}$  relation for three basketball players  $A$ ,  $B$ , and  $C$ . Now, consider a “specialization” of this relation obtained by leaving out certain edges:

$$\equiv_{BB} = \{\langle A, A \rangle, \langle A, B \rangle, \langle B, A \rangle, \langle B, B \rangle, \langle C, C \rangle\}.$$

This relation is an equivalence relation, as can be easily verified.

Note that  $\equiv_{BB} = \leq_{BB} \cap \leq_{BB}^{-1}$ . In other words, this equivalence relation is obtained by taking the preorder  $\leq_{BB}$  and intersecting it with its inverse. The fact that  $\leq_{BB} \cap \leq_{BB}^{-1}$  is an equivalence relation is not an accident. The following section demonstrates a general result in this regard.

### 14.2.1 Intersecting a preorder and its inverse

**Theorem 14.1** *The relation obtained by intersecting a preorder  $r$  with its inverse  $r^{-1}$  is an equivalence relation.*

**Proof:** First we show that  $R \cap R^{-1}$  is reflexive. Let  $R$  be a preorder over set  $S$ . Therefore,  $R$  is reflexive, i.e., it contains  $\langle x, x \rangle$  pairs for every  $x \in S$ . From the definition of  $R^{-1}$ , it also contains these pairs. Hence, the intersection contains these pairs. Therefore,  $R \cap R^{-1}$  is reflexive.

Next we show that  $R \cap R^{-1}$  is symmetric. That is, to show that for every  $x, y \in S$ ,  $\langle x, y \rangle \in R \cap R^{-1}$  implies  $\langle y, x \rangle \in R \cap R^{-1}$ . If the antecedent, i.e.,  $\langle x, y \rangle \in R \cap R^{-1}$  is false, the assertion is vacuously true. Consider when the antecedent is true for a certain  $\langle x, y \rangle$ . These  $x$  and  $y$  must be such that  $\langle x, y \rangle \in R$  and  $\langle x, y \rangle \in R^{-1}$ . The former implies that  $\langle y, x \rangle \in R^{-1}$ . The latter implies that  $\langle y, x \rangle \in R$ . Hence,  $\langle y, x \rangle \in R \cap R^{-1}$ . Hence,  $R \cap R^{-1}$  is symmetric.

Next we prove that  $R \cap R^{-1}$  is transitive. Since  $R$  is a preorder, it is transitive. We now argue that the inverse of any transitive relation is transitive. From the definition of transitivity, for every  $x, y, z \in S$ , from the antecedents  $\langle x, y \rangle \in R$  and  $\langle y, z \rangle \in R$ , the consequent  $\langle x, z \rangle \in R$  follows. From these antecedents, we have that  $\langle y, x \rangle \in R^{-1}$  and  $\langle z, y \rangle \in R^{-1}$  respectively. From the above conclusion  $\langle x, z \rangle \in R$ , we can infer that  $\langle z, x \rangle \in R^{-1}$ . Hence,  $R^{-1}$  is transitive and so is the conjunction of  $R$  and  $R^{-1}$ .

### 14.2.2 Identity relation

Given a set  $S$ , the identity relation  $R$  over  $S$  is  $\{\langle x, x \rangle \mid x \in S\}$ . An identity relation is one extreme (special case) of an equivalence relation. This relation is commonly denoted by the equality symbol,  $=$ , and relates equals with equals. Please note the contrast with Theorem 14.1.

### 14.2.3 Universal relation

The *universal relation*  $R$  over  $S$  is  $S \times S$ , and represents the other extreme of relating everything to everything else. This is often an uninteresting binary relation.<sup>3</sup>

### 14.2.4 Equivalence class

An equivalence relation  $R$  over  $S$  partitions  $\text{elements}(R) = \text{domain}(R) \cup \text{codomain}(R)$  into *equivalence classes*. Intuitively, the equivalence classes  $E_i$  are those subsets of  $\text{elements}(R)$  such that every pair of elements in  $E_i$  is related by  $R$ , and  $E_i$ s are the maximal such subsets. More formally, given an equivalence relation  $R$ , there are two cases:

1.  *$R$  is a universal relation.* In this case, there is a single equivalence class  $E_1$  associated with  $R$ , which is  $\text{elements}(R)$  itself.
2.  *$R$  is not a universal equivalence relation.* In this case, an equivalence class  $E_i$  is a maximal proper subset of  $\text{elements}(R)$  such that the restriction of  $R$  on  $E_i$ ,  $R|_{E_i}$ , is universal (meaning that every pair of elements inside each of the  $E_i$ s is related by  $R$ ).

---

<sup>3</sup>This is sort of what would happen if one were to give everyone in a theory class an ‘A’ grade.

Putting it all together, the set of all equivalence classes of an equivalence relation  $R$  is written “ $\text{elements}(R)_{/R}$ .” It can be read, “the elements of  $R$  partitioned according to  $R$ .” In general, we will write  $S/\equiv$ , meaning “set  $S$  partitioned according to the equivalence relation  $\equiv$ .”

in Section 14.4.1, we will demonstrate Theorem 14.1 on the *Power* relation that relates machine types.

### 14.2.5 Reflexive and transitive closure

The *reflexive closure* of  $R$ , denoted by  $R^0$ , is

$$R^0 = R \cup \{\langle x, x \rangle \mid x \in S\}.$$

This results in a relation that is reflexive.

The *transitive closure* of  $R$ , denoted by  $R^+$ , is

$$R^+ = R \cup \{\langle x, z \rangle \mid \exists y \in S : \langle x, y \rangle \in R \wedge \langle y, z \rangle \in R^+\}.$$

$R^+$  is the least such set. The use of ‘+’ highlights the fact that transitive closure relates items that are “one or more step away.”

The reflexive *and* transitive closure of a relation  $R$ , denoted by  $R^*$ , is

$$R^* = R^0 \cup R^+.$$

The use of ‘\*’ highlights the fact that reflexive and transitive closure relates items that are “zero or more steps away.”

**Example:** Consider a directed graph  $G$  with nodes  $a, b, c, d, e$ , and  $f$ . Suppose it is necessary to define the *reachability* relation among the nodes of  $G$ . Oftentimes, it is much easier to instead define the one-step reachability relation

$$\text{Reach} = \{\langle a, b \rangle, \langle b, c \rangle, \langle c, d \rangle, \langle e, f \rangle\}$$

and let the users perform the reflexive and transitive closure of *Reach*. Doing so results in  $\text{Reach}_{RTclosed}$ , that has all the missing reflexive and transitive pairs of nodes in it:

$$\begin{aligned} \text{Reach}_{RTclosed} = & \{\langle a, b \rangle, \langle b, c \rangle, \langle c, d \rangle, \langle e, f \rangle, \langle a, a \rangle, \langle b, b \rangle, \langle c, c \rangle, \langle d, d \rangle, \\ & \langle e, e \rangle, \langle f, f \rangle, \langle a, c \rangle, \langle a, d \rangle, \langle b, d \rangle\}. \end{aligned}$$

### 14.2.6 Illustration of Transitive Closure Using Prolog

We illustrate the reflexive and transitive closure construction on the relation *Reach* through the Prolog program below. We will use relation *reach1* to explicitly capture the one-step relations that a user might care to provide. We also provide the set of nodes in the graph through the relation *nodes*(*L*) where *L* is a list.

We then introduce *reach* to denote the reflexive and transitive closure of *reach1*. There are two cases: one being that *X* can reach itself if *X* is a node. The second is that *X* can reach *Y* if *X* can *reach1* *Z* and *Z* can reach *Y*. We then collect all solutions of the *reach(X,Y)* call and put the list *[X,Y]* for which *reach(X,Y)* is true into a bag *Bag*. The final result shows that *Bag* indeed represents the reflexive and transitive closure of *reach1*.

```
nodes([a,b,c,d,e,f]).  
  
reach1(a,b).  
reach1(b,c).  
reach1(c,d).  
reach1(e,f).  
  
reach(X,X) :- nodes(N), member(X,N).  
  
reach(X,Y) :- reach1(X,Z), reach(Z,Y).  
  
findall([X,Y], reach(X,Y), Bag).  
  
Bag = [[a,a],[b,b],[c,c],[d,d],[e,e],[f,f],[a,b],  
       [a,c],[a,d],[b,c],[b,d],[c,d],[e,f]]
```

## 14.3 Powersets

Powersets are an extremely good example of partial orders. In this section we discuss powersets, how to generate them in Python, and some of the “real world” situations where Powersets occur.

Suppose we have a set of elements  $S = \{0, 1, 2\}$  and we put an on/off switch above each element. Suppose we generate new sets by playing with the switches: when all switches are off, we generate  $\{\}$ ; next with the switch associated with 2 and 0 on, we generate  $\{0, 2\}$ ; and so on. The number of

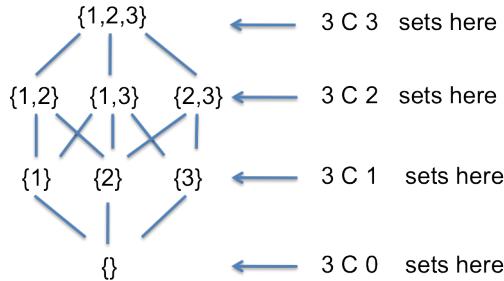


Figure 14.2: Powerset as a Lattice

these sets generated is  $2^3$  (all Boolean combinations of the switches). The set of all these is the *powerset*.

We can count the number of sets in another way also:  $\sum_{k=0}^N \binom{N}{k} = 2^N$ . In Figure 14.2, we perform this summation for  $N = 3$ , giving us the eight (8) subsets of  $\{1, 2, 3\}$ .

Note that powerset of  $S$  is the set of all its subsets (not merely proper subsets, but *all subsets*). This is why  $\{\}$  has a powerset, which equals  $\{\{\}\}$ .

The way the Powerset algorithm works is easy to explain with respect to the structure of the recursion in Figure 14.3:

- The powerset of the empty set  $\{\}$  is  $\{\{\}\}$  because we are supposed to return the set of subsets of  $\{\}$ ; and there is only one subset for  $\{\}$ , which is itself.

```
L=list(S)
if L==[]:
    return([[]])
```

- For a non-empty set, the powerset is calculated as follows:
  - First, calculate the powerset of the *rest* of the set:
 

```
else:
            pow_rest0 = pow(L[1:])
```
  - Then calculate the set obtained by pasting the first element of the original set onto every set in `pow_rest0`:
 

```
pow_rest1 = list(map(lambda ls: [L[0]]+ls, pow_rest0))
```
  - Finally, compute a set of sets, containing all the sets within `pow_rest0` and `pow_rest1`:
 

```
return(pow_rest0 + pow_rest1)
```

```

def pow(S):
    """Powerset of a set L. Since sets/lists are unhashable,
    we convert the set to a list, perform the powerset operations,
    leaving the result as a list (can't convert back to a set).
    pow(set(['ab', 'bc'])) --> [['ab', 'bc'], ['bc'], ['ab'], []]
    """
    L=list(S)
    if L==[]:
        return([[]])
    else:
        pow_rest0 = pow(L[1:])
        pow_rest1 = list(map(lambda ls: [L[0]]+ls, pow_rest0))
        return(pow_rest0 + pow_rest1)
    ----
>>> pow
<function pow at 0x026E1FB0>
>>> pow({1,2,3})
[[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]
>>> pow({})
[]
>>> pow({'hi','there',5})
[[], [5], ['there'], ['there', 5], ['hi'], ['hi', 5], ['hi', 'there'], ['hi', 'there', 5]]
>>> len(pow(range(1)))
2
>>> len(pow(range(2)))
4
>>> len(pow(range(4)))
16
>>> len(pow(range(10)))
1024
>>> len(pow(range(20)))
1048576

```

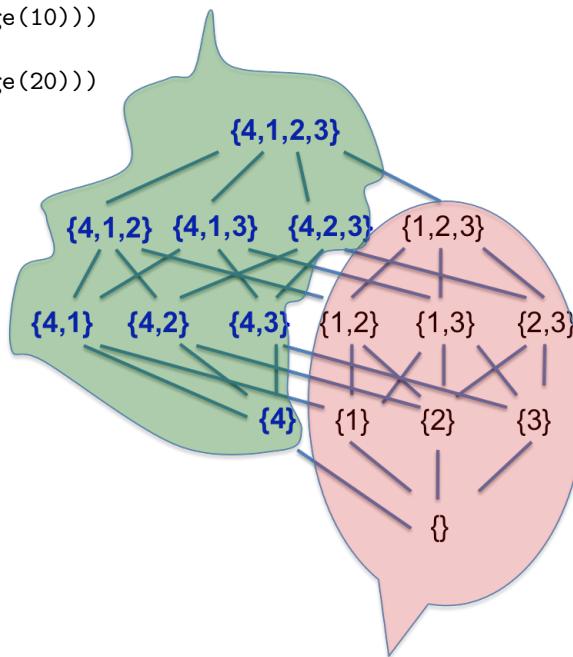


Figure 14.3: The Powerset function, and how it recurses

### 14.3.1 Application: Electoral Maps



Figure 14.4: Recent electoral maps of the USA

You have seen maps such as in Figure 14.4. There are a total of  $2^{50}$  such electoral maps possible, with Republican (red) and Democrat (blue) states shown [4].

## 14.4 The *Power* Relation between Machines

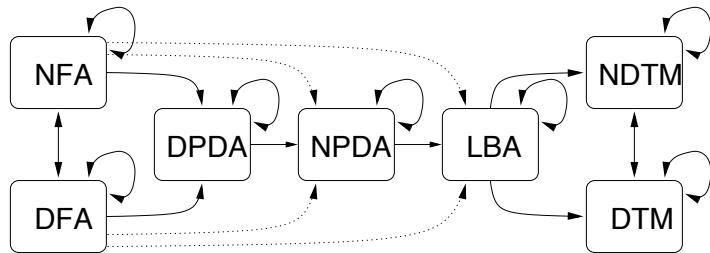


Figure 14.5: The binary relation *Power* is shown. The dotted edges are *some* of the edges implied by transitivity. Undotted and dotted means the same in this diagram. Therefore, *Power* actually contains: (i) the pairs corresponding to the solid edges, (ii) the pairs indicated by the dotted edges, (iii) and those pairs indicated by those dotted transitive edges not shown.

An example that nicely demonstrates the versatility of preorders is the one that defines the “power” of computing machines. Let

$$MT = \{dfa, nfa, dpda, npda, lba, dtm, ndtm\}$$

represent the set of machine types studied in this book. These acronyms stand for deterministic finite automata, nondeterministic finite automata,

deterministic push-down automata, nondeterministic push-down automata, linear bounded automata, deterministic Turing machines, and nondeterministic Turing machines, respectively. A binary relation called *Power* that situates various machine types into a dominance relation is shown in Figure 14.5. Each ordered pair in the relation shows up as an arrow ( $\rightarrow$ ). We draw an arrow from machine type  $m_1$  to  $m_2$  if for every task that a machine of type  $m_1$  can perform, we can find a machine of type  $m_2$  to do the same task. Spelled out as a set, the relation *Power* is

$$\begin{aligned} \text{Power} = & \{\langle dfa, dfa \rangle, \langle nfa, nfa \rangle, \\ & \langle dpda, dpda \rangle, \langle npda, npda \rangle, \\ & \langle lba, lba \rangle, \\ & \langle dtm, dtm \rangle, \langle ndtm, ndtm \rangle, \\ & \langle dfa, nfa \rangle, \langle nfa, dfa \rangle, \langle dtm, ndtm \rangle, \\ & \langle ndtm, dtm \rangle, \langle dfa, dpda \rangle, \langle nfa, dpda \rangle, \\ & \langle dfa, lba \rangle, \langle nfa, lba \rangle, \\ & \langle dpda, npda \rangle, \langle npda, dtm \rangle, \langle npda, ndtm \rangle, \\ & \langle dpda, lba \rangle, \langle npda, lba \rangle, \\ & \langle lba, dtm \rangle, \langle lba, ndtm \rangle, \\ & \langle dfa, npda \rangle, \langle nfa, npda \rangle, \\ & \langle dpda, dtm \rangle, \langle dpda, ndtm \rangle, \\ & \langle dfa, dtm \rangle, \langle dfa, ndtm \rangle, \\ & \langle nfa, dtm \rangle, \langle nfa, ndtm \rangle \\ & \}. \end{aligned}$$

We will now study this dominance relation step by step.

Any machine is as powerful as itself; hence, *Power* is reflexive, as shown by the ‘self-loops.’ *Power* is *transitive* relation because for every  $m_1, m_2, m_3 \in MT$ , if  $\langle m_1, m_2 \rangle \in \text{Power}$  and  $\langle m_2, m_3 \rangle \in \text{Power}$ , then certainly  $\langle m_1, m_3 \rangle \in \text{Power}$ . (We do not show the transitive edges in the drawing). *Power* is *not* antisymmetric because even though *dfa* and *nfa* dominate each other, they have *distinct* existence in the set of machine types  $MT$ . *Power* is a preorder, and in our minds captures exactly how the space of machines must be subdivided. It is not a partial order.

#### 14.4.1 The equivalence relation over machine types

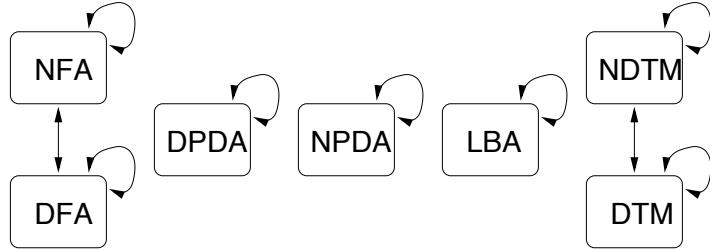


Figure 14.6: The equivalence relation  $\text{Power} \cap \text{Power}^{-1}$

Applying Theorem 14.1 to  $\text{Power}$ , we obtain the equivalence relation  $\equiv_{MT}$  over machine types:

$$\begin{aligned} \text{Power} \cap \text{Power}^{-1} = & \{ \langle dfa, dfa \rangle, \langle nfa, nfa \rangle, \\ & \langle dpda, dpda \rangle, \langle npda, npda \rangle, \\ & \langle dtm, dtm \rangle, \langle ndtm, ndtm \rangle, \\ & \langle lba, lba \rangle, \\ & \langle dfa, nfa \rangle, \langle nfa, dfa \rangle, \langle dtm, ndtm \rangle, \langle ndtm, dtm \rangle \\ & \}. \end{aligned}$$

Figure 14.6 illustrates  $\equiv_{MT}$ . We can see that  $\equiv_{MT}$  subdivides  $MT$  into five equivalence classes:  $\{dfa, nfa\}$ ,  $\{dpda\}$ ,  $\{npda\}$ ,  $\{lba\}$ , and  $\{dtm, ndtm\}$ . Let us now look at this equivalence relation formally. As pointed out earlier, an equivalence relation partitions the underlying set into equivalence classes. The set of equivalence classes is denoted by  $\text{elements}(R)_R$  below. The mathematical definition below elaborates on equivalence relations and equivalence classes:

$$\begin{aligned} \text{elements}(R)_R = \{ \rho \mid & \rho \subseteq \text{elements}(R) \\ & \wedge \rho \times \rho \subseteq R \\ & \wedge \neg \exists Y : \rho \subset Y \wedge Y \subseteq \text{elements}(R) \\ & \wedge Y \times Y \subseteq R \} \end{aligned}$$

This definition says the following:

- Each equivalence class  $E_i$  is a subset of the elements of the underlying set

- For each  $E_i$ ,  $E_i \times E_i$  is a subset of the equivalence relation.
- Each such set  $E_i$  is maximal (no bigger  $Y$  containing each  $E_i$  exists)

Coming to our specific example, there are five equivalence classes over  $\text{Power} \cap \text{Power}^{-1}$ :  $\{\text{dfa}, \text{nfa}\}$ ,  $\{\text{dpda}\}$ ,  $\{\text{npda}\}$ ,  $\{\text{lba}\}$ , and  $\{\text{dtm}, \text{ndtm}\}$ . As can be seen from Figure 14.6, they are five maximal universal relations. Given an equivalence relation  $R$ , the equivalence class of an element  $x \in \text{elements}(R)$  is denoted by  $[x]$ .

## 14.5 Review and Labs

1. Characterize these human relationships in as many ways as you can (attribute as many attributes as you can think of). For instance, answer whether these relations are
  - Reflexive?
  - Irreflexive?
  - Non-reflexive?
  - Symmetric?
  - Asymmetric?
  - Non-symmetric?
  - Transitive?

(a)  $x$  is the spouse of  $y$   
 (b)  $x$  is a sibling of  $y$   
 (c)  $x$  is an ancestor of  $y$   
 (d)  $x$  likes  $y$   
 (e)  $x$  does not know  $y$
2. Draw a directed graph  $G$  with nodes being the subsets of set  $\{1, 2\}$ , and with an edge from node  $n_i$  to node  $n_j$  if either  $n_i \subseteq n_j$  or  $|n_i| = |n_j|$ .  $|S|$  stands for the size or cardinality of set  $S$ . Now, view the above graph  $G$  as a binary relation  $R_G$ . Write down the pairs in this relation. A relation is written as

$$\{\langle a_0, b_0 \rangle, \langle a_1, b_1 \rangle, \dots\}$$

where  $\langle a_i, b_i \rangle$  are the pairs in the relation.

- (a) Is  $R_G$  a preorder? Explain.
- (b) Is  $R_G$  a partial order? Explain.
- (c) Is  $R_G$  a functional relation? Why, or why not? A functional relation is one for which for one ‘input’, at most one ‘output’ is produced. As far as an  $n$ -tuple  $\langle a_1, a_2, \dots, a_n \rangle$  goes, the last position

$a_n$  is regarded as the ‘output’ and the remaining  $n - 1$  positions are regarded as the ‘input.’ Therefore, for a functional binary relation, if  $\langle a_i, b_j \rangle$  and  $\langle a_i, b_k \rangle$  are in the relation, then  $b_j = b_k$ .

3. Define the following relation  $\leq$  over intervals of *Reals* as follows. Let  $[a, b]$  for  $a, b \in \text{Real}$  be an interval. For intervals  $[a, b]$  and  $[c, d]$ ,  $[a, b] \leq [c, d]$  iff  $a \leq c \wedge d \leq b$ . Now, is  $\leq$  a preorder? Is it a partial order? Show that  $\leq \cap \leq^{-1}$  is an identity relation.
4. Continuing with the above question, let’s change  $\leq$  as follows. For intervals  $[a, b]$  and  $[c, d]$ , define

$$[a, b] \leq [c, d] \Leftrightarrow \exists x : (a \leq x \leq b) \wedge (c \leq x \leq d).$$

Now, is  $\leq$  a preorder or a partial order? Is  $\leq \cap \leq^{-1}$  an identity equivalence relation or simply an equivalence relation?

5. Consider the discussion in §13.5.2 pertaining to canadian maps. In particular, study the definition of `east_of`. Is this relation
  - (a) Reflexive?
  - (b) Irreflexive?
  - (c) Non-reflexive?
  - (d) Symmetric?
  - (e) Asymmetric?
  - (f) Non-symmetric?
  - (g) Transitive?
  - (h) What is the formal relationship between `east_of` and `west_of` (what relational operator converts one to the other)?
6. Modify the `east_of` definition along the lines of what is discussed in §14.2.6 to end up defining the `reach` relation (not merely the `east_of`). Run this program, producing the Bag of all reachable pairs. How many pairs are expected and are produced?
7. Consider electrical *resistors*. Let the set of resistors be defined as follows. If  $x$  is a *Real* number, then the term `res(x)` is a resistor. If  $r_1$  and  $r_2$  are in `resistor`, then `series(r1, r2)` is a resistor. (Again, `series(r1, r2)` is a term). Some of the elements in set `Resistor` are:

- $\text{res}(1), \text{res}(2), \text{res}(3), \text{series}(\text{res}(1), \text{res}(2)),$
- $\text{series}(\text{res}(1), \text{series}(\text{res}(2), \text{res}(3))),$  and
- $\text{series}(\text{series}(\text{res}(2), \text{res}(3)), \text{res}(1)).$

In any<sup>4</sup> circuit, we must be able to substitute  $\text{res}(3)$  by  $\text{series}(\text{res}(1), \text{res}(2))$ , as series connected resistors add up the resistivity. However, we *cannot* regard  $\text{res}(3)$  and  $\text{series}(\text{res}(1), \text{res}(2))$  as *identical* because they have distinct representations in the set. To compare two resistors “properly,” we define the relation  $\text{Resistor\_leq}$ :

$$\text{Resistor\_leq} = \{\langle x, y \rangle \mid \text{sumR}(x) \leq \text{sumR}(y)\}$$

where  $\text{sumR}$  is defined as follows:

$$\begin{aligned}\text{sumR}(\text{res}(i)) &= i, \\ \text{sumR}(\text{series}(x, y)) &= \text{sumR}(x) + \text{sumR}(y).\end{aligned}$$

Given the above, show that  $\text{Resistor\_leq}$  is reflexive and transitive, and hence a preorder. Also show that  $\text{Resistor\_leq}$  is not antisymmetric.

---

<sup>4</sup>We are ignoring aspects such as size, weight, tolerance, etc.

# Chapter 15

## Mastermind Game Using Logic

This chapter ties together some of the concepts you have been studying so far, and introduces your course project: that of designing a version of the game of Mastermind! A basic game will be provided to you as a Python program and described in these notes. The aim of your project is to improve this game in a few prescribed ways (basically improving the number of player options and making solution search more efficient). These steps will soon be elaborated; but first, let us discuss the rules of Mastermind.

### 15.1 One Popular Version of Mastermind

One of the popular versions of Mastermind involves a game board as described in [11]. A *code setter* sets a secret code consisting of four (4) pegs drawn from a supply of colored pegs, with the total number of peg colors typically being six (6). The *solver* makes an initial guess—also consisting of four pegs from the same supply of colored pegs. It is assumed that the code setter and the solver never run out of pegs of any desired color. In response to the initial guess made by the solver, the code setter gives scores based on the following criteria:

- How many pegs are of the right color and are in the right place? This number is reflected in the number of *red*<sup>1</sup> pegs in the score computed and displayed by the scorer (typically by means of red pegs placed alongside the solver's pegs).

---

<sup>1</sup>In many game boards, instead of red, other colors—typically black—are employed.

- How many *additional* pegs are of the right color and are in the wrong place (the “right place” is of course defined by the secret code)? This count is reflected in the number of white pegs that the code setter places in response to each move of the solver.



Figure 15.1: A Mastermind Game, showing the Code Setting (left; the board has been rotated around) and a Score fetched by the Solver (right). The peg positions are defined with respect to the Solver, going left to right.

### 15.1.1 Formulation of Scoring: Approach 1

More precisely, let  $g$  be the solver’s guess, as an array of color values, where  $g_i$  is the solver’s guess for peg  $i$ ; and let  $c$  be the code created by the code setter, also as an array of color values. Define arrays  $\rho$  and  $\omega$  as follows:

$$\rho_i = \begin{cases} 1 & \text{if } g_i = c_i \\ 0 & \text{otherwise} \end{cases}$$

$$\omega_i = \begin{cases} j & \text{if } \rho_i = \rho_j = 0, g_i = c_j, \text{ and } \omega_k \neq j \text{ for } k < i \\ 0 & \text{otherwise} \end{cases}$$

*redScore* and *whiteScore* are the number of non-zero entries in  $\rho$  and  $\omega$ , respectively.

Figure 15.1 illustrates the scoring method.

- Notice that the first Solver move fetched two Reds because of the Blue and Orange peg presented by the Solver matching the ones set by the Coder. The single white is due to the pink being present in the Code,

but in the wrong place, *and since the Solver had not set another pink matching the Code Pink in the correct position*. In fact, what the Solver set opposite to the Coder's pink is a Yellow.

- The second Scorer move fetched four whites because all colors set by the Scorer are present, but *none* are in the right place.
- The sets and scores evaluate to the following (where we count the positions 1,2,3,4 going left-to-right from the Scorer's perspective):

– First round

- \* We have  $redScore = 2$ . This is because  $g_3 = c_3$  and  $g_4 = c_4$ , so the number of non-zero entries in  $\rho$  is 2.
- \* We have  $whiteScore = 1$ . Notice that  $\omega_3 = \omega_4 = 0$  because  $\rho_3 = \rho_4 = 1$ .  $\omega_1 = 0$  because  $g_1$  is not equal to any  $c_j$ .  $\omega_2 = 1$  because  $g_2 = c_1$ , and  $\omega_k \neq 1$  for any  $k < 2$  (we haven't previously used this peg for a white). There is only one non-zero entry in  $\omega$ , so  $whiteScore = 1$ .

– Second round

- \* We have  $redScore = 0$ . This is because  $g_i \neq c_i$  for all  $i$ .
- \* We have  $whiteScore = 4$ . One way to score white pegs is

- $\omega_1 = 2$
- $\omega_2 = 1$
- $\omega_3 = 4$
- $\omega_4 = 3$

Another way is

- $\omega_1 = 3$
- $\omega_2 = 1$
- $\omega_3 = 4$
- $\omega_4 = 2$

In either case, the number of non-zero entries is 4, so  $whiteScore = 4$ .

If the current score assigned is four reds, the game is deemed to have been won, and hence ends. The game is deemed to have been lost if it is not solved within the prescribed number of moves (typically 8); then also the game ends. While the game has not ended, the solver reacts to these scores and selects the next move, typically designed to reveal more information about the secret code and also to enhance the overall score.

### 15.1.2 Formulation of Scoring: Approach 2

Here is another way to formulate scoring. Informally, the rules of this approach are easy to state:

- Ignore the positions that fetched a red score (correct color and place)
- For each color  $C$ , let  $n_C$  = lesser of
  - code-side number of occurrences of  $C$ , and
  - solver-side number of occurrences of  $C$ .
- White-score =  $\sigma_{C \in Colors} : n_C$

For the example below, focusing on Red (R) pegs:

Pos	:	0	1	2	3
Code	:	B	R	G	R
Guess	:	R	G	R	R

We will have  $n_R = 1$ , and  $n_G = 1$ , and  $n_B = 0$ , giving a white score of 2.

**More formal version of the above idea:** Suppose  $code_{pos,col}$  be the assertion that the code has color  $col$  in position  $pos$ , and  $solve_{pos,col}$  be a similar assertion from the point of view of the solver. Then,

- Let there be  $Pos$  positions to place either the code pegs or the solver pegs. Let there be  $Col$  colors also.
- View numbers as sets. Thus,  $Pos = \{0, \dots, Pos - 1\}$  and  $Col = \{0, \dots, Col - 1\}$ .
- Let  $redPos$  denote the set of positions where a red score will be assigned:

$$redPos = \{\langle pos \rangle \mid \exists col \in Col : code_{pos,col} \wedge solve_{pos,col}\}$$

- Then  $redScore$  denotes the red score earned:  $redScore = |redPos|$
- Let  $CodeWhite_{col}$  stand for the number of code words that earn a “white score” with respect to color  $col$ .
- Let  $nonRedPos = Pos - redPos$
- $CodeWhite_{col} = \{p_c \mid p_c \in nonRedPos \wedge code_{p_c,col}\}$
- $SolveWhite_{col} = \{p_s \mid p_s \in nonRedPos \wedge solve_{p_s,col}\}$
- Note that if there is no  $col$  on the code or solver side, then  $CodeWhite_{col}$  (respectively  $SolveWhite_{col}$ ) will be the empty set.
- $whiteScore_{col} = \min(|CodeWhite_{col}|, |SolveWhite_{col}|)$
- $whiteScore = \sum_{col \in Col} whiteScore_{col}$ .

### Illustration of Scoring

Consider the scoring situation shown below:

```
Pos : 0 1 2 3
Code : B R G R
Guess: R G R R
```

Let us compute the stated quantities:

- $redPos = \{3\}$
- $nonRedPos = \{0, 1, 2\}$
- $CodeWhite_R = \{1\}$
- $SolveWhite_R = \{0, 2\}$
- The smaller of the two sets has size 1, giving a white-score of 1 for color  $redsetR$ .
- Similarly,  $G$  earns a white-score of 1, and  $B$  has a white score of 0
- This gives a total white-score of 2.

## 15.2 A Simple Mastermind Program

We will now describe a Mastermind program, beginning with user's interactions with the program, and subsequently describing the code.

### 15.2.1 Usage Scenarios

- The user begins playing a new game (see below). It begins with three wasted moves. But wait: the system has learned constraints meanwhile:

---

Play again? [Y/N] : y

```
4546 is the secret.
0000
0000
0000
0000
0000
0000
1122 :

4546 is the secret.
0000
0000
```

```

0000
0000
0000
0000
2211 :
1122 :

4546 is the secret.
0000
0000
0000
0000
0000
3333 :
2211 :
1122 :

```

By virtue of the learned constraints, the solver now earns two reds (corresponding to two of the 4s are matching). However, the 5 and 6 don't find a match:

```

4546 is the secret.
0000
0000
0000
0000
0000
4444 : rr
3333 :
2211 :
1122 :

```

OK, now we have a 4 in position 3 fetching a red, while a 4 set in position 4 by the solver earning a white:

```

4546 is the secret.
0000
0000
0000
0000
5544 : rrw
4444 : rr
3333 :
2211 :
1122 :

```

The system now earns rrw twice:

```

4546 is the secret.
0000
0000
0000
4554 : rrw
5544 : rrw
4444 : rr
3333 :
2211 :
1122 :

```

```

4546 is the secret.
0000
5454 : www
4554 : rrw
5544 : rrw
4444 : rr
3333 :
2211 :
1122 :

```

The system now earns rrr, but too late: no more moves are allowed:

```

4546 is the secret.
4545 : rrr
5454 : www
4554 : rrw
5544 : rrw
4444 : rr
3333 :
2211 :
1122 :

```

The constraints generated for each move are listed on demand:

Press <ENTER> to see constraints

```

[And(s_1 >= 1, s_1 <= 6),
 And(s_2 >= 1, s_2 <= 6),
 And(s_3 >= 1, s_3 <= 6),
 And(s_4 >= 1, s_4 <= 6),
 And(s_1 != 1, s_2 != 1, s_3 != 2, s_4 != 2),
 And(s_1 != 2, s_2 != 2, s_3 != 1, s_4 != 1),
 And(s_1 != 3, s_2 != 3, s_3 != 3, s_4 != 3),
 Or(And(s_1 == 4, s_2 == 4, s_3 != 4, s_4 != 4),
    And(s_1 == 4, s_2 != 4, s_3 == 4, s_4 != 4),
    And(s_1 == 4, s_2 != 4, s_3 != 4, s_4 == 4),
    And(s_1 != 4, s_2 == 4, s_3 == 4, s_4 != 4),
    And(s_1 != 4, s_2 != 4, s_3 == 4, s_4 == 4)),
 Or(And(s_1 == 5, s_2 == 5, s_3 != 4, s_4 != 4),
    And(s_1 == 5, s_2 != 5, s_3 == 4, s_4 != 4),
    And(s_1 == 5, s_2 != 5, s_3 != 4, s_4 == 4),
    And(s_1 != 5, s_2 == 5, s_3 == 4, s_4 != 4),
    And(s_1 != 5, s_2 == 5, s_3 != 4, s_4 == 4),
    And(s_1 != 5, s_2 != 5, s_3 == 4, s_4 == 4)),
 Or(And(s_1 == 4, s_2 == 5, s_3 != 5, s_4 != 4),
    And(s_1 == 4, s_2 != 5, s_3 == 5, s_4 != 4),
    And(s_1 == 4, s_2 != 5, s_3 != 5, s_4 == 4),
    And(s_1 != 4, s_2 == 5, s_3 == 5, s_4 != 4),
    And(s_1 != 4, s_2 == 5, s_3 != 5, s_4 == 4),
    And(s_1 != 4, s_2 != 5, s_3 == 5, s_4 == 4)),
 Or(And(s_1 == 4, s_2 == 5, s_3 == 4, s_4 != 5),
    And(s_1 == 4, s_2 == 5, s_3 != 4, s_4 == 5),
    And(s_1 == 4, s_2 != 5, s_3 == 4, s_4 == 5),
    And(s_1 != 4, s_2 == 5, s_3 == 4, s_4 == 5),
    And(s_1 != 4, s_2 == 5, s_3 != 4, s_4 == 5),
    And(s_1 != 4, s_2 != 5, s_3 == 4, s_4 == 5))]

```

The user chooses to play again, with better results:

```
Play again? [Y/N]: y
```

---

```
6616 is the secret.
0000
0000
0000
0000
0000
0000
0000
1122 : w
```

In the play of 2211 below, notice that the play earns only one r:

```
6616 is the secret.
0000
0000
0000
0000
0000
0000
0000
2211 : r
1122 : w
```

This is because of the following sets:

- $redSet = \{3\}$ ;  $redScore = 1$ .
- $whiteSet = \{\}$ ;  $whiteScore = 0$ . This is because while  $code_{3,1} \wedge solve_{4,1}$ , we do have  $3 \in redSet$ . View it as: “That code peg has already scored a solver peg at position 3. Hence it may not be used again to score another solver peg.”

Continuing on, we get these moves:

```
6616 is the secret.
0000
0000
0000
0000
0000
0000
3233 :
2211 : r
1122 : w
```

```
6616 is the secret.
0000
0000
0000
0000
0000
4341 : w
3233 :
2211 : r
1122 : w
```

```

6616 is the secret.
0000
0000
0000
5414 : r
4341 : w
3233 :
2211 : r
1122 : w

6616 is the secret.
0000
0000
6515 : rr
5414 : r
4341 : w
3233 :
2211 : r
1122 : w

Congratulations, you won!
The secret is 6616
0000
6616 : rrrr
6515 : rr
5414 : r
4341 : w
3233 :
2211 : r
1122 : w

```

The constraints are, as before, displayed on demand:

Press <ENTER> to see constraints

```

[And(s_1 >= 1, s_1 <= 6),
 And(s_2 >= 1, s_2 <= 6),
 And(s_3 >= 1, s_3 <= 6),
 And(s_4 >= 1, s_4 <= 6),
 And(s_1 != 1, s_2 != 1, s_3 != 2, s_4 != 2),
 Or(And(s_1 == 2, s_2 != 2, s_3 != 1, s_4 != 1),
    And(s_1 != 2, s_2 == 2, s_3 != 1, s_4 != 1),
    And(s_1 != 2, s_2 != 2, s_3 == 1, s_4 != 1),
    And(s_1 != 2, s_2 != 2, s_3 != 1, s_4 == 1)),
 And(s_1 != 3, s_2 != 2, s_3 != 3, s_4 != 3),
 And(s_1 != 4, s_2 != 3, s_3 != 4, s_4 != 1),
 Or(And(s_1 == 5, s_2 != 4, s_3 != 1, s_4 != 4),
    And(s_1 != 5, s_2 == 4, s_3 != 1, s_4 != 4),
    And(s_1 != 5, s_2 != 4, s_3 == 1, s_4 != 4),
    And(s_1 != 5, s_2 != 4, s_3 != 1, s_4 == 4)),
 Or(And(s_1 == 6, s_2 == 5, s_3 != 1, s_4 != 5),
    And(s_1 == 6, s_2 != 5, s_3 == 1, s_4 != 5),
    And(s_1 == 6, s_2 != 5, s_3 != 1, s_4 == 5),
    And(s_1 != 6, s_2 == 5, s_3 == 1, s_4 != 5),
    And(s_1 != 6, s_2 == 5, s_3 != 1, s_4 == 5),
    And(s_1 != 6, s_2 != 5, s_3 == 1, s_4 == 5)]

```

```
Play again? [Y/N]: n
```

---

One of the goals of your project is to make this game win more often. You will see opportunities to make the code do so when you study the code—which is what we discuss next.

### 15.2.2 Mastermind: Preliminary Implementation

We will now walk through the simple mastermind program whose actions were discussed above.

```
#!/usr/bin/env python

# == Author : Bruce Bolick == with mild clean-up done by Ganesh ==
# ==

# Mastermind is NP-complete with n pegs per row and 2 colors.
#
# That is, a given a partially played game and the scoring so far,
# whether there exists a secret key such that the partially played game
# has this scoring is the decision problem.
#
# Note that instead of the red scoring peg (right peg color, right place), often
# a black scoring peg is used, with the same meaning. Here, we stick with
# "red". In the scoring table, you will see "r" and "w".
#
# From Wikipedia:
# ---
# Given a set of guesses and the number of red and white pegs
# scored for each guess, is there at least one secret pattern that
# generates those exact scores? (If not, the codemaker must have
# incorrectly scored at least one guess.)

# This alpha version of the game only considers red pegs.
# (where the right digit is in the right location.) It still wins most
# of the time.
```

First, we import some of the necessary packages. The main function is `playOnce()`.

```
import random
from z3 import *

# Build and play the game once
#
def playOnce():
    """Builds a solver, and plays one round with the simple strategy:
    that of adding red-specific constraints.
    """
    #the list s is the actual secret code
```

```

s = []
for i in range(4):
    s.append(random.randint(1,6))

#string version of the actual secret code
s_string = str(s[0])+str(s[1])+str(s[2])+str(s[3])

#the matrix S is a 1x4 matrix which represents the computer's solution
S = [ Int("s_%s" % (i+1)) for i in range(4) ]

#define solver and initial constraint
# each cell contains a value in {1, ..., 6}
solver = Solver()

# == First, add default constraints for each position ==
# == Only the allowed colors 1..6 may be placed ==
# ==
range_c = [ And(1 <= S[i], S[i] <= 6) for i in range(4) ]
solver.add(range_c)

#list of guesses
#prime first guess with 1,1,2,2
g = [[1,1,2,2],[0,0,0,0],[0,0,0,0],[0,0,0,0],
      [0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]

# == used while printing the board
g_string = ["1122", "0000", "0000", "0000",
            "0000", "0000", "0000", "0000"]

#== This is the main loop ==
# It runs through turns 1 through 8
# If a solution is not found, it exits
# If found, the user may see the constraints as well
# ==
for i in range(0, 8):
    # == red and white counts are accumulated in here
    # == after each turn
    red = 0
    white = 0

    #for all guesses except the 0'th, use smt solver for guess
    if(i > 0):
        #use smt solver
        solver.check()
        m = solver.model()
        r = [ m.evaluate(S[j]) for j in range(4) ]

        #== convert each from z3.IntNumRef to string to int
        g[i][0] = int(r[0].as_string())
        g[i][1] = int(r[1].as_string())
        g[i][2] = int(r[2].as_string())
        g[i][3] = int(r[3].as_string())

        #== string version of this guess
        g_string[i] = r[0].as_string() + r[1].as_string() + r[2].as_string() + r[3].as_string()

    #deep copy of this turn and secret

```

```

#because we will be altering as we check
thisturn = g[i][:]
thissecret = s[:]

#check for red match.  0 and -1 prevent duplicate reds/whites
for j in range(4):
    m = int(thissecret[j])
    n = int(thisturn[j])
    if(m==n):
        red = red + 1
        thisturn[j] = 0
        thissecret[j] = -1

    if(red == 4):
        won = True
        print("Congratulations, you won!")
        #update string version of guess with white/red marks
        g_string[i] = g_string[i] + " : "

        for j in range(red):
            g_string[i] = g_string[i] + "r"

        print("The secret is " + s_string)
        break
    #--- break from for i in range(0, 8):

#check for white match. 'x' and 'y' prevent dups
for j in range(4):
    for k in range(4):
        if(thisturn[j] == thissecret[k]):
            white = white + 1
            thisturn[j] = 0
            thissecret[k] = -1
            break

    #update string version of guess with white/red marks
    g_string[i] = g_string[i] + " : "

    for j in range(red):
        g_string[i] = g_string[i] + "r"

    for j in range(white):
        g_string[i] = g_string[i] + "w"

    #--- GENERATE CONSTRAINTS AND ADD TO SOLVER ---
    solver.add(GenConstraints(red, S, g, i))

    #--- White constraint generation part is elided; you'll be adding this

    #print turns as of now
    print(s_string + " is the secret.")
    for j in range(0,8):
        print(g_string[7-j])

    raw_input()
    #==== end of for i in range(0, 8) ====

```

```

==== exit point after break, or upon loss (when all 8 turns are exhausted) ===
for j in range(0,8):
    print(g_string[7-j])

print("Press <ENTER> to see constraints")
raw_input()
print(solver)

```

Function GenConstraints takes

- red, the number of reds (the only basis currently for generating constraints),
- S, the list of integer variables which are being constrained,
- g, a list of lists that records the solver actions so far, and
- i, the current iteration number (index into g) that actually defines the rank of the current play.

Clearly, you will be working on improving GenConstraints in your project, as will be described in §15.3:

```

-----
# Generates constraints for each game-board situation
# You'll be cleaning this up using the combinations package

#
def GenConstraints(red, S, g, i):
    """Perform case analysis over the number of reds
    and return the constraint list.
    """
    #add constraints for red pegs
    if(red==0):
        #      => ~r1 ^ ~r2 ^ ~r3 ^ ~r4
        return [And(S[0]!=g[i][0], S[1]!=g[i][1], S[2]!=g[i][2], S[3]!=g[i][3])]

    if(red==1):
        # r      => r1 ^ ~r2 ^ ~r3 ^ ~r4
        #      v r2 ^ ~r1 ^ ~r3 ^ ~r4
        #      v r3 ^ ~r1 ^ ~r2 ^ ~r4
        #      v r4 ^ ~r1 ^ ~r2 ^ ~r3
        return [Or(
            And(S[0]==g[i][0], S[1]!=g[i][1],
                S[2]!=g[i][2], S[3]!=g[i][3]),
            And(S[0]!=g[i][0], S[1]==g[i][1],
                S[2]!=g[i][2], S[3]!=g[i][3]),
            And(S[0]!=g[i][0], S[1]!=g[i][1],
                S[2]==g[i][2], S[3]!=g[i][3]),
            And(S[0]!=g[i][0], S[1]!=g[i][1],
                S[2]!=g[i][2], S[3]==g[i][3])
        )
    ]

    if(red==2):
        # rr      => r1 ^ r2 ^ ~r3 ^ ~r4

```

```

#      v r1 ^ r3 ^ ~r2 ^ ~r4
#      v r1 ^ r4 ^ ~r2 ^ ~r3
#      v r2 ^ r3 ^ ~r1 ^ ~r4
#      v r2 ^ r4 ^ ~r1 ^ ~r3
#      v r3 ^ r4 ^ ~r1 ^ ~r2
return [Or(
    And(S[0]==g[i][0], S[1]==g[i][1],
        S[2]!=g[i][2], S[3]!=g[i][3]),
    And(S[0]==g[i][0], S[1]!=g[i][1],
        S[2]==g[i][2], S[3]!=g[i][3]),
    And(S[0]==g[i][0], S[1]!=g[i][1],
        S[2]!=g[i][2], S[3]==g[i][3]),
    And(S[0]!=g[i][0], S[1]==g[i][1],
        S[2]==g[i][2], S[3]!=g[i][3]),
    And(S[0]!=g[i][0], S[1]==g[i][1],
        S[2]!=g[i][2], S[3]==g[i][3]),
    And(S[0]!=g[i][0], S[1]!=g[i][1],
        S[2]==g[i][2], S[3]==g[i][3])
)
]
]

if(red==3):
    # rrr => r1 ^ r2 ^ r3 ^ ~r4
    #      v r1 ^ r2 ^ r4 ^ ~r3
    #      v r1 ^ r3 ^ r4 ^ ~r2
    #      v r2 ^ r3 ^ r4 ^ ~r1
return [Or(
    And(S[0]==g[i][0], S[1]==g[i][1],
        S[2]==g[i][2], S[3]!=g[i][3]),
    And(S[0]==g[i][0], S[1]==g[i][1],
        S[2]!=g[i][2], S[3]==g[i][3]),
    And(S[0]==g[i][0], S[1]!=g[i][1],
        S[2]==g[i][2], S[3]==g[i][3]),
    And(S[0]!=g[i][0], S[1]==g[i][1],
        S[2]==g[i][2], S[3]==g[i][3])
)
]
]

```

The main loop is straightforward:

```

--- main ---
#
def main(argv=None):
    play_again = "y"
    while (play_again in ["y", "Y"]):
        playOnce()
        play_again = raw_input("Play again? [Y/N]: ")

if __name__ == "__main__":
    sys.exit(main())

== end of file ==

```

## 15.3 Project Specification

The aim of your project is twofold:

1. Add constraint generation corresponding to whites also (at present, the number of whites are being counted, but there is no constraint being generated with respect to them).
2. Generalize the game as follows:
  - (a) for arbitrary sizes, as defined by these constants: Npegs, Ncolors, MaxGuesses;
  - (b) incorporate the constraints using the `itertools` package of Python. This package provides the permutations and combinations functions. If you notice carefully, the body of the `GenConstraints` functions is doing precisely this, but in a hard-wired manner. This is clearly untenable for generalized game/board/peg/color sizes.

To see how the combinations package can help you, see its behavior:

```
>>> from itertools import *
>>> list(combinations([1,2,3,4], 1))
[(1,), (2,), (3,), (4,)]
>>> list(combinations([1,2,3,4], 2))
[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
>>> list(combinations([1,2,3,4], 3))
[(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)]
>>> list(combinations([1,2,3,4], 4))
[(1, 2, 3, 4)]
```

Now, if you study the body of `GenConstraints`, you will hopefully see a similar pattern in terms of how red scores are turned into And/Or constraints.

### 15.3.1 Extra Credit Items

Once you get the basic constraint mechanisms working, you may make a number of improvements:

1. Allowing the user also to play, in addition to the computer playing itself. Ideally, any step must be playable by the user or the computer.
2. If the person's play is *not a wise* move, the computer ought to explain why this is so. One way is to conjoin the constraints generated by the computer (so far) and conjoin it with the move the user is about to attempt. If this move will not improve the score *and* generate no

new constraints, then warn the user. Hint: The constraints generated so far end up implying the constraints that the users' new move will generate.

3. Score a user's game in terms of:
  - (a) the total number of moves (fewer is better)
  - (b) the number of wasted moves (fewer is better)
4. Experiment with heuristics. At this point, the constraint generation simply relies on the z3 solver to give you new cell values. There are multiple choices for an effective next move. Could you do something different compared to the default z3 solver values returned at line `solver.check()`? Hint: you may need to generate some auxiliary constraints before you call the solver again.
5. (Last but not least) Provide some kind of a graphical display for enhancing the user experience of playing the game. Note that while improved graphics is going to earn you extra points, merely working on the graphics without improving the basic constraint generation capability will not earn you very many points. That said, the TAs will be very happy to help you with ideas leading to improved graphics.

# Chapter 16

## Recursion and Induction

Recursion is one of the fundamental programming approaches of computer science. Recursive programs are an expression of one's ability to break down a given problem into simpler problems. There will also be extremely simple instances of the problem that are solved directly.

Induction is one of the fundamental proof techniques of computer science. Inductive proofs are an expression of one's ability to break down the proof into simpler proofs. There will also be extremely simple instances of the proof that are solved directly.

The tight connection between the above two paragraphs is no accident. Recursion and induction are two sides of the same coin. In Wand's book [16], this connection is brought out at every juncture. This book is an attempt to bring some of those ideas into this class.

### Example 1: Factorial

Let us take an example of problem decomposition to compute the factorial of a number:

- factorial is a function with signature  $Nat \rightarrow Nat$
- The factorial of  $n$  is nothing but  $n$  times the factorial of  $n - 1$ .
- The factorial of 0 is 1.

The above recipe can be encoded as a recursive program of Figure 16.1:

The above recipe can be turned into an inductive proof. We assume that the mathematical function  $!$  is predefined, and obtains the factorial of any number.

- Basis case:  $n = 0 : 0! \equiv 1$  which is consistent with the program behavior.

```
def fac(n):
    """ You may feed n >= 0, or else the code will loop.
       If you want to prevent looping, add another test.
    """
    if (n == 0):
        return 1
    else:
        return n * fac(n-1)
```

Figure 16.1: A Factorial Program

- Inductive case: Assume that  $fac(n - 1) = n!$ . Prove that  $fac(n) = n!$ . By tracing the program behavior, we find this also to be in agreement.

## Example 2: String Concatenation

Let us take an example of problem decomposition to concatenate two strings:

- The concatenation of two strings  $s_1$  and  $s_2$  is obtained by concatenating the first character of  $s_1$  with the concatenation of the *rest* of  $s_1$  and  $s_2$ .
- The concatenation of an empty string and  $s_2$  is  $s_2$ .

The above recipe can be encoded as a recursive program of Figure 16.2:

```
def append_strings(s1,s2):
    if (s1==""):
        return s2
    else:
        return s1[0] + append_strings(s1[1:], s2)
```

Figure 16.2: A String Concatenation Program

The above recipe can be turned into an inductive proof.

- Basis case:  $s_1$  is empty: the program behavior of `append_strings` matches.
- Inductive case: Assume that `append_strings` works for  $s_1$  of upto and including length  $(n - 1)$ , by successfully appending strings. Now for  $s_1$  of length  $n$ , it works correctly, as we find by tracing the program behavior.

### Example 3: Shuffling Decks of Cards

Let us take an example of problem decomposition – now to compute all possible shuffles of two decks of cards:

- The shuffle of two decks of cards is obtained by
  1. setting apart one card from either deck (two cases here),
  2. shuffling the rest of the cards, and finally
  3. restoring the card set apart as the first card.
- The shuffle returns one deck if the other deck is empty.

The above recipe can be encoded as a recursive program of Figure 16.3:

```
def shuffles(L1,L2):
    if (L1==[]):
        return [L2]
    else:
        if (L2==[]):
            return [L1]
        else:
            return
                list(map(lambda x:\n                    ([L1[0]] + x), shuffles(L1[1:], L2)))
            +
            list(map(lambda x:\n                    ([L2[0]] + x), shuffles(L1, L2[1:])))
```

Figure 16.3: Shuffling Two Decks of Cards

Here, the following part of the code merits more explanation, and we do so by interlacing comments as in Figure 16.4:

The above recipe can be turned into an inductive proof.

- Basis case: Argue for the shuffles being trivial.
- Inductive case: Assume that we are given decks  $(L_1, L_2)$ . Assume that shuffling works for all decks  $(L'_1, L'_2)$  that are *below* the size of  $(L_1, L_2)$ . Tracing the code, we find that it works for the whole.

## 16.1 What is Induction? How many kinds?

Induction is a way to prove  $\forall x : \text{good}(x)$ , or in general  $\forall x : P(x)$  where  $x$ 's domain is some set  $S$ . Typically  $S$  is  $\text{Nat}$ , but as we saw with cards, we may need to trace the downward growth of *two* quantities, such as  $(L_1, L_2)$

```

list(map(lambda x:\n    ([L1[0]] + x), shuffles(L1[1:], L2)))\n\n    ^\n    |-- Take the first card of the first deck; set it apart\n\n    ^\n    |-- Run over every shuffle "x" that is in the shuffles\n        of the "rest" of L1 and L2\n\n    ^\n    |-- Place L1[0] at the head of each of these shuffles\n\n    +\n\n    ^\n    |-- Add to the above construction..\n\nlist(map(lambda x:\n    ([L2[0]] + x), shuffles(L1, L2[1:])))\n\n    ^\n    |-- what we obtain by setting apart from L2 similarly.

```

Figure 16.4: Shuffling Code Explanation

shrinking into  $(L1[1:], L2)$  or  $(L1, L2[1:])$ . Also, two decks don't form a total order. They in fact form a partial order; see Figure 16.5. We illustrate it for decks ab and 12 so that you may visually discern the shuffles. (To run under Python, I strictly should write `['a', 'b']` and `[1, 2]`, but for brevity I am omitting the quotes and brackets.)

The idea of carrying out induction over a well-founded partial order is illustrated in Figure 16.5. Here are the details:

- A partial ordered set  $S$  is one that is similar to that shown in Figure 16.5. The elements of the set are the pairs themselves, i.e. ab, 12, ab, "", etc.
- Although you may know a partial order to be *reflexive*, *anti-symmetric*, and *transitive*, you can simply carry one idea: for every element, there are a finite number of elements “below” it.
- **Well-founded** means **the whole set has a finite set of minimal elements**.
- Really speaking, we are setting aside *reflexivity*. We are inducting over

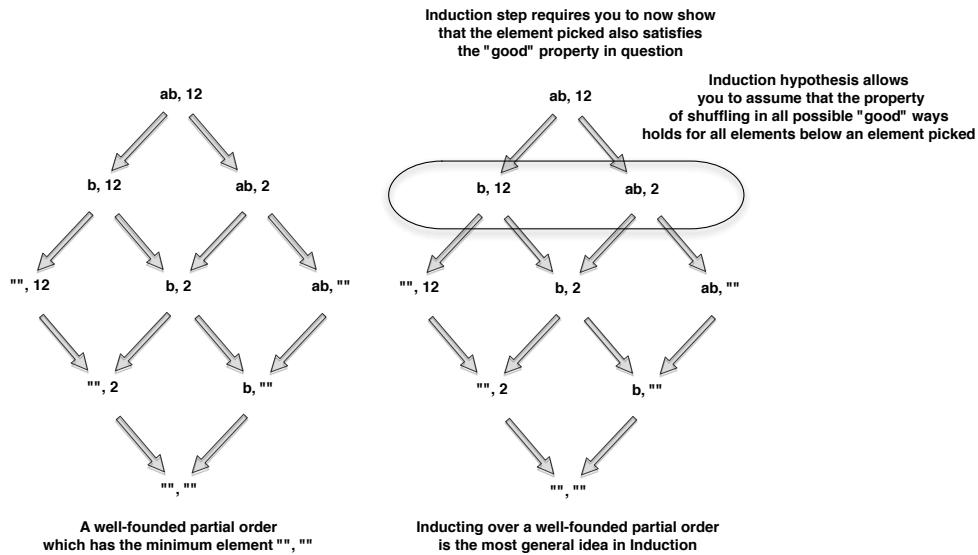


Figure 16.5: Induction over a partial order

*irreflexive and well founded* partial orders.

- Pick the property to be proved. In this case it is “shuffling works” for all decks.
- Show it for all the minimal elements. Here, show it for the basis case “”, “”.
- Pick a *general* element. In this example, we illustrate it for ab, 12 but obviously in a proof, we pick a general element.
- Assume that the property holds for all the elements below it. See the elements in the oval.
- Prove that the property now holds for the element picked.

### 16.1.1 Arithmetic Induction

For a property  $P$  and set  $S$  with minimal element  $b$ ,

- Show that  $P(b)$  is true.
- Assuming  $P(n - 1)$  to be true, show that  $P(n)$  is true.

The standard choices are  $b = 0$ ,  $b = 1$ , etc., depending on the property to be shown. Here are three examples:

- Prove that summation of all numbers below  $n \in \text{Nat}$  is  $n(n + 1)/2$ .

- Here,  $b = 0$  is a natural choice.
- Prove that summation of  $1 \dots n$  is  $n(n + 1)/2$ .
  - Here,  $b = 1$  is a natural choice.
- Prove that summation of the first  $n$  odd numbers from  $1 \dots (2n - 1)$  is  $n^2$ .
  - Here,  $b = 1$  is a natural choice.

In general, the set of elements is one of these:

- $n = 0, 1, 2, \dots$  as in the first example above.
- $n = 1, 2, 3, \dots$  as in the second example above.
- Generated in terms of  $n = 1, 2, 3, \dots$  as in the third example above.

Thus, while doing arithmetic induction, the typical basis elements are 0 or 1.

### 16.1.2 Complete Induction

For a property  $P$  and set  $S$  with minimal element  $b = 0$  (chosen for specific illustration here; the idea is similar for any  $b$ ), complete induction works as follows:

- Pick a general element  $n$ .
- Assuming  $P(x)$  to be true for all  $x < n$ , show that  $P(n)$  is true.

Where did the basis case go? When applying complete induction, you will find that for the general element  $n = 0$ , you will be addressing the basis case, because there is no  $x < 0$  in  $\text{Nat}$ .

### 16.1.3 Structural Induction

This is a good way to think about induction when items in question are somehow “structured.” Here are examples:

- If you look at trees, we can induct along the construction of trees.
- Jeff Erickson’s notes illustrate this idea in case of Triominos. We are “structuring” the triominos in a systematic way.

In reality, structural induction is no different from arithmetic induction (or Noetherian induction, if partial orders are involved).

### 16.1.4 Subgoal Induction

Subgoal induction is a way to think of inductive proofs when the program recursion guides your recursive thinking. Again, this is really no different

in terms of the core idea. All these words fly around (“structural,” “subgoal,” etc.), and all boil down to instances of Noetherian induction. See additional details in §16.2.

## 16.2 How to present inductive proofs?

### 16.2.1 Case 1: When you have a program to stare at

While referring to a program and its recursive make-up, adopt the subgoal induction approach, and present the proof as follows:

- Refer to the case(s) when the program terminates. For that, prove the basis case.
- Refer to the inductive (recursive) cases. State the induction hypothesis with respect to them.
- Refer to the “main” recursive call. Prove that the assertion you desire holds for the main call.

You may wish to try applying this inductive proof method to our recursive definition of Primes, introduced in §6.3.

### 16.2.2 Case 2: When you are doing it just over “data”

Often you won’t have a recursive program serving you as a template for recursion. It may be a word problem similar to the ones in your assignment. Even in this case, you will pretty much follow the recipe suggested so far:

- State and prove the basis case.
- Assume the property to hold for “ $n - 1$ ”.
- Prove the property holding for “ $n$ ”.



# Chapter 17

## Permutations and Combinations

**Permutations:** The topic of *permutations* is best introduced through an example. In a room with  $N$  equally desirable seats,  $N$  people can be seated in  $N!$  ways (the first person can be chosen in  $N$  ways, then the second person in  $(N - 1)$  ways, etc.). Here ! is “factorial,” definable in many ways:

```
def fac(n):
    return 1 if (n <= 0) else n * fac(n-1)

>>> mulf = lambda x,y:x*y

>>> reduce(mulf,range(1,6))
120

def fac_alt(n):
    return reduce(mulf, range(1,n+1))

>>> fac_alt(5)
120
```

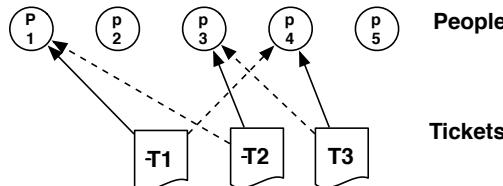


Figure 17.1: With five people and three tickets,  $3!$  ticket swaps are possible for each assignment of people chosen to have tickets in the first place

**Combinations:** Suppose there are  $N$  Justin Bieber fans and, through a lottery,  $k$  have been chosen as lucky winners of front row seats. How many ways are there to choose them?

- Well, if  $N = k$ , then there is really no choice; all are eligible to attend. More precisely, there is only one choice. More precisely,  $\binom{N}{N}$  (“ $N$  choose  $N$ ”) is 1.
- Suppose  $N = 5$  and  $k = 4$ , then you can choose the lucky four or the unlucky one; both are equal in effect. Clearly, there are five choices.  $\binom{5}{4} = \binom{5}{1} = 5$ .
- Suppose  $N = 5$  and  $k = 0$  (the lottery was a hoax); then we have  $\binom{5}{0}$ . We will see that this is 1.

How do we determine a general formula? Well, we can give the first ticket  $N$  ways, the second ticket  $(N - 1)$  ways, all the way to giving the  $k$ -th ticket  $(N - k + 1)$  ways. Thus there are  $N.(N - 1).(N - 2) \dots (N - k + 1)$  ways of giving out tickets, if we are mindful of who gets which ticket number. But then, the Bieber fans don’t care; *they just want some ticket!* Thus, we must eliminate the “excess counting” by dividing out with  $k!$ .

For extra clarity, see Figure 17.1 for a situation where for some choice of ticket holders, the ticket swaps are shown. In other words, if we go by the formula

$$N.(N - 1).(N - 2) \dots (N - k + 1),$$

we are going to have buried within it the two “arrow situations” being counted as two *separate* counts. There are thus  $k!$  “useless arrow situations.” Thus we get  $N.(N - 1) \dots (N - k + 1)/(k!)$ , or the same as  $N!/(k!).(N - k)!$ . In other words, reading

$$\binom{N}{k} = N.(N - 1) \dots (N - k + 1)/(k!) = N!/(k!).(N - k)!$$

In Figure 17.1, the solid lines represent one of the 5.4.3 initial assignments. But then the dotted lines come and show one of the  $3!$  ticket swaps possible; *we don’t need to be counting each of these swapped situations as separate.* That is why we must divide 5.4.3 by  $3!$ , giving us 10 ways to choose 3 ticket holders out of 5 eligible people. In other words, the number of choices of 3 items out of 5 items is  $\binom{5}{3}$ , or 10.

**Putting it all together:** Let us now let’s merge all the definitions so far into the following series of definitions that present how permutations and combinations can be evaluated in Python:

```

def facNK(n,k):
    """Assume k <= n. Multiply n, n-1, ... , k"""
    return k if (n <= k) else n * facNK(n-1, k)

def fac(n):
    return 1 if (n==0) else facNK(n, 1)

>>> fac(5)
120

def comb(n,k):
    """Assume k <= n."""
    return 1 if (n==k) else facNK(n,k+1)/fac(n-k)

>>> comb(5,2)
10

>>> comb(5,3)
10

```

## 17.1 Illustrations on Using Venn Diagrams

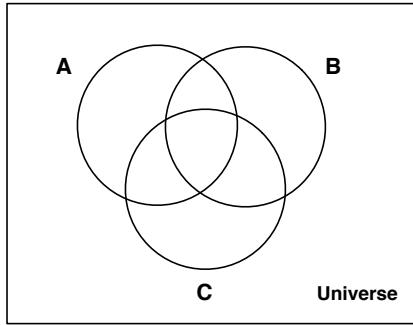


Figure 17.2: The Familiar Venn Diagram of 3 sets

Venn diagrams are one of the most widely used of notations to depict sets and their inclusion relationships. Usually one draws the “universal set” as a rectangle, and within it depicts closed curves representing various sets. I am sure you have seen simple venn diagrams showing three circles representing three sets A, B, and C, and showing all the regions defined by the sets (e.g., Figure 5.2 on Page 247) namely: the eight sets:  $A \cap B \cap C$  (points in all three

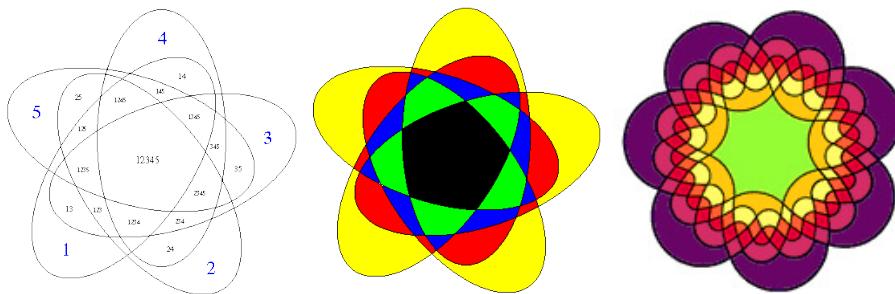


Figure 17.3: Venn Diagrams of order 5 (left); of order 5 with regions colored (middle); and order 7 (right). Images courtesy of <http://mathworld.wolfram.com/VennDiagram.html> and <http://www.theory.csc.uvic.ca/~cos/inf/comb/SubsetInfo.html#Venn>.

sets),  $A \cap B$ ,  $B \cap C$ , and  $A \cap C$  (points in any two sets chosen among the three), and then  $A$ ,  $B$ , and  $C$  (points in the three individual sets), and finally  $\emptyset$  (points in no set at all—shown outside of the circles).

Venn diagrams are schematic diagrams used in logic theory to depict collections of sets and represent their relationships [15, 18]. More formally, an order- $N$  Venn diagram is a collection of simple closed curves in the plane such that

1. The curves partition the plane into connected regions, and
2. Each subset  $S$  of  $\{1, 2, \dots, N\}$  corresponds to a unique region formed by the intersection of the interiors of the curves in  $S$  [13].

Venn diagrams involving five and seven sets are beautifully depicted in these websites, and also the associated combinatorics is worked out. Two illustrations from the latter site are shown in Figure 17.3 on Page 248, where the colors represent the number of regions included inside the closed curves.

### 17.1.1 Venn Diagram Regions and the Binomial Theorem

Binomial theorem, invented by Sir Issac Newton, is quite a fundamental result in mathematics. According to it:

$$(x + y)^N = \sum_{k=0}^N \binom{N}{k} x^{N-k} y^k$$

One can prove the Binomial Theorem using either combinatorics or using induction. Here are some illustrations:

- $(x+y)^2 = x^2 + 2xy + y^2$
- $(x+y)^3 = x^3 + 3x^2y + 3xy^2 + y^3$

**Venn Diagram Regions using the Binomial Theorem:** Since there are  $\binom{N}{k}$  ways to pick  $k$  members from a total of  $N$ , the number of regions  $V_N$  in an order  $N$  Venn diagram is

$$V_N = \sum_{k=0}^N \binom{N}{k} = 2^N$$

(where the region outside the diagram is included in the count, as it corresponds to  $\binom{N}{0}$  which is 1).

**Note:** We obtain the connection with the Binomial Theorem because one can view  $2^N$  as  $(1+1)^N$  and expand using the Binomial theorem, resulting in the formula for  $V_N$  (since “ $x$ ” and “ $y$ ” are both 1 now).

### 17.1.2 Alternative Derivation

There is another way to arrive at the  $2^N$ : we want points included in some combination of sets. Well, a point may say: “I’ll use an  $N$ -bit vector, and turn on my  $i$ -th bit if I want to be included in the  $i$ -th set.” Viewed this way, there can be  $2^N$  connected spaces within which each point can situate itself.

**Evaluating  $\sum_{k=0}^N \binom{N}{k}$  in Python:** We can evaluate the summation defining  $V_N$  on a simple example and check its correctness. Let us pick  $N = 8$ ; then, according to the formula, we expect the answer to be 256:

```
>>> [comb(8,k) for k in range(8+1)]
[1, 8, 28, 56, 70, 56, 28, 8, 1]

>>> reduce(addf, [comb(8,k) for k in range(8+1)])
256
```

## 17.2 Illustration in the Context of Powersets

Suppose we have a set of elements  $S = \{0, 1, 2\}$  and we put an on/off switch above each element. Suppose we generate new sets by playing with the

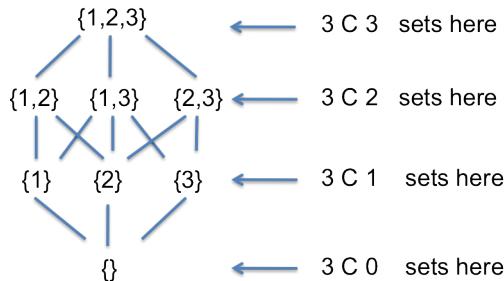


Figure 17.4: Powerset as a Lattice

switches: when all switches are off, we generate  $\{\}$ ; next with the switch associated with 2 and 0 on, we generate  $\{0, 2\}$ ; and so on. The number of these sets generated is  $2^3$  (all Boolean combinations of the switches). The set of all these is the *powerset*.

We can count the number of sets in another way also:  $\sum_{k=0}^N \binom{N}{k} = 2^N$ . In Figure 17.4, we perform this summation for  $N = 3$ , giving us the eight (8) subsets of  $\{1, 2, 3\}$ .

Note that powerset of  $S$  is the set of all its subsets (not merely proper subsets, but *all subsets*). This is why  $\{\}$  has a powerset, which equals  $\{\{\}\}$ .

The way the Powerset algorithm works is easy to explain with respect to the structure of the recursion in Figure 17.5:

- The powerset of the empty set  $\{\}$  is  $\{\{\}\}$  because we are supposed to return the set of subsets of  $\{\}$ ; and there is only one subset for  $\{\}$ , which is itself.

```
L=list(S)
if L==[] :
    return([[]])
```

- For a non-empty set, the powerset is calculated as follows:
  - First, calculate the powerset of the *rest* of the set:
 

```
else:
            pow_rest0 = pow(L[1:])
```
  - Then calculate the set obtained by pasting the first element of the original set onto every set in *pow\_rest0*:
 

```
pow_rest1 = list(map(lambda ls: [L[0]]+ls, pow_rest0))
```
  - Finally, compute a set of sets, containing all the sets within *pow\_rest0* and *pow\_rest1*:

```

def pow(S):
    """Powerset of a set L. Since sets/lists are unhashable,
    we convert the set to a list, perform the powerset operations,
    leaving the result as a list (can't convert back to a set).
    pow(set(['ab', 'bc'])) --> [['ab', 'bc'], ['bc'], ['ab'], []]
    """
    L=list(S)
    if L==[]:
        return([[]])
    else:
        pow_rest0 = pow(L[1:])
        pow_rest1 = list(map(lambda ls: [L[0]]+ls, pow_rest0))
        return(pow_rest0 + pow_rest1)
    ----
>>> pow
<function pow at 0x026E1FB0>
>>> pow({1,2,3})
[[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]
>>> pow({})
[]
>>> pow({'hi','there',5})
[[], [5], ['there'], ['there', 5], ['hi'], ['hi', 5], ['hi', 'there'], ['hi', 'there', 5]]
>>> len(pow(range(1)))
2
>>> len(pow(range(2)))
4
>>> len(pow(range(4)))
16
>>> len(pow(range(10)))
1024
>>> len(pow(range(20)))
1048576

```

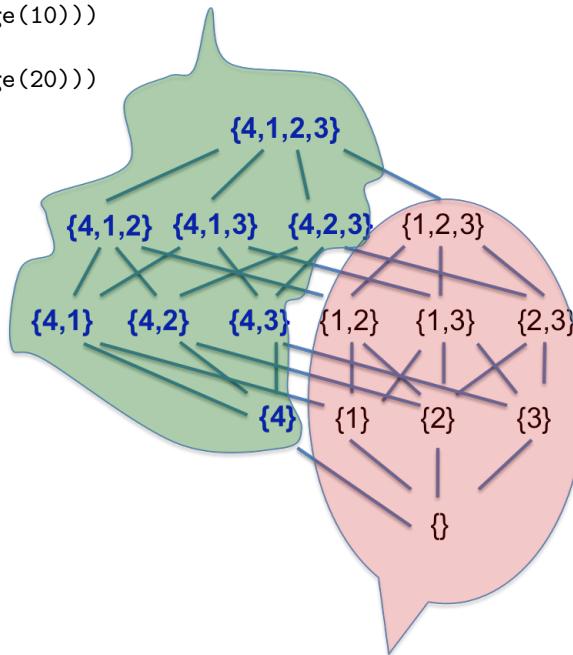


Figure 17.5: The Powerset function, and how it recurses

```
return(pow_rest0 + pow_rest1)
```

# Chapter 18

## Probability

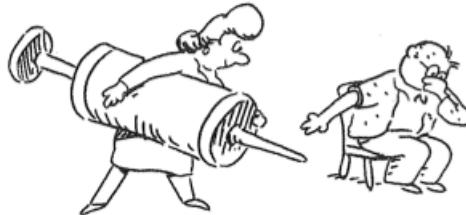
Probability theory allows you to study real-world situations that may appear subject to chance, but still can be precisely analyzed to draw crucial life-and-death conclusions. It is one of the most important of areas in computer science, especially in this era of “big data.” One of the best explanations of Probability and Statistics that I know of is in “Cartoon Guide to Statistics” by Larry Gonick and Woollcott Smith, Harper Publishing Co, 1993. They explain how this area originated from attempts to analyze gambling.

Probability theory is of fundamental importance in modern times, and often studied along with Statistics which pertains to the organization and disambiguation of data. For those taking CS 2100, probability theory also serves as a glue that binds together a significant amount of the material taught to you so far. We also get to have some fun bringing these ideas to practice (see Figure 18.1 for the sheer joy of reading the Gonick et al. book, and relating it to practice). We also refer to Lehman et al. for an in-depth understanding of the topics, and also for many really insightful examples.

It would be motivating to see the kinds of problems we will be solving with our knowledge of probability.

- A deck of cards has four suites with 13 cards each. A *full house* is three cards of one kind and two of another – e.g., QQ77, 44433, 222KK, etc. Here, QQ77 means three queens and two sevens; 44433 means three fours and two threes; and 222KK means three 2s and two kings. What is the probability of a random draw of five cards turning out to be a full suit?
- Refer to the section on Bayes Theorem in the Gonick cartoons (around Page 25 of 27 in your handout). Suppose one in 1000 people have a

WHAT'S THE PHYSICIAN TO DO? JOE BAYES ADVISES HER NOT TO START TREATMENT ON THE BASIS OF THIS TEST ALONE. THE TEST DOES PROVIDE INFORMATION, HOWEVER: WITH A POSITIVE TEST THE PATIENT'S CHANCE OF HAVING THE DISEASE INCREASED FROM 1 IN 1000 TO 1 IN 23. THE DOCTOR FOLLOWS UP WITH MORE TESTS.



JOE BAYES COLLECTS HIS CONSULTING CHECK BEFORE ADMITTING THAT ALL THOSE STEPS HE WENT THROUGH CAN BE COMPRESSED INTO THE SINGLE FORMULA CALLED BAYES THEOREM:

$$P(A|B) = \frac{P(A)P(B|A)}{P(A)P(B|A)+P(\text{NOT } A)P(B|\text{NOT } A)}$$



Figure 18.1: Gonic and Woollcott: A Classic (much like all other Gonick cartoon books)

certain disease. Suppose medical testing is not perfect (as is the case in real life) and consequently, only 99% of the people with the disease are tested positive. Suppose 2% of the people who don't have the disease also test positive. What is the probability of actually having the disease, given someone tests positive? **What if you were told that less than 5% of the people who test positive have the disease?**

- Suppose we have a room full of approximately 100 people (like this class, which has so many people on a crucial day). Suppose we go around the room asking people for their birthdays. Roughly how many people must we visit before we find two that have the same birthday? Is it like 10 people? 100? How does this relate to the design of hash-tables?

It is impossible to model and solve these problems unless we have the apparatus of probability theory. This is what we will do now.

## 18.1 Basic Definitions with Illustrations

In all our illustrations in this section, we will assume a pair of dice<sup>1</sup> and follow many of the illustrations in Gonick and Woollcott. We will add additional illustrations – and especially those pertaining to tree diagrams – borrowing the four-step method from Lehman et al.

### 18.1.1 One Die: Sample Space, Elementary Outcomes, Events, Probabilities

**Sample Space:** Let's assume that you have exactly one die with you, with the usual markings  $\{1, 2, 3, 4, 5, 6\}$ . Call this set your **sample space**  $S$ . Thus,  $S = \{1, 2, 3, 4, 5, 6\}$  as our first example.

**Elementary Outcomes:** Call  $w \in S$  an *elementary outcomes*. There are six outcomes for the above  $S$ .

**Outcome Probability:** Let the die be unbiased, meaning all rolls are equally likely (the way you understand this from experience). Thus, in 6 tosses, you'll very likely have each elementary outcome appear nearly 1/6th of the times. The more you toss, the closer this will get to 1/6th. This example illustrates the idea of *outcome probability* which we know through our experience with real-world (almost perfectly fair) die.<sup>2</sup>

In general, however, the sum of all outcome probabilities must be 1, and all outcome probabilities must be between 0 and 1. Thus, in a heavily biased die, it is possible for some outcomes to have a probability of 0. It is also possible that one of the outcomes has probability 1, in which case, all other outcomes have a probability of 0.

We will use the operator  $Pr()$  for denoting probabilities<sup>3</sup>

---

<sup>1</sup>Singular form is *die* as in die-cast metal; plural is *dice*, which rhymes with mice.

<sup>2</sup>We will ignore facts such as “face 1” being heavier than “face 6”—perhaps because six dots are gouged out to make the latter face.

<sup>3</sup>I may slip into  $P()$ , which is also widely used; if you please alert me, I can switch back to  $Pr()$ .

### 18.1.2 Events

An event  $E$  is a subset of  $S$ . In other words, an even is a one-ary *predicate* on  $S$ .

Events represent “more relevant outcomes”. Here are example events:

- A die toss revealing an even number:  $E_1 = \{2, 4, 6\}$
- A die toss revealing a prime number:  $E_2 = \{2, 3, 5\}$
- A die toss revealing a perfect square:  $E_3 = \{1, 4\}$
- A die toss revealing 6. This is a special case of an event, which degenerates to an elementary outcome. Specifically, here,  $E_4 = \{6\}$ .

**Exercises:**

1. ..

### 18.1.3 Event Probability

For an event  $E$ , the *event probability* is the sum of outcome probabilities for each outcome  $w \in E$ . The event probabilities in the above examples are  $Pr(E_1) = 1/2$ ,  $Pr(E_2) = 1/2$ ,  $Pr(E_3) = 1/3$ , and  $Pr(E_4) = 1/6$ .

### 18.1.4 Defining New Events Using Set Operations

Since events are subsets of the sample space, we can form new events from existing events using set operations. Here are examples of different *derived* events:

- $E_5$  = Probability that a single toss is even *and* prime.
- Naturally,  $E_5 = E_1 \cap E_2 = \{2\}$ , and so  $Pr(E_5) = 1/6$ .
- Let  $E_6$  = Probability that a single toss is even *or* prime.
- We have  $E_6 = E_1 \cup E_2 = \{2, 3, 4, 5, 6\}$ , and so  $Pr(E_6) = 5/6$ .<sup>4</sup>
- Let  $E_7$  = Probability that a single toss is not divisible by 3, and  $E_8$  = Probability that a single toss is divisible by 3. In other words,  $E_7 = \{1, 2, 4, 5\}$  and  $E_8 = \{3, 6\}$ .
- We can see that  $Pr(E_7) = Pr(\overline{E_8}) = 2/3$ , and  $Pr(E_8) = Pr(\overline{E_7}) = 1/3$ .
- The complementation is relative to the sample space  $S$  which is our universal set.
- Furthermore,  $Pr(E_7) = 1 - Pr(\overline{E_7})$ , and  $Pr(E_8) = 1 - Pr(\overline{E_8})$ .

Thus, given a sample space  $S$ , many different new events can be defined using  $\cap$ ,  $\cup$ , and  $\bar{\cdot}$  (often denoted by the *and*, *or*, and *not* of events).

**Disjoint (or Mutually Exclusive) Events:** Two events are disjoint (or mutually exclusive) if the sets they represent are disjoint, or mutually exclusive. In other words,  $E_1$  and  $E_2$  are disjoint if  $E_1 \cap E_2 = \emptyset$ .

### 18.1.5 Independent Events

Given a sample space  $S$ , two events  $E_1 \subseteq S$  and  $E_2 \subseteq S$  are, by definition, **independent** if

$$Pr(E_1 \cap E_2) = Pr(E_1) \cdot Pr(E_2)$$

Later, we will have another definition of independent in terms of conditional probabilities.

I would encourage you not to confuse disjoint from independent, even though they both sound alike. Here are two mnemonic aids:

- Think of the Gonick cartoon of two die coupled by a spring as the situation of being dependent! Performing event affects the other.
- For disjoint events, independence is an assumption (as pointed out in Lehman et al.).

Let us examine and see which of the events above are independent and which are dependent. Also be sure to go through the examples in Lehman et al.'s book, as it has many excellent discussions on independence.

- $E_1$  and  $E_2$  are **dependent** (not independent) because even though  $E_5 = E_1 \cap E_2 = \{2\}$ , we have  $Pr(E_5) = 1/6$  whereas  $Pr(E_1) \cdot Pr(E_2) = (1/2) \cdot (1/2) = 1/4$ .
- $E_7$  and  $E_8$  are disjoint. It is clear that they are dependent because while  $Pr(E_7 \cap E_8) = Pr(\emptyset) = 0$ , we have  $Pr(E_7) \cdot Pr(E_8) = 2/9$ . Therefore, disjoint events are *never* independent unless one of them has a probability of 0.

In case one of the events has probability 0, of course  $Pr(E_7 \cap E_8)$  and  $Pr(E_7) \cdot Pr(E_8)$  become the same, i.e., 0.

---

<sup>4</sup>Recall that 1 is not a prime—we had a long discussion about it.

**Exercise:** Consider all pairs of events  $E_1$  and  $E_2$  formed over  $S$ . It appears that all the non-empty as well as non-universal events derived out of the sample space of one die are dependent. In other words, consider the powerset of events obtained from  $S$ , and two elements  $E_1$  and  $E_2$  from this powerset such that:

- $E_1$  and  $E_2$  are not disjoint
- Neither  $E_1$  nor  $E_2$  are either 0 or 1

It now appears that  $E_1$  and  $E_2$  must be dependent. Argue why.

### 18.1.6 Illustrations on One Die Tossed Twice

We now consider one die that is tossed twice. How do we model the sample-space? The convention is to model the tosses as pairs, and so the sample space is modeled using sets of pairs, i.e.

$$\begin{aligned} S &= \{1, 2, 3, 4, 5, 6\} \times \{1, 2, 3, 4, 5, 6\} \\ &= \{(1, 1), (1, 2), \dots, (1, 6), (2, 1), \dots, (2, 6), \dots, (6, 1), \dots, (6, 6)\}. \end{aligned}$$

Let us now define several events:

- $E_a =$  Toss-1 yields a 1: This event is comprised of 6 outcomes, i.e.

$$E_a = \{(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6)\}$$

- $E_b =$  Toss-2 yields a 1: This event is comprised of 6 outcomes, i.e.

$$E_b = \{(1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1)\}$$

- $Pr(E_a) = 1/6$
- $Pr(E_b) = 1/6$
- $Pr(E_a \cap E_b) = Pr(\{(1, 1)\}) = 1/36$
- But notice that  $Pr(E_a \cap E_b) = Pr(E_a) \cdot Pr(E_b)$ .

This means that **the prob. of getting a “1” in Toss-1 is independent of the prob. of getting a “1” in Toss-2**. This is despite the fact that these sets are **not disjoint**. So, for the first time in our notes, we are able to see two events that are *not disjoint* and yet *independent*.

**Dependent and Non-Disjoint Events: Example-1** Let us find two events that are dependent as well as non-disjoint. Consider this slew of events now:

- $Sum_3$  = the two throws sum to 3.  $Pr(Sum_3) = 1/18$  because it is the set  $\{(1, 2), (2, 1)\}$  and  $|S| = 36$ .
- $Toss1_1$  = Toss-1 is a 1.  $Pr(Toss1_1) = 1/6$  because it is the set

$$\{(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6)\}$$

and  $|S| = 36$ .

- $Sum_3 \cap Toss1_1 = \{(1, 2)\}$  and so  $Pr(Sum_3 \cap Toss1_1) = 1/36$ .
- $Pr(Sum_3).Pr(Toss1_1) = 1/18 \cdot 1/6 = 1/108$ .
- Clearly,  $Pr(Sum_3 \cap Toss1_1) \neq Pr(Sum_3).Pr(Toss1_1)$ , and so these are *dependent* events.

Thus we have seen two event pairs, one dependent and one independent, in this example with one die and two tosses.

**The Or Rule: What is  $Pr(E_1 \cup E_2)$ ?** We spent a lot of time discussing the *And* (or  $\cap$ ) operation on two events.

- We learned that two events are *defined* to be independent if  $Pr(A \cap B) = Pr(A).Pr(B)$ .
- We learned that mutually exclusive (or disjoint) events are always dependent (unless the events are not trivial, having zero probability).
- The **Or** rule is

$$Pr(A \cup B) = Pr(A) + Pr(B) - Pr(A \cap B)$$

because the **Or** rule simply builds the union space of two events  $A$  and  $B$ , and they need not be disjoint events. Therefore, in order to avoid including  $A \cap B$  twice, we subtract that region (and correspondingly its probability).

- If  $A$  and  $B$  are disjoint, then

$$Pr(A \cup B) = Pr(A) + Pr(B)$$

because then  $Pr(A \cap B) = Pr(\emptyset) = 0$ .

**Illustrations on Two Dice Tossed Together:** We now consider two dice that are tossed together. How is this related to what was discussed with respect to one die thrown in succession? Nothing at all! We still model the sample space as a set of 36 pairs.

## 18.2 Conditional Probability

It is of importance to specify the probability of an event  $E_1$  conditioned upon event  $E_2$  occurring. For instance,  $E_1$  could be the event that “tomorrow is rainy” and  $E_2$  the event that “tomorrow is cloudy”. Clearly, these are dependent events, and in this case, the probability of  $E_1$  given  $E_2$  appears to be high, and this idea must be accurately understood. Now consider another situation where the efficacy of medical testing depends on whether a person has a disease or not. Even in this situation, a clear understanding of conditional probability is essential.

**Conditional Probability:** The conditional probability of  $E_1$  given  $E_2$  is denoted as  $Pr(E_1 | E_2)$ , and is defined as follows:

$$Pr(E_1 | E_2) = \frac{Pr(E_1 \cap E_2)}{Pr(E_2)}$$

This is defined assuming that  $Pr(E_2) \neq 0$ .

**Alternative Definition of Independence:** Events  $E_1$  and  $E_2$  are independent if (1) either  $Pr(E_2) = 0$ , or (2)  $Pr(E_1 | E_2) = Pr(E_1)$ . In other words, the occurrence of  $E_2$  does not alter the probability of  $E_1$ .

### 18.2.1 Bayes' Rule

Bayes' rule allows us to obtain the conditional probability  $Pr(E_1 | E_2)$  in terms of  $Pr(E_2 | E_1)$ . Here it is:

$$Pr(E_1 | E_2) = \frac{Pr(E_2 | E_1) \cdot Pr(E_1)}{Pr(E_2)}$$

**Proof of Bayes' Rule:** We have

$$Pr(E_1 | E_2)Pr(E_2) = Pr(E_1 \cap E_2) = Pr(E_2 | E_1)Pr(E_1)$$

By diving throughout by  $Pr(E_2)$  (assuming it is non-zero), we obtain the result.

**The Law of Total Probability:** The law of total probability allows us to compute  $Pr(E_1)$  piece-meal in terms of  $Pr(E_2)$  and  $Pr(\bar{E}_2)$ , as follows:

$$Pr(E_1) = Pr(E_1 | E_2).Pr(E_2) + Pr(E_1 | \bar{E}_2).Pr(\bar{E}_2)$$

This follows from the fact that

$$Pr(E_1) = Pr(E_1 \cap E_2) + Pr(E_1 \cap \bar{E}_2),$$

and we have

$$Pr(E_1 \cap E_2) = Pr(E_1 | E_2).Pr(E_2)$$

and

$$Pr(E_1 \cap \bar{E}_2) = Pr(E_1 | \bar{E}_2).Pr(\bar{E}_2).$$

### 18.3 Illustration: Medical Testing

Refer to the section on Bayes Theorem in the Gonick cartoons (around Page 25 of 27 in your handout). Suppose one in 1000 people have a certain disease. Suppose medical testing is not perfect (as is the case in real life) and consequently, only 99% of the people with the disease are tested positive. Suppose 2% of the people who don't have the disease also test positive. What is the probability of actually having the disease, given someone tests positive?

We solve it as follows:

- We are required to find  $Pr(Dis | Pos)$
- $Pr(Dis | Pos) = \frac{Pr(Dis \cap Pos)}{Pr(Pos)}$
- $Pr(Pos) = Pr(Pos | Dis).Pr(Dis) + Pr(Pos | \bar{Dis}).Pr(\bar{Dis})$
- We have  $Pr(Pos | Dis) = .99$ , and  $Pr(Dis) = 0.001$ , so the first product is 0.00099.
- We have  $Pr(Pos | \bar{Dis}) = .02$ , and  $Pr(\bar{Dis}) = 0.999$ , and so the second product is 0.01998.
- The denominator is  $0.00099 + 0.01998 = 0.02097$ .
- The answer is:  $0.00099/0.02097 = .04721$  or 4.271%.

### 18.4 Applications: Birthday “Paradox”

Let us compute the probability that  $N$  birthdays picked in a row are all distinct. Then, one minus this probability will indicate the probability that

out of  $N$  birthdays, two are the same. There are  $N$  choices for  $b_1$ ,  $N - 1$  choices for  $b_2$ , etc, and finally  $N - (m - 1)$  choices for  $b_m$ . Since the sample space is uniform, we have  $\Pr(\text{Distinct}) = \frac{N!}{N^m(N-m)!}$ . This approximates to

$$e^{(N-m+\frac{1}{2}).\ln(\frac{N}{N-m})-m}$$

### 18.4.1 Python-based Evaluation of Probabilities

We are now providing you some code using which you can plot a histogram of the number of collisions as a function of the range of values.

```
import random, sys

def nUnique(bound, r, cnt, S):
    """ Given a bound, an initial random number r in the range [1..bound],
    an accumulated count, and a set S, see if r is in S; if so, return cnt.
    Else generate another random number r, add 1 to count, and recurse.
    This function determines the number of unique values that could be
    generated in the range 1..bound. The idea is that we will call nUnique
    multiple times and plot a histogram.
    """
    if r in S:
        return cnt
    else:
        return nUnique(bound, random.randrange(1,bound+1), cnt+1, S | {r})

def plotUnique(Nbins, bound, ntrials):
    """Given the # of bins, divide up the data generated between min and max
    among these Nbins. Then put a "y-bar" that is as long as the # of data items
    in the bin. The bound variable is the range in which rand #'s are generated.
    The ntrials variable is the num. of trials we run.
    Call plotUnique(20, 20000, 200) several times, to get an idea.
    """
    min = 100000
    max = -1
    L = [] # a list used to record the values generated
    for i in range(ntrials):
        # Generate the # of times the rand call is repeated to collide with
        # a previously generated number.
        val = nUnique(bound, random.randrange(1,bound+1), 0, set({}))
        # Update max and min for each trial
        if (val > max):
            max = val
        if (val < min):
            min = val
        # val is used to decide the bin.
        L = [val] + L
    #--
    # Now, histogram the contents of L bounded by min and max
    # binWidth is the width of each bin
    # the last bin does exist in case there is truncation in this division
    #
    binWidth = (max - min)/Nbins + 1
```

```

#
print "binWidth, min, max = ", binWidth, min, max
# A hash-table called hist is to be filled now
hist = { i:0 for i in range(Nbins) }
# print hist
for n in L:
    binN = (n - min) / binWidth
    hist[binN] += 1
print "-----"
for i in range(Nbins):
    # print str(i) + ": " + ("." * hist[i])
    print '{0:4d}: '.format(min+i),
    print ("." * hist[i])
print "-----"

```

**Runs of plotUnique** The call `plotUnique(10,400,100)` shows that in a range of 400 executed over 100 trials, when we plot the histogram into bins of size 10, we observe these items:

- There are 7 collisions within the first bin. The first bin begins at 4 and goes till 14.
- This means that in the first bin of 10 points beginning at 4, there are 7 collisions.
- By the 12th bin that begins at 84, we have all the collisions.

```

-----
>>> plotUnique(10,400,100)
plotUnique(10,400,100)
binWidth, min, max =  6 4 57
-----
4: .....
5: .......
6: .......
7: .....
8: .......
9: .....
10: .....
11: ....
12: ....
13: ...
-----
```

**Applications: Hash-table Design:** Please read from Lehman et al the fact that in hash-table design, the value of  $N$ , the size of the hash-table, has to grow quadratically with the number of entries anticipated to be entered in order to avoid collisions.

## 18.5 Conditional Probabilities thru Gobblers

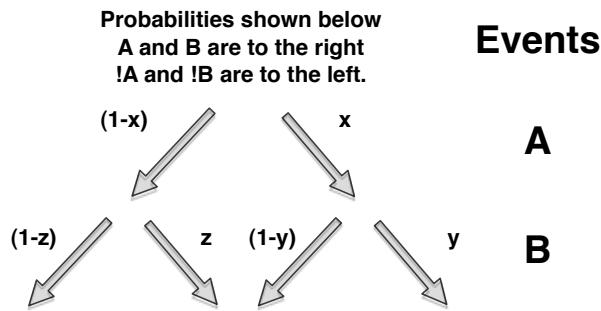


Figure 18.2: A Probability Tree: Event  $A$  could be “You Ate Turkey” and Event  $B$  could be “You Developed Indigestion”

Approaching the Thanksgiving, we would like to calculate the probability of developing indigestion, given the impending copious ingestion of our famous national bird – *i.e.* the gobbler!<sup>5</sup> Consider the probability tree of Figure 18.2. Here are some derivations pertaining to it:

$$1. \ Pr(A \cap B) = xy$$

$$2. \ Pr(A) = x$$

$$3. \ Pr(B | A) = y$$

$$4. \ Pr(A | B) = \frac{Pr(A \cap B)}{Pr(B)}$$

$$\begin{aligned} &= \frac{Pr(A \cap B)}{Pr(B | A).Pr(A) + Pr(B | \bar{A}).Pr(\bar{A})} \\ &= \frac{xy}{(xy + z.(1-x))} \end{aligned}$$

Since  $Pr(B).Pr(A | B) = Pr(A).Pr(B | A)$ , we have

$$5. \ Pr(B) = \frac{xy}{(xy + z(1-x))}$$

---

<sup>5</sup>Historic fact: if Ben Franklin had his way, the gobbler – and not the Bald Eagle – would have adorned our national seals. How fun it would have been!

$$= xy + z.(1 - x)$$

6. Now, **IF**  $z = y$ , then  $Pr(B) = y = z$ .
7. Then A and B would be independent also (*i.e.* the probability of your getting indigestion is independent of whether you consume the gobblor or not).



# Chapter 19

## Lecture Notes

These add to the chapters by way of lecture notes.

### 19.1 K-maps

An undirected graph consists of vertices (or nodes)  $V$  and edges  $E$ . The vertices (nodes)  $V$  are a set of names, and the edges  $E$  are a set of pairs of names. We will use “vertex” and “node” interchangeably, but denote it by  $V$ . Since the graph is undirected, we don’t care how we list the edges (in what order; thus  $(a,b)$  and  $(b,a)$  are equally valid listings of the same edges).

For example, consider the graph described by the state capitals of Utah (SLC), Nevada (CSN or Carson City), Arizona (Phoenix or PHX), and California (Sacramento, or SMF) going by the airport codes. Let every pair of capitals be connected. We obtain the graph shown in Figure 19.1. This graph, say “ $G_{SC4}$ ” standing for “the graph of four state capitals” is described

```
CSN----SLC
| \ / |
|   | |
| / \ |
SMF---PHX
```

Figure 19.1: An Undirected Graph of Four State Capitals of the Western US

as follows:

$$G_{SC4} = (V, E)$$

where

$$V = \{SLC, CSN, PHX, SMF\}$$

and

$$E = \{(SLC, CSN), (SLC, SMF), (SLC, PHX), (CSN, PHX), (CSN, SMF), (SMF, PHX)\}$$

There should be *six* edges, because we are choosing from *four* (4) capitals, taking *two* (2) at a time; or  $\binom{4}{2}$ , as we will soon study/recap. We can also present the graph as

$$\begin{aligned} G_{SC4} = \\ (\{SLC, CSN, PHX, SMF\}, \\ \{(SLC, CSN), (SLC, SMF), (SLC, PHX), (CSN, PHX), (CSN, SMF), (SMF, PHX)\}) \end{aligned}$$

Let us place a traveling salesperson in SLC and ask her to visit each state capital exactly once, returning to SLC (the return “touch” of SLC does not count as the second visit of SLC; the starting “touch” of SLC does count).

Clearly, the salesperson can visit in any one of these orders:

- SLC, PHX, SMF, CSN (and back to SLC, not explicitly listed).
- SLC, PHX, CSN, SMF (and back to SLC, not explicitly listed).
- SLC, CSN, SMF, PHX (and back to SLC, not explicitly listed).
- etc. etc. (many such – how many?)

Each of these is a Hamiltonian cycle.

**Hamiltonian cycle:** Given an undirected graph  $G$ , a *Hamiltonian cycle* is a *sequence* listing the nodes of  $G$  exactly once, starting with an arbitrary node  $n \in V$  of  $G$ , such that each adjacent pair of nodes in  $G$  is connected by an edge that is in  $E$ . For our example graph, different Hamiltonian cycles were listed above.

**Hamiltonian Cycle for Modified Graphs** In Figure 19.2(A), there are fewer Hamiltonian cycles. In Figure 19.2(B), there *no* Hamiltonian cycles (for what reason?). In Figure 19.2(C) also, there *no* Hamiltonian cycles (for a different reason; what is it?). In Figure 19.2(D), there are many Hamiltonian cycles (how many?), but a tour that goes as DEN, PHX, SLC, . . . cannot be extended to become a Hamiltonian cycle (why?).

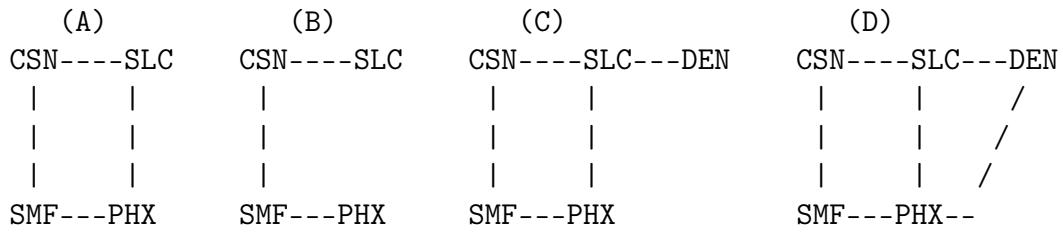


Figure 19.2: A Modified Graph of Four States

## 19.2 Chapter on BDDs

Lecture on BDD by showing what they are (as per Chapter 11).

<http://formal.cs.utah.edu:8080/pbl/BDD.php>, however we'll have to disable if load becomes intolerable (will have to host it somewhere else soon).

Walk thru the examples.

### 19.3 First Order Logic

Here are lecture plans for 10/22 (L15) and 10/24 (L16).

First order logic (FOL) is a logic in which assertions can be made about **universes** containing **individuals**.

Concretely, a **universe** is a collection of ordinary (day-to-day) values such as numbers, strings, etc.

Here is how FOL is set up:

1. Choose a universe of individuals. Universes are **non-empty**. They may be finite or infinite.
  - (a) Example:  $\{0, 1, 2, 3\}$
  - (b) Example:  $Nat = \{0, 1, 2, 3, \dots\}$
  - (c) Example:  $Animals = \{fido, howard, felix\}$
2. Choose predicates of interest. Predicates are used to convey the truth you would like to assert of individuals.
  - (a) Example:  $cat, dog, odd, even, prime$
  - (b) Predicates denote **relations**, a topic that we are going to study in Chapter 14 (except in Chapter 14, we are concerned with **binary** predicates (or **binary** relations)).
  - (c) Example of usage:  $cat(felix), dog(howard), odd(1), even(2)$ , etc.
3. Predicates have *arities*
  - (a) A one-ary predicate takes one argument: Example:  $odd(arg1), dog(arg1)$
  - (b) A two-ary predicate takes two arguments: Example:  $>, gt, lt$ , etc.
  - (c) Usage of two-ary predicates:  $x > y, >(x, y), gt(x, y), older(x, y)$ , etc.
  - (d) Three-ary predicates:  $ascending3(3, 4, 5)$  is true;  $ascending3(3, 4, 2)$  is false.
  - (e) More three-ary predicates: Pythagorean triples  $pyth3(3, 4, 5), pyth3(4, 3, 5), pyth3(5, 12, 13)$ , etc can be defined to be true as follows.  $pyth3(x, y, z)$  is true exactly when  $x^2 + y^2 = z^2$ .

4. Let us denote the universe by  $U$ . Then, predicates of arity  $N$  are subsets of  $U^N$ . For instance,
- (a) The two-ary predicate  $>$  is
- $$\{\langle 1, 0 \rangle, \langle 2, 0 \rangle, \langle 2, 1 \rangle, \langle 4, 2 \rangle, \dots\}$$
- (b) This is clearly a subset of  $\text{Nat} \times \text{Nat}$ .
  - (c)  $\text{dog} \subset \text{animals}$  because (let us say)  $\text{dog} = \{fido, howard\}$ .
  - (d)  $\text{cat} \subset \text{animals}$  because (let us say)  $\text{cat} = \{felix\}$ .
  - (e) Predicates can be empty. Suppose Lochness monsters don't exist. Then  $\text{lochness}(x)$  is always false. Thus,  $\text{lochness} \subset \text{animals}$  where  $\text{lochness} = \emptyset$ .
  - (f) Suppose we define  $\text{natbetween01}(x)$  to be the natural number between 0 and 1. Since this does not exist,  $\text{natbetween01} = \emptyset$ .
  - (g) Suppose we have  $\text{isanat}$  to be a predicate defined over our universe  $U$  being  $\text{Nat}$ . Since  $\text{isanat}(x)$  is true of every  $x$ , we have  $\text{isanat} = \text{Nat}$ .
5. Assigning truth values: It is clear that  $\text{odd}(1)$  is true, and  $\text{odd}(2)$  is false. How do we make a general statement about odd numbers? We can say  $\text{odd}(x)$  and leave  $x$  unspecified.
6. The way FOL has devised is to give you **two** ways to make general statements:
- (a) “For all” – or “For every”
  - (b) “There exists” – or “For some”
7.  $\forall$  means “for all”
8.  $\exists$  means “there exists”

9. When we say  $\forall x : p(x)$  for some predicate, the meaning is very similar to  $\Pi$  or  $\Sigma$  in mathematics (these stand for repeated multiplication and repeated addition). Likewise

- (a)  $\forall$  is a tool for performing repeated  $\wedge$  (conjunction)
- (b)  $\exists$  is a tool for performing repeated  $\vee$  (disjunction)

10.  $\forall x : \text{dog}(x)$  means “for every value of  $x$  over  $U$ ,  $\text{dog}(x)$ ”; i.e.,

$$\begin{aligned} &\wedge \text{dog}(\text{fido}) \\ &\wedge \text{dog}(\text{howard}) \\ &\wedge \text{dog}(\text{felix}) \end{aligned}$$

11. We know that this evaluates to false. That is OK.

12.  $\exists x : \text{dog}(x)$  means “there exists an  $x$  over  $U$ , for which  $\text{dog}(x)$ ”; i.e.,

$$\begin{aligned} &\vee \text{dog}(\text{fido}) \\ &\vee \text{dog}(\text{howard}) \\ &\vee \text{dog}(\text{felix}) \end{aligned}$$

13. We know that this evaluates to true.

14. We now can see that  $\neg(\forall x : \text{dog}(x)) = \exists x : \neg \text{dog}(x)$  – try to apply the negation to the expansion of  $\forall$  and simplify using DeMorgan’s law, and then rewrite the result in the  $\exists$  form.

15.  $\neg(\forall x : \text{dog}(x)) =$

$$\begin{aligned} &\neg( \\ &\quad \wedge \text{dog}(\text{fido}) \\ &\quad \wedge \text{dog}(\text{howard}) \\ &\quad \wedge \text{dog}(\text{felix}) \\ &\quad ) \end{aligned}$$

=

$$\begin{aligned} & \vee \neg \text{dog}(fido) \\ & \vee \neg \text{dog}(howard) \\ & \vee \neg \text{dog}(felix) \end{aligned}$$

16. which reduces to  $\exists x : \neg \text{dog}(x)$ .
17. Often we like to quantify over subsets of  $U$ . This usage is also important to understand.
18. Let  $D = \{fido, howard\}$  and  $C = \{felix\}$ .
19. Let  $Odd = \{1, 3, 5, \dots\}$ ,  $Prime = \{2, 3, 5, 7, \dots\}$ , and  $Even = \{0, 2, 4, \dots\}$ .
20. Now we can define  $\forall x \in D : \text{dog}(x)$  which means  $\forall x : x \in D \Rightarrow \text{dog}(x)$ . Here, the  $\forall x :$  still ranges over  $U$ , the universe. But because of the implication's antecedent  $x \in D$ , we are “guarded” to effectively evaluate  $\text{dog}(x)$  only over dogs.
21. Now we can define  $\exists x \in D : \text{dog}(x)$  which means  $\exists x : x \in D \wedge \text{dog}(x)$ .
22. Now consider  $\forall x \in Odd : even(x + 1)$
23. This means  $\forall x : x \in Odd \Rightarrow even(x + 1)$
24. This is true.
25. How about  $\neg(\forall x \in Odd : even(x + 1))$ ?

$$\begin{aligned} & \neg(\forall x : (x \in Odd \Rightarrow even(x + 1))) \\ &= (\exists x : \neg(x \in Odd \Rightarrow even(x + 1))) \\ &= (\exists x : (x \in Odd \wedge \neg even(x + 1))) \\ &= (\exists x \in Odd : \neg even(x + 1)) \end{aligned}$$

26. Thus, the explicit domain specified for quantification does not seem to bother the negation process (it is true; it works the same as the case where we applied DeMorgan's law to the "plain case" of quantification).

### 19.3.1 Exercises

Test your knowledge and understanding by answering these problems:

1.  $U = \text{Nat}$
2. *Predicates:  $<$ ,  $>$ ,  $\leq$ ,  $\geq$ , odd, even, prime*
3. Special domains of interest: *Even, Odd, Prime*
4. Answer whether true or false. Discuss the details.
5.  $\forall x : \exists y : y > x$
6.  $\forall x : \exists y : y < x$
7.  $\forall x \in \text{Nat} : \exists y \in \text{Nat} : y > x$
8.  $\forall x \in \text{Nat} : \exists y \in \text{Nat} : y < x$
9.  $\forall x \in \text{Prime} : \text{odd}(x)$
10.  $\forall x \in \text{Prime} : \exists y \in \text{Prime} : (y > x \wedge \text{prime}(y))$
11. Related to Fermat's last theorem
12. Are these equivalent? If so, prove:

13.  $\forall a, b, c \in Nat : \neg(\exists N \in Nat : (N \geq 3 \wedge a^N + b^N = c^N))$
14.  $\forall a, b, c \in Nat : (\exists N \in Nat : a^N + b^N = c^N \Rightarrow N \leq 2)$
15.  $\forall a, b, c \in Nat : (\forall N \in Nat : (N < 3 \vee a^N + b^N \neq c^N))$
  
16. Notice that  $\forall x \in \emptyset : p(x)$  is true for any  $p$ . There are two ways to think about this:
  - (a) The antecedent of the implication in the expansion of  $\forall x \in \emptyset$  would be  $x \in \emptyset$  which is false, making the implication true.
  - (b) When you perform **repeated**  $\wedge$ , you must start with the basis case of *True*.
  
17. Notice that  $\exists x \in \emptyset : p(x)$  is false for any  $p$ . There are two ways to think about this:
  - (a) The first conjunct in the expansion of  $\exists x \in \emptyset$  would be  $x \in \emptyset$  which is false, making the conjunction false.
  - (b) When you perform **repeated**  $\vee$ , you must start with the basis case of *False*.
  
18. Look at the problems at the end of Chapter 13.
19. Let us learn Prolog from Chapter 13; specifically, Section 13.5.

## 19.4 Relations: Chapter 14

We can express the ideas in this chapter using first-order logic. Let  $S$  be the set over which the binary relation  $R$  is being expressed. We are making a second order assertion by saying  $\text{reflexive}(R)$  because we are defining predicates over relations. But for this minor distraction, we can happily use the style of “first order logic” assertions as we have been using.

**NOTE:** Because of the very slippery nature of the topics here, we will really expect you to know **preorder** and **partial order**. The rest are there for detail, and to serve as first-order logic exercises.

1. Reflexive:  $\text{reflexive}(R) \Rightarrow \forall x \in S : R(x, x)$
2. “No dot without a loop”
3. Irreflexive:  $\text{irreflexive}(R) \Rightarrow \forall x \in S : \neg R(x, x)$
4. “No dot has a loop”
5. Non-reflexive:  $\text{nonreflexive}(R) \Rightarrow \exists x \in S : \neg R(x, x)$ . This is the negation of reflexive.
6. “Exists one dot with a loop and one dot without a loop”
  
7. Symmetric:  $\text{symmetric}(R) \Rightarrow \forall x, y \in S : R(x, y) \Rightarrow R(y, x)$ . Food for thought: why did we not need to use  $\Leftrightarrow$ ? Note that  $x, y$  need not be distinct.
8. “No single arrows. If it holds one way, it holds the other way also.”
  
9. Asymmetric:  $\text{asymmetric}(R) \Rightarrow \forall x, y \in S : R(x, y) \Rightarrow \neg R(y, x)$ . **It is important to note** that  $x, y$  need not be distinct.
10. “No double arrows. If the relation holds one way, it does not hold the other way.”
  
11. A relation can **neither** be symmetric nor be asymmetric: Example is  $\leq$ .
12. A relation can **both** be symmetric and asymmetric: Example is the empty relation.
  
13. Non-symmetric:  $\text{nonsymmetric}(R) \Rightarrow (\neg \text{symmetric}(R) \wedge \neg \text{asymmetric}(R))$ .
14. “At least one single arrow and at least one double arrow.”

15. Antisymmetry:  $\text{antisymmetric}(R) \Rightarrow \forall x, y \in S : (\langle x, y \rangle \in R \wedge \langle y, x \rangle \in R \Rightarrow x = y)$
16. “No double arrow unless it is a loop.”
17. Transitive:  $\text{transitive}(R) \Rightarrow \forall x, y, z \in S : R(x, y) \wedge R(y, z) \Rightarrow R(x, z).$
18. “No broken journey without a shortcut.”
19. Intransitive:  $\text{intransitive}(R) \Rightarrow \forall x, y, z \in S : R(x, y) \wedge R(y, z) \Rightarrow \neg R(x, z).$
20. “No broken journey *with* a shortcut.”
21. Non-transitive: One broken journey with a shortcut and one broken journey without a shortcut.
22.  $\text{Preorder}(R) \Rightarrow \text{reflexive}(R) \wedge \text{transitive}(R)$
23.  $\text{Partialorder}(R) \Rightarrow \text{reflexive}(R) \wedge \text{antisymmetric}(R) \wedge \text{transitive}(R)$
24. In other words,  $\text{Partialorder}(R) \Rightarrow \text{Preorder}(R) \wedge \text{antisymmetric}(R).$

## 19.5 How to think about induction?

I've known about and employed induction for decades now. I've often listened to the advice "induct early, induct often" and done exactly that. Yet, it bothers me that there are all these induction styles being discussed, without a strict separation of concerns (and to name a few, here they are):

- Arithmetic
- Complete
- Structural
- Subgoal
- Noetherian

Here is another attempt to address the heart of the matter in a methodical fashion. Please read along, making sure that you follow each subsection. Ask questions if you have any.

### 19.5.1 Induction: An Approach to Proving General Facts

Induction is an approach for proving general facts. More specifically, it is an approach to prove

$$\forall x : \text{good}(x)$$

One can proceed with induction only after these two requirements are met:

- One sees the "forall" question being asked. *Unfortunately, sometimes, the forall is not apparent. Sometimes, one has to tease it out.*
- One has to know what the `good()` predicate is.

**Explicit Statement of Forall:** Sometimes, the general proof being requested is crystal-clear from the problem statement. Here are a few examples:

- Prove for all  $n \in \text{Nat}$  that either  $n$  is even or  $n + 1$  is even.
- Prove for all  $n \in \text{Nat}$  that  $\sum_{i=1}^n i = n.(n + 1)/2$ .
- Prove for all  $n \in \text{Nat}$  that  $n$  is uniquely expressible as the product of primes.
- Prove for all  $n \in \text{Nat}$  that there are  $2^n$  elements in any powerset of  $n$  elements.

In all the examples below, the question itself asks you "prove for all  $n \in \text{Nat}$  that *blah*."

**Implicit Statement of Forall:** At other times, the general proof being requested isn't so clear, and we must tease it out. Here are examples:

- Prove that  $\sqrt{2}$  is irrational.
- Prove that the largest postage value that *cannot* be realized using a 3-cent stamp and a 7-cent stamp is 11 cents.

**Question:** In the above two questions, where is the “forall” ??

**Answer:** We have to massage the question to yield the “forall.”

**Explicating the Forall:** Here is how we might re-state the questions to make the “forall” visible:

- Prove that for all  $a, b \in \text{Nat}$ , it is the case that  $\sqrt{2} \neq a/b$ . *Ah ha, now you are talking! I do see the forall!*
- Prove that for all  $n \geq 11$ , we can realize any postage value of  $n$  cents using only 3-cent and 7-cent stamps. *Ah ha, now you are talking! I do see the forall even here!*

**Identifying the *good()* predicate:** The next task is to state the *good()* predicate. For our examples, it looks so (respectively):

- *good(n)* is: “ $n$  is even or  $n + 1$  is even”.
- *good(n)* is:  $\sum_{i=1}^n i = n.(n+1)/2$ .
- *good(n)* is: “ $n = p_1^{e_1} \times p_2^{e_2} \times \dots \times p_k^{e_k}$ ” for primes  $p_1 \dots p_k$  and exponents  $e_1 \dots e_k$ .
- *good(n)* is “ $2^n$  elements in the powerset of an  $n$ -element set.”
- *good(n)* is “ $\forall a, b \in \text{Nat} : a/b \neq \sqrt{2}$ .”
- For the stamps problem, it is: “ $n \geq 11 \Rightarrow \exists i, j \in \text{Nat} : 7i + 3j = n$ .”

### 19.5.2 Taking Stock of $\forall x : \text{good}(x)$

Now that we have seen enough examples of how the problem to be solved can be stated as  $\forall x : \text{good}(x)$ , we notice these facts:

- The  $x$  in  $\forall x$  is often a member of  $\text{Nat}$
- But in one example, we have something different:
  - The proof goal is  $\forall a, b \in \text{Nat} : a/b \neq \sqrt{2}$
  - **Thus, “x” are pairs  $\langle a, b \rangle$ .**
- **Thus, in general**,  $x$  in  $\forall x : \text{good}(x)$  template ranges over the underlying set of values which could be:
  - A set of numbers (subset of  $\text{Nat}$ )

- A set of *pairs of numbers*
- Many other variations, such as a set of pairs  $\langle a, b \rangle$  where  $a$  is a number and  $b$  is a string.

### 19.5.3 Inductive Proof of $\forall x : good(x)$

The inductive proof of  $\forall x : good(x)$  proceeds as follows. There are two styles:

- The style of arithmetic induction.
- The style of complete induction.

To understand these styles, think of the domino analogy:<sup>1</sup>

- Induction “works” by letting some set of earlier dominoes trip the current domino.
- The basis-case domino is shown to be tripped.

**Arithmetic Induction:** In arithmetic induction, we have the following plan (following the domino analogy):

- Assuming that the  $(n - 1)$ st domino is tripped, we must show that the  $n$ th domino is tripped.
- We must show that the 0th domino (basis case domino) is tripped.

Following this recipe, we attempt to write the following style of proof:

- We prove the basis case. *i.e.*  $good(0)$  or  $good(a_0, b_0)$  for the smallest arguments of  $good()$ , depending on what the underlying set for  $x$  in  $good(x)$  is.
- We assume that  $good(n - 1)$  is true (or  $good(a_{n-1}, b_{n-1})$ , if the underlying set consists of pairs).
- We show  $good(n)$  (or  $good(a_n, b_n)$ , if the underlying set consists of pairs).

**Complete Induction:** In complete induction, we have the following plan (following the domino analogy):

*Assuming that all dominoes prior to the  $n$ th domino are tripped,  
we must show that the  $n$ th domino is tripped.*

This often causes one to wonder: *what happened to the basis case?* The answer is that in actual usage, we will be forced to do a case split:

---

<sup>1</sup>Not the edible kind—but as in the game of dominoes.

- The current ( $n$ th) element is not the smallest element in the order: then we can assume that all elements below  $n$  (all those for  $n - i$ , for  $0 < i \leq n$ ) satisfy *good*.
- $good(0)$  is explicitly shown.

### 19.5.4 Induction Principles in Second Order Logic

It turns out that induction is a method of proof applicable to *any* predicate at all. Thus, it is not a method for a particular *good* predicate—but all such. In capturing this, we can write the arithmetic induction principle as follows:

**Arithmetic Induction Principle (Theorem):** Induction principles are theorems in the underlying set theory that one assumes “while doing math.” The Arithmetic Induction principle is as follows:

$$\begin{aligned} \forall P : & \wedge P(0) \\ & \wedge \forall x > 0 : P(x - 1) \Rightarrow P(x) \\ \Rightarrow & (\forall x : P(x)). \end{aligned}$$

Likewise, the Complete Induction principle is as follows:

$$\begin{aligned} \forall P : \forall y : & (\forall x < y : P(x)) \Rightarrow P(y) \\ \Rightarrow & (\forall x : P(x)). \end{aligned}$$

It is possible to prove that these two statements are entirely equivalent. We will not delve into this proof here (see my book *Computation Engineering: Applied Automata Theory and Logic*, Spring 2006 for details.)

### 19.5.5 Connection Between Induction and Recursion

Induction and recursion are intimately related. While induction is a proof principle, recursion is a definitional device to define functions. Here are some facts surrounding recursion and induction:

- When functions are defined via recursion, we might want to prove something “good” about the definition. Such a proof (by induction) can exploit the structure of the recursion. An example is provided in §19.5.7.

- When functions are defined via recursion, we definitely want to show that the function terminates for all arguments. While this is often possible in practice, in general this faces difficulties. This is highlighted by one example in §19.5.9.
- In general, termination is impossible to mechanically prove. This famous “Halting problem” will be studied in CS 3100.

Whenever we define something

### 19.5.6 Example: Proving that $\sqrt{2}$ is Irrational

(Adapted from Charlie Jacobsen’s notes).

**Part 1:** Our goal is to prove  $\sqrt{2}$  is irrational. Let’s re-phrase that into:

For all  $x$  and  $y \geq 0$ ,  $x / y \neq \sqrt{2}$

So, we need to consider all pairs  $(x, y)$  and show that  $x/y \neq \sqrt{2}$ . We’ll return to this, but set it aside for a moment.

We are going to discuss how to compare things. We have a standard way of comparing two numbers (e.g.,  $2 < 3$ ,  $10 \geq -1.5$ ), but how do we compare ordered pairs?

Draw yourself an x-y plane, labeling points  $(0,0)$ ,  $(1,0)$ ,  $(2,0)$ , ...,  $(0,1)$ ,  $(0,2)$ , ...,  $(1,1)$ ,  $(1,2)$ ,  $(2,1)$ ,  $(2,2)$ , etc.

How do we say one ordered pair is smaller than another? For example, is  $(2,3) < (10,15)$ ? We really can’t answer that until we -define-  $<$  for ordered pairs. This is what the list does:

$(0,0), (0,1), (1,0), (0,2), (1,1), (2,0), \dots$

This means  $(0,0) < (0,1)$ ,  $(0,2) < (2,0)$ , and  $(0,0) < (2,0)$ , for example. Now, I’ll re-write the list, but grouping the pairs in a certain way:

$(0,0), (0,1), (1,0), (0,2), (1,1), (2,0), (0,3), (1,2), (2,1), (3,0), \dots$

**Part 2:** If you connect the points in each group, you get diagonal lines. What happens when you add up the x- and y-coordinates of points on the same line?

- Okay, now for a test: Using the way we have defined  $<$  for ordered pairs,

- is  $(0,1) < (3,0)$ ? (ans: yes)
- is  $(1,2) < (0,4)$ ? (ans: yes)
- which one is 'bigger',  $(10,15)$  or  $(11,5)$ ? (ans:  $(10,15)$  is bigger)
- which one is 'bigger',  $(100, 125)$  or  $(75, 150)$ ? (ans:  $(100, 125)$  is bigger)
- What two rules can you write down for determining when  $(x,y) < (u,v)$ ?

**Part 3:** Now, return to the goal, stated in Part 1. Here are some hints:

1. Eliminate the edge cases when  $x$  or  $y$  is 0 -before- you start the proof by induction.
2. Take  $P(x,y)$  to be  $x^2 \neq 2y^2$  (this is equivalent to  $x/y \neq \sqrt{2}$  once we remove the edge cases).
3. Base case is  $(1,1)$  (now that we have eliminated the edge cases).
4. The induction hypothesis is "for all  $(u,v) < (x,y)$ ,  $P(u,v)$  is true". See the  $<?$  \*\*That's where the ordering comes in.\*\* It will also enter into our proof in the inductive step.
5. Your goal in the inductive step is to prove  $P(x,y)$  is true. Here, you can break it down into cases as was done in Erickson's notes (so you can follow much of the same structure).
6. The even-even case is when you will need to use the induction hypothesis.

### 19.5.7 Proving a Recursive $\text{gcd}(x,y)$ Correct

Consider the program  $\text{gcd}(a,b)$  where  $a,b$  are assumed to be  $\geq 1$ :

```
# Examples of usage:
#
# gcd(22,545)
# 1
#
# >>> gcd(2331, 696)
# 3
#
def gcd(a,b):
    if (a==b):
        return a
    elif (a < b):
        return gcd(a, b-a)
    else:
        return gcd(b, a-b)
```

Here is one way to write a proof of correctness:

- We have to think of the program not going into an infinite loop, *i.e.*, it must *halt*.

- We have to ensure that the result is “correct.”

Both are related ideas. We will prove correctness by induction, and in doing that, we will assume that no matter how the program is invoked (with  $a, b \geq 1$ ), the basis case of  $a = b$  will be hit. Here are the details:

- We observe that the *sum* of the gcd arguments decreases during each iteration; and so, it can’t go into an infinite loop. It has to eventually hit  $(a==b)$ . Reason: each time, we shrink the arguments as well as leave both arguments above 0. Thus, we cannot go below  $a = b = 1$ . It may settle at some higher  $a = b$ , for instance if we call it with  $a = 4, b = 8$ , we will settle at 4,4.
- The  $good(a,b)$  assertion is this:  $gcd(a,b) = c$  where  $c$  divides  $a$  and  $c$  divides  $b$  and is greater than or equal to any alternate divisor. Thus, we have  $good(a,b,c)$  as the assertion, where  $c$  is calculated as  $gcd(a,b)$ .
- Write  $(c | a)$  to mean  $c$  divides  $a$ , and similarly for  $(c | b)$ .
- In other words,  $good(a,b,gcd(a,b))$  is assumed. That is,  $gcd(a,b)$  is indeed a function call that returns the required  $c$  which acts as the greatest common divisor.
- Define “below” or  $<$  to be a partial order:  $(a_1, b_1) < (a_2, b_2)$  holds if  $(a_1 + b_1) < (a_2 + b_2)$ .
- Proof is by induction: Assume that all  $(a,b)$  pairs below  $gcd(a,b)$  satisfy *good*.
- As induction step, we have to show  $gcd(a,b) = c$  also satisfies *good*. More specifically, we have to show these:
  - $good(a,b-a,c) \Rightarrow good(a,b,c)$ .
    - \* Since  $(c | a)$  and  $(c | b-a)$ , it is obvious that  $(c | b)$ .
    - \* We know that  $c$  is the highest divisor such that  $(c | a)$  and  $(c | b-a)$ .
    - \* But suppose there is *another* divisor  $d$  that divides  $a$  and  $b$ , and  $d > c$ .
    - \* Then  $d$  would also divide  $b-a$  and exceed  $c$ —assumed to be the GCD by induction hypothesis. That would be a contradiction. Therefore such a  $d$  must be less than or equal to  $c$ .
    - \* Hence  $good(a,b,c)$  indeed holds.
  - $good(b,a-b,c) \Rightarrow good(a,b,c)$ . Write a similar proof.

### 19.5.8 McCarthy's “91 function”

This section is to illustrate the degree of creativity that might be needed  
[http://en.wikipedia.org/wiki/McCarthy\\_91\\_function](http://en.wikipedia.org/wiki/McCarthy_91_function)

Consider this famous function, called “McCarthy's 91 function” ( $M$  for McCarthy):

```
M(n) = if n > 100 then n - 10 else M(M(n+11))
```

We are asked to show that this function returns “91” for all  $n \neq 100$ . (It is obvious that it returns  $n - 10$  for higher  $n$ , so we don't prove anything there.) Finding the induction argument is tricky; but, using the arguments (refer to the above Wikipedia article), we do it as follows:

- We observe that for  $n \in [90, 100]$ , the function returns 91. (Test this entire block of 11 numbers.) Take *this* to be the basis case!
- Now induct *downwards*. That is, go down from  $[90, 100]$  through blocks  $[79, 89], [68, 78]$ , etc. toward  $-\infty$ .
- Assume that for a block of 11, i.e.,  $[a, a + 10]$ , the property of “91” is true. Here,  $a \leq 80$ .
- Now we have to show that it holds for “ $(n + 1)$ ” which is, for the block  $[a - 11, a - 1]$ , the property of “91” is true.
- When we call the  $M$  function for  $n \in [a - 11, a - 1]$ , the recursion immediately calculates  $M(M(n + 11))$ . The inner  $M(n + 11)$  falls into the induction hypothesis, returning “91”.
- The whole call then returns 91, as  $M(91) = 91$ .

### 19.5.9 Collatz's Conjecture or “3n+1” Problem

This problem is also known as “3 n plus 1”. Described in many places, including

[http://en.wikipedia.org/wiki/Collatz\\_conjecture](http://en.wikipedia.org/wiki/Collatz_conjecture)

**it is still not known whether this function terminates for all  $n$ .** Of course, whenever it terminates, it returns a 1. Let's define the function (with tracing):

```
#-----
# The "3n+1" function is defined below
#
# TNP1(44)
# 1
#
# >>> TNP1(449)
# 1
#
# >>> TNP1(4492929)
```

```

# 1
#
import sys
from functools import wraps

class TraceCalls(object):
    """ Use as a decorator on functions that should be traced. Several
        functions can be decorated - they will all be indented according
        to their call depth.
    """
    def __init__(self, stream=sys.stdout, indent_step=2, show_ret=False):
        self.stream = stream
        self.indent_step = indent_step
        self.show_ret = show_ret

        # This is a class attribute since we want to share the indentation
        # level between different traced functions, in case they call
        # each other.
        TraceCalls.cur_indent = 0

    def __call__(self, fn):
        @wraps(fn)
        def wrapper(*args, **kwargs):
            indent = ' ' * TraceCalls.cur_indent
            argstr = ', '.join(
                [repr(a) for a in args] +
                ["%s=%s" % (a, repr(b)) for a, b in kwargs.items()])
            self.stream.write('%s%s(%s)\n' % (indent, fn.__name__, argstr))

            TraceCalls.cur_indent += self.indent_step
            ret = fn(*args, **kwargs)
            TraceCalls.cur_indent -= self.indent_step

            if self.show_ret:
                self.stream.write('%s--> %s\n' % (indent, ret))
            return ret
        return wrapper

    #-----
    @TraceCalls()
    def TNP1(n):
        if (n==1):
            return 1
        elif (n%2 == 0):
            return TNP1(n/2)
        else:
            return TNP1(3 * n + 1)

    #-----

```

Let's play with this function:

```

Python 2.7.2 (v2.7.2:8527427914a2, Jun 11 2011, 15:22:34)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

```

```

>>> execfile("TNP1.py")
>>> TNP1(49)
TNP1(4)
TNP1(4)
    TNP1(2)
        TNP1(1)
1

>>> TNP1(49)
TNP1(49)
    TNP1(148)
        TNP1(74)
            TNP1(37)
                TNP1(112)
TNP1(56)
    TNP1(28)
        TNP1(14)
            TNP1(7)
                TNP1(22)
TNP1(11)
    TNP1(34)
        TNP1(17)
            TNP1(52)
                TNP1(26)
                    TNP1(13)
                        TNP1(40)
                            TNP1(20)
                                TNP1(10)
                                    TNP1(5)
                                        TNP1(16)
                                            TNP1(8)
                                                TNP1(4)
                                                    TNP1(2)
                                                        TNP1(1)

1

>>> TNP1(493)
TNP1(493)
TNP1(493)
    TNP1(1480)
        TNP1(740)
            TNP1(370)
                TNP1(185)
TNP1(556)
    TNP1(278)
        TNP1(139)
            TNP1(418)
                TNP1(209)
TNP1(628)
    TNP1(314)
        TNP1(157)
            TNP1(472)
                TNP1(236)
                    TNP1(118)
                        TNP1(59)
                            TNP1(178)
                                TNP1(89)
                                    TNP1(268)
                                        TNP1(134)

TNP1(67)
    TNP1(202)
        TNP1(101)
            TNP1(304)
                TNP1(152)
                    TNP1(76)
                        TNP1(38)
                            TNP1(19)
                                TNP1(58)
                                    TNP1(29)
                                        TNP1(88)

TNP1(44)
    TNP1(22)
        TNP1(11)
            TNP1(34)
                TNP1(17)
                    TNP1(52)

```

```

TNP1(26)
TNP1(13)
TNP1(40)
TNP1(20)
TNP1(10)
TNP1(5)
TNP1(16)
TNP1(8)
TNP1(4)
TNP1(2)
TNP1(1)

1

```

```

>>> TNP1(371)
TNP1(371)
TNP1(371)
TNP1(1114)
TNP1(557)
TNP1(1672)
TNP1(836)
TNP1(418)
TNP1(209)
TNP1(628)
TNP1(314)
TNP1(157)
TNP1(472)
TNP1(236)
TNP1(118)
TNP1(59)
TNP1(178)
TNP1(89)
TNP1(268)
TNP1(134)
TNP1(67)
TNP1(202)
TNP1(101)
TNP1(304)
TNP1(152)
TNP1(76)
TNP1(38)
TNP1(19)
TNP1(58)
TNP1(29)
TNP1(88)
TNP1(44)

TNP1(22)
TNP1(11)
TNP1(34)
TNP1(17)
TNP1(52)
TNP1(26)
TNP1(13)
TNP1(40)
TNP1(20)
TNP1(10)

TNP1(5)
TNP1(16)
TNP1(8)
TNP1(4)
TNP1(2)
TNP1(1)

1

```

```

>>> TNP1(313)
TNP1(313)
TNP1(313)
TNP1(940)
TNP1(470)
TNP1(235)
TNP1(706)
TNP1(353)
TNP1(1060)
TNP1(530)
TNP1(265)
TNP1(796)

```

```

TNP1(398)
  TNP1(199)
    TNP1(598)
      TNP1(299)
        TNP1(898)
TNP1(449)
  TNP1(1348)
    TNP1(674)
      TNP1(337)
        TNP1(1012)
          TNP1(506)
            TNP1(253)
              TNP1(760)
                TNP1(380)
                  TNP1(190)
TNP1(95)
  TNP1(286)
    TNP1(143)
      TNP1(430)
        TNP1(215)
          TNP1(646)
            TNP1(323)
              TNP1(970)
                TNP1(485)
                  TNP1(1456)
TNP1(728)
  TNP1(364)
    TNP1(182)
      TNP1(91)
        TNP1(274)
          TNP1(137)
            TNP1(412)
              TNP1(206)
                TNP1(103)
                  TNP1(310)
TNP1(155)
  TNP1(466)
    TNP1(233)
      TNP1(700)
        TNP1(350)
          TNP1(175)
            TNP1(526)
              TNP1(263)
                TNP1(790)
                  TNP1(395)
TNP1(1186)
  TNP1(593)
    TNP1(1780)
      TNP1(890)
        TNP1(445)
          TNP1(1336)
            TNP1(668)
              TNP1(334)
                TNP1(167)
                  TNP1(502)
TNP1(251)
  TNP1(754)
    TNP1(377)
      TNP1(1132)
        TNP1(566)
          TNP1(283)
            TNP1(850)
              TNP1(425)
                TNP1(1276)
                  TNP1(638)
TNP1(319)
  TNP1(958)
    TNP1(479)
      TNP1(1438)
        TNP1(719)
          TNP1(2158)
            TNP1(1079)
              TNP1(3238)
                TNP1(1619)
                  TNP1(4858)
TNP1(2429)
  TNP1(7288)

```

```

TNP1(3644)
TNP1(1822)
TNP1(911)
TNP1(2734)
TNP1(1367)
TNP1(4102)
TNP1(2051)
TNP1(6154)

TNP1(3077)
TNP1(9232)
TNP1(4616)
TNP1(2308)
TNP1(1154)
TNP1(577)
TNP1(1732)
TNP1(866)
TNP1(433)
TNP1(1300)

TNP1(650)
TNP1(325)
TNP1(976)
TNP1(488)
TNP1(244)
TNP1(122)
TNP1(61)
TNP1(184)
TNP1(92)
TNP1(46)

TNP1(23)
TNP1(70)
TNP1(35)
TNP1(106)
TNP1(53)
TNP1(160)
TNP1(80)
TNP1(40)
TNP1(20)
TNP1(10)
TNP1(5)
TNP1(16)
TNP1(8)
TNP1(4)
TNP1(2)
TNP1(1)

```

1

&gt;&gt;&gt;

### 19.5.10 Connection With Termination Arguments

The subject of program termination is of immense practical interest for those programs that are deployed in critical devices (e.g., device drivers). In CS 3100, you will learn that in general, program termination cannot be mechanically checked. That is, one cannot write a program  $P$  that takes an arbitrary program  $X$  and checks whether  $X$  always halts.

However, in practice, one can often solve program termination. Read about Terminator in the article below, where the practical impact of attacking the halting problem head-on (for commonly occurring special cases) is mentioned:

<http://www.zdnet.com/why-the-blue-screen-of-death-no-longer-plagues-windows-users-7000021327/>

This article shows that while theory teaches us things such as “it is impossible to do X in general,” in special cases, we can AND MUST do X (Here, X is program termination detection.) In general:

- If a problem is UNDECIDABLE (can’t tell whether the computation will halt with an answer or loop), don’t run away from it .... but run TOWARDS it, finding special cases that can be decided
- If a problem is NP-Complete (exponential, for all practical purposes), don’t run away from it ... but run TOWARDS it, finding special cases that still run in Polynomial time.

Such is the real import of theory.



# Bibliography

- [1] [http://en.wikipedia.org/wiki/Boolean\\_algebra](http://en.wikipedia.org/wiki/Boolean_algebra).
- [2] Lewis Carroll. Lewis carroll's puzzles. <http://tinyurl.com/Gerald-Hiles-Lewis-Carroll>.
- [3] Claude Shannon's MS thesis is kept on the class Moodle page under Week 3. Its URL was also given on a Wikipedia site on Claude Shannon: <http://dspace.mit.edu/bitstream/handle/1721.1/11173/34541425.pdf?sequence=1>.
- [4] <http://www.270towin.com/>.
- [5] <http://research.microsoft.com/en-us/people/gonthier/>.
- [6] Halmos, Paul R. *Naïve Set Theory*. Van Nostrand, 1968.
- [7] Introduction to Logic. <http://logic.philosophy.ox.ac.uk/>.
- [8] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik. Replacing testing with formal verification in intel coretm i7 processor execution engine validation. In *CAV'09*, pages 414–429, 2009.
- [9] Lorraine Lica. <http://home.earthlink.net/~llica/wichthat.htm>.
- [10] Loeckx, J. and Sieber, K. *The Foundations of Program Verification: Second Edition*. John Wiley & Sons, 1987.
- [11] [http://en.wikipedia.org/wiki/Mastermind\\_%28board\\_game%29#Studies\\_on\\_Mastermind\\_complexity\\_and\\_the\\_satisfiability\\_problem](http://en.wikipedia.org/wiki/Mastermind_%28board_game%29#Studies_on_Mastermind_complexity_and_the_satisfiability_problem).

- [12] [http://en.wikipedia.org/wiki/Register\\_allocation](http://en.wikipedia.org/wiki/Register_allocation).
- [13] F. Ruskey, C. D. Savage, and S. Wagon. The search for simple symmetric venn diagrams. *Notth Amer. Math. Soc.*, 53:1304–1311, 2006.
- [14] Alan Tucker. *Applied Combinatorics*. Wiley, New York, 1980.
- [15] The University of Victoria website <http://www.theory.csc.uvic.ca/~cos/inf/comb/SubsetInfo.html#Venn>.
- [16] Mitchell Wand. *Induction, Recursion, and Programming*. Elsevier Science, August 1980.
- [17] [http://en.wikipedia.org/wiki/Four\\_color\\_theorem](http://en.wikipedia.org/wiki/Four_color_theorem).
- [18] The Wolfram website <http://mathworld.wolfram.com/VennDiagram.html>.