Discrete Structures via Circuits, BDD, SAT, SMT, and Functional Programming in Python

Ganesh Gopalakrishnan, Bruce Bolick, Tony Tuttle,
Charles Jacobsen and Swetha Machanavajhala

January 21, 2013

Contents

1 Course Design and Implementation **1**

2 Python, Sets, Induction, Recursion: A First Look **9**

2.1 Chapter Material: History/Applications 9

2.1.1 Sets, Sequences, Strings, Tuples, Lists 10

2.1.2 Life. . . Represented 10

2.1.3 Venn Diagrams 11

2.1.4 Lambdas 12

2.1.5 The Python Language 12

2.2 The Baby Python of Sets, Tuples, Lists 14

2.2.1 A Tour of the Python Subset of Interest 14

2.2.2 Indentation 14

2.2.3 Characters 15

2.2.4 Strings 15

2.2.5 Lists 16

2.2.6 Sequences 19

2.3 Sets 19

2.4 Reviewing Basics Using Python 22

2.4.1 Arithmetic Progressions 22

2.4.2 Permutations 24

2.4.3 Combinations 24

2.5 Venn Diagrams 26

2.6 Induction/Recursion: Sides of Same Coin 28

2.6.1 Recursion 29

2.6.2 Polynomial versus Exponential Growth 29

2.6.3 Sets and Tuples in Mathematics 30

2.7 *Review and Labs* 31

3 Boolean Basics and Propositional Potpourri 41
3.1 Chapter Material: History/Applications 42
3.1.1 History of Boolean Functions in Computing 42
3.1.2 History of Digital Design and Circuit Simulation 43
3.1.3 History of Propositional Reasoning 43
3.2 Truth Values and Functions 44
3.2.1 Functions, and Boolean Functions 45
3.2.2 Mathematics of Functions, Signature 47
3.2.3 How Many Functions over One Input Exist? 50
3.2.4 How Many Functions over Two Inputs Exist? 50
3.2.5 Counting All Possible N-input Truth Tables 52
3.2.6 Syntax for Boolean Formulae 53
3.3 Uses of Universal Gate Sets 53
3.3.1 Realizing Inclusive and Exclusive Or using Nands . . . 53
3.3.2 An "Implication Gate" 53
3.3.3 A 2-to-1 Mux Using Nand Gates 55
3.4 Realizing all 2-input Functions via Muxes 55
3.4.1 A 2-to-1 Multiplexor or an `if-then-else` gate 56
3.4.2 Creation of a general 2-input function module 58
3.4.3 Counting Boolean Functions: the Mux Approach . . . 63
3.5 Boolean Identities 63
3.6 Proof of Universality of Gates 66
3.6.1 Universality Condition 67
3.6.2 Showing that XOR is *Not* Universal 67
3.7 Propositional Reasoning 68
3.7.1 Axiomatization of Propositional Logic 68
3.7.2 Validity, Satisfiability, Contradiction 70
3.8 Sets Using Characteristic Predicates 71
3.9 *Review and Labs* 72

4 The Higher Order of Programming 87

5 Putative Proper Proof Principles 89

6 Googols of Graphs 91

7 Boosted Boolean Brawn 93

8 Combinatorial Cornucopia 95

9 Rocking Recursion and Impressive Induction 97

10 Probable Certainties and Certain Probabilities 99

Chapter 1

Course Design and Implementation

This book is on "Discrete Structures" which traditionally means specific skill sets ranging over Logic, Combinatorics, Recursion, Induction, Functional Programming, Probability, and Graphs. Ours is a modern take on Discrete Structures motivated by the need to tell the real story: *these topics are not esoteric diversions* that may prove useful at some future date—rather, *"living and breathing" subject areas that are fundamental to all aspects of computer science*—theoretical and practical.

We aim to rescue this topic from its unfortunately much maligned status to one of being truly enjoyable, insightful, intuitive, and interactive! We believe that the maligned status is due to existing textbooks which seem to hold on to lofty (and often unattainable, especially in real practice today) standards of rigor. In our opinion, practicing discrete structures problems merely on paper and relying only on human thought to resolve questions is making things unnecessarily hard on oneselves. In fact, it is more helpful if students are told that writing math is like writing programs in many respects, and also crucially different in many respects (and tell them these differences). Mathematical structures are built much like data structures in computer programs. Similar to functions in programming languages, mathematical functions take input parameters, manipulate the data according to precise rules, and produce outputs. We will emphasize the similarities, and use the Python programming language to demonstrate the concepts we learn about. Along the way, you will also become reasonably proficient in Python—an added bonus! And in our humble opinion, this is precisely what a student experiencing discrete structures for the first time is looking for: intuitions and connections, not artificially mandated rigor. Rigor is essential, but it comes afterwards (in later passes over this material).

Although writing math is quite similar to writing programs, the language of math is quite powerful, and it takes a lot of getting used to. One of the main virtues of a well chosen mathematical language is that it comes without heavyweight IDEs, core dumps, compiler errors, system requirements, and the other clutter of computer software. Math is ultra-compact! This innately helps us explore large subsystems more efficiently.

But the above virtues of math are also its limitations. Things written in mathematical notations must be viewed with suspicion until one has checked their correctness exhaustively over a small domain—one of the main lessons coming out of *model-checking*. Unfortunately, very often one cannot run most mathematical constructions, and without running the definitions, one must rely upon good (human) proof-readers for debugging. Unfortunately, this manual process is non-scalable. There is a perpetual short supply of good and capable reviewers: it seems that nowadays, more want to write something, and fewer want to read others' writings!

So instead of peer reviews, we need to increasingly rely upon automated checking tools. This underscores why we will strive to employ subsets of Python that are closer to math (often called *purely functional*) but is runnable at least over finite domains. Although no programming language is perfect, Python is widely available, and allows us to use programmatic notation to capture math pretty closely. Last but not least, studies show that when students learn a topic *actively* (through hands-on experimentation) they tend to retain the knowledge longer.

Motivation for this book

Our approach requires largely ignoring past books in this area and starting afresh. The motivation to do so is best phrased by Allen Downey in his preface to the book "ThinkStats: Probability and Statistics for Programmers" [4]:

> How we wrote this book: When people write a new textbook, they usually start by reading a stack of old textbooks. As a result, most books contain the same material in pretty much the same order. Often there are phrases, and errors, that propagate

> from one book to the next; Stephen Jay Gould pointed out an example in his essay, "The Case of the Creeping Fox Terrier"...

I also want to bring into this book some of the fruits of modern research on software and hardware correctness checking based on formal approaches. This is the reason why I have included the following topics:

Circuits: By presenting Boolean principles using circuits, one obtains a visual (and hence tangible) representation of propositional assertions. One can mix and match, for instance doing proofs by contradiction using circuits! We shall illustrate this by translating some of Lewis Carroll's puzzles into circuits. There are other reasons as well:

- If taught the right way using the right tools, circuits are fun (we will use the `tkgate` tool for simulation).
- There is no need for computer scientists to declare "I am a hardware type" or "I am a software type." Doing such early self-selection of professional preferences (and biases) is especially harmful in this era where computing is increasingly going into the power-constrained hardware-aware and locality-aware computing modality.

BDD: Binary Decision Diagrams have been in the formal methods research for over 25 years. Yet, as Knuth observes in his Volume IV(a), very few textbooks introduce BDDs in a fun and tangible way. Knuth puts BDDs as "one of the most important of data structures to be discovered in the last 25 years." We will show BDDs in the following interesting ways:

- Read out sum-of-product and product-of-sum forms out of BDDs
- Directly realize BDDs using multiplexors
- Use BDDs to represent (and count) the number of ways to 4-color the map of the USA
- Show proofs by contradiction using BDDs.

SAT/SMT: Boolean Satisfiability is the canonical NP-complete problem in computer science. SAT tools have become three orders of magnitude faster over the last decade, turning them into industrial work-horses. We will introduce SAT in many interesting ways:

- Show you how to formulate games (such as Sudoku and N-Queens)

- Mention modern uses of SAT in software/hardware verification, and even in user interface design (*e.g.*, Microsoft Office Excel's "Flash Fill" feature[5])
- Formulate graph-problems using SAT

SMT stands for Satisfiability Modulo Theories, and extends the idea of SAT to richer theories, such as the theory of integers, bit-vectors, strings, etc. The added power will allow us to encode very interesting search problems using a Python interface to a state-of-the-art SMT solver from Microsoft Research called `z3py`.

Functional Programming: Teaching about functions in general, and about induction and recursion in particular, is best done by having students write simple recursive programs. As Mitch Wand has observed [18], induction and recursion are two sides of the same coin. The need to teach functional programming has been mentioned by several notable researchers already. Here are some illustrations:

- One can teach how to calculate the number of shuffles of 2 decks of k cards.
- One can write a functional program in a recursive style, recursing upon 2 decks of $k-1$ cards, and building up the final answer.
- One can prove the program by induction (thus proving the combinatorics). One can also run the program and count the number of shuffles.
- One can immediately motivate one practical connotation: shuffling card decks is similar to the generation of interleavings of sequential programs that are run concurrently.

A Two-Pass Approach

Telling our story on Discrete Structures in the above described fashion requires a completely non-traditional approach. It first of all requires that we present *contexts* in which these ideas and topics can be gainfully employed. It requires that we present *tools* that can illuminate the material as well as show its practical power. It also requires an approach where we show the *connectedness* of the topics in this realm—and not teach these topics as if they were separate.

Showing these connections requires that we present the topics first through an *overview pass* in which we only touch upon the *main highlights of the topic.* The initial chapters of this book will be written with the overview pass in mind. This will allow us to rapidly discuss many topics and interplay between them.

Later chapters will consist of a *rounding pass*. In the rounding pass, you will get to appreciate the depths of these topics and cultivate more abstract thinking capabilities. The time gap between the overview pass and rounding pass will also permit your brain to be working subconsciously in the interim. In short, this book takes a spiral approach where the outer spirals frequently refer to the inner ones.

Organization

See Figure 1.1 on Page 8 for an overview of the course. Here is a rough (*and evolving!*) description. In Chapter 1 entitled Course Design and Implementation, we introduce our basic approach to re-designing the traditional and much maligned Discrete Structures course. Some of our guiding principles are these: (1) The best form of learning is teaching. This applies also to the students, who will, in our approach, be "teaching" a computer to concretely represent the abstract knowledge they gain and make the computer do something for them. This may increase the "half life" of knowledge that tends to rapidly decay over time in students' mind. (2) The best way to introduce math is "just in time"—precisely when some context demands. We strive to create a reasonable amount of context for each idea put forth. (3) Programming and doing mathematics exercise similar principles of organizing information and being precise, with different degrees of abstraction and tool support. One activity does not supplant the other; they are best woven together. (4) The subject we are about to study is not ossified knowledge; rather, it is a living/breathing entity that is constantly growing by absorbing the latest from the research world. We strive to transfer some of these research tools and the thinking itself into this course. (5) Ideally one must learn hardware and software, and how the uses of terminology in one area (*e.g.*, Boolean gates in hardware) relate to things one does in the other (*e.g.*, proofs by contradiction). To help bridge this gap, we do fun things such as building a circuit mirroring the reasoning during proof by contradiction. In the modern context, every computer scientist must be comfortable dealing with simple circuit design situations as well as concurrent programming sitations. Through combinatorics done on decks of cards, we illustrate how the number of interleavings in a concurrent program grow with the number of decks and the number of cards per deck.

In Chapter 2 entitled Python, Sets, Induction, Recursion: A First Look, we introduce a core set of ideas that we will need throughout the course, and in particular: The mathematics of Sets and Tuples; programming in Python including Lambda, Reduce, Filter, and Set Comprehensions; Sequences; the basics of Permutations and Combinations; and Venn Diagrams. We will also briefly look at recursive definitions and mathematical induction.

In Chapter 3 entitled Propositional Potpourri and Boolean Basics, We introduce the basics of the propositional style of thinking and also the Boolean view of things. We will mainly focus on Boolean methods in an *introductory manner*. Later chapters will expand on this material.

In Chapter 4 entitled The Higher Order of Programming, we introduce the concept of lambdas and provide a taste for higher order programming. This helps us to revisit and properly understand some of the material seen earlier—plus present beautiful definitions of basic concepts such as the *Powerset*.

In Chapter 5 entitled Putative Proper Proof Principles, we present some commonly prescribed proof approaches. We will give the students the ability to check many of their proofs so that they know they are submitting something that is correct (and our burden of grading is minimized!).

In Chapter 6 entitled Googols of Graphs (where Googol is the word used to denote 10^{100}—and is the root word for "Google"), we present notions around graphs. We introduce the idea of relations, and conduct various tours on graphs—again showing how constraint solvers (*e.g.*, Satisfiability Modulo Theory or Decision Procedure tools) can help express and solve many of these problems rather elegantly (*e.g.*, Hamiltonian Cycles in a few lines of code).

In Chapter 7 entitled Boosted Boolean Brawn, we show why Boolean reasoning is now a "workhorse"—allowing us to handle large problem instances. We present BDDs—referred to by Knuth as "one of the most important of data structures in the last 25 years." We present techniques to solve popular games (you may write many of these solvers easily).

In Chapter 8 entitled Combinatorial Cornucopia, we present combinatorial arguments, taking advantage of induction and recursion as well as constraints in many problems (*e.g.*, the number of 4-colorings of the US map using BDDs).

In Chapter 9 entitled Rocking Recursion and Impressive Induction, We reinforce the fact that recursion and induction are two sides of the same coin.

In Chapter 10 entitled Probable Certainties and Certain Probabilities, we present basic ideas about probabilities and how one may apply it to designing good Hashing functions.

Books and References

Four free books have provided us considerable insights and also valuable material:

- Mathematics for Computer Science by Eric Lehman et al
- ThinkStats by Allen Downey
- Course notes on induction by Jeff Erickson
- Cartoon Guide to Statistics by Larry Gonick and Wolcott Smith. *This is so good that I recommend buying a copy; it is only around $12 and will give you hours of laughter and real knowledge.*

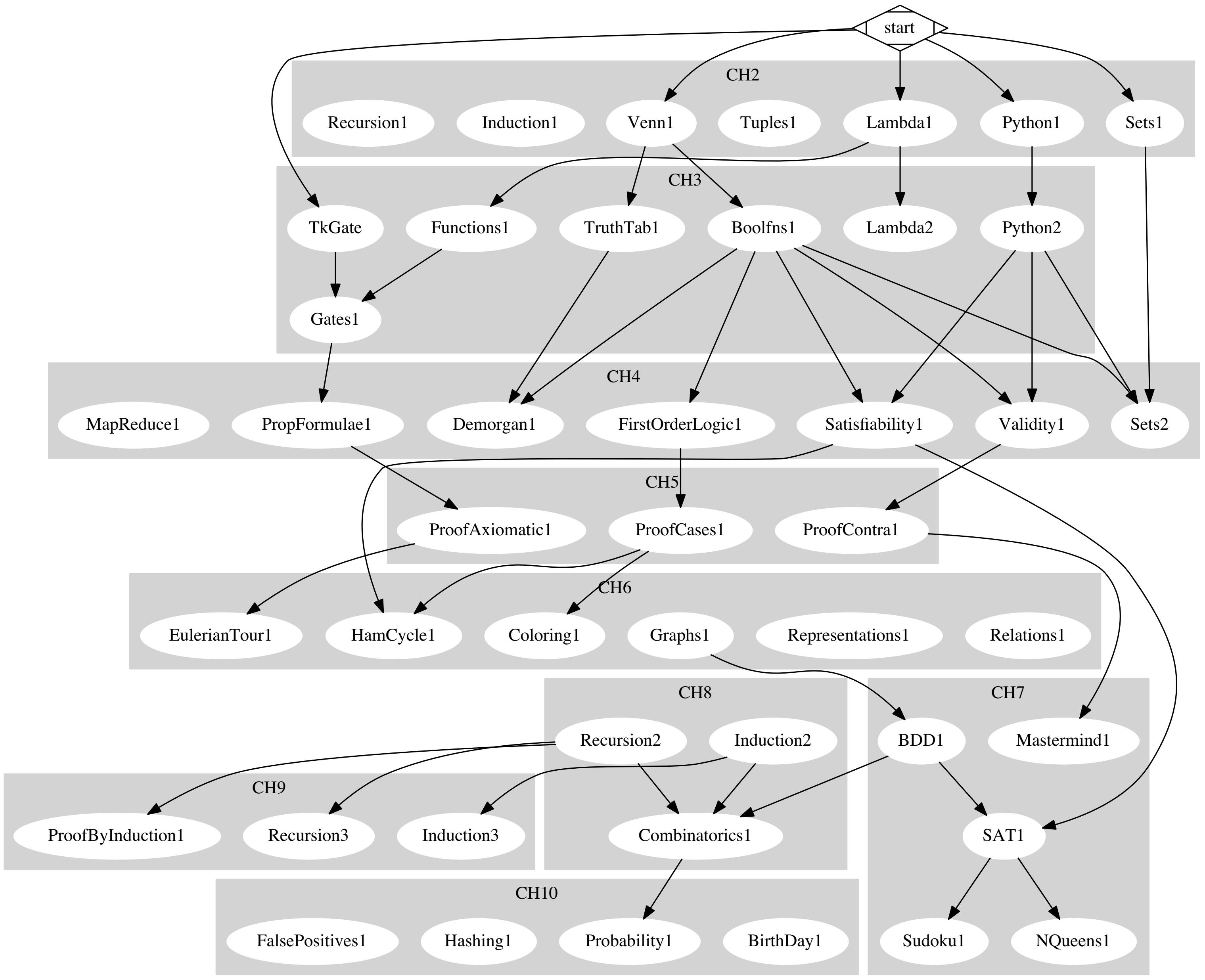


Figure 1.1: Rough Ordering of Book Material

Chapter 2

Python, Sets, Induction, Recursion: A First Look

Because of our interest in making you good at Python (to help you get better at "teaching the computer" how to do things with sets and lists), we will first introduce you to some basic Python. Then we will introduce the basics of sets. We will describe mathematical as well as programmatic approaches to manipulate sets. We will also introduce allied notions such as tuples. We will also introduce lists and operations on them.

2.1 Chapter Material: History/Applications

At the beginning of each chapter, we will provide a few historical details of the material contained in the chapter. Hopefully, this will do some justice toward representing the hard work and creativity of countless past scientists and engineers whose shoulders we stand upon. Knowing computing history[1]also helps us appreciate the fact that seemingly esoteric mathematical ideas have, time and again, crept up into programming practice; this is the single most important reason for studying Discrete Structures (it makes you "future ready"). by the time you are a successful computing professional out in the field in about a decade years or less, you will be well situated to react wisely to "new" ideas being put forth. Instead of viewing new ideas with unwarranted skepticism or being the last to adopt them, you might

[1]A smart aleck once said "knowing history also equips you to recognize mistakes when you make them again!"

actually be able to recognize or even *originate* many of them.

2.1.1 Sets, Sequences, Strings, Tuples, Lists

Sets, sequences, strings, tuples, and lists are all mathematical structures that have found their way into programming (as well as daily) practice. As an example, consider three buttons A, B, and C on some machine. If you want to record whether any button has ever been pushed, you can use a set. Thus, if A has been pushed a few times (one or more times) while B and C have not been pushed, then you can model the situation using a set $\{A\}$. Now if A and C have been pushed, model it using $\{A, C\}$. There is a variation of sets called *bags*[2]that you may look up if you are curious. We will not study bags in this course (thus even if A has been pushed 20 times and C pushed 2 times, we will use $\{A, C\}$. There are of course so many other imaginative ways of using sets.

Continuing with the same example, suppose one presses A, C, A, C, A, A. Then a *sequence* helps record this. We will use two kinds of sequences:

- Sequences of characters, called *strings*. The above example can be modeled as a sequence `"ACACAA"` which happens to be a string.
- Sequences of any data type at all. This is called a *list*. One can represent the above button-pushing example using the list `['A','C','A','C','A','A']`.

Suppose you have to tell someone to press exactly four buttons, and want to record their order. Then you may use tuples to record the situation. Example tuples would be $\langle A, C, A, C\rangle$, $\langle A, B, B, A\rangle$, etc. Sometimes, one uses (A, C, A, C) in lieu of the angle brackets $\langle \ldots \rangle$. We will use these interchangeably.

2.1.2 Life... Represented

Let us now use the above data types to use in a real-life situation: that of representing DNA. The next time you ponder about the meaning of life, why not view it as described in [7] as a "DNA data structure"? In particular, the human genome is

[2]What if you want to remember how many times each button has been pushed but not in what order: then use a *bag* or *multi-set*. We won't talk about bags here on; but it is useful to know that they exist.

- a list of length *25* (representing the 25 chromosomes which include the mitochondrian DNA and the sex chromosomes); and
- each element of the list *base pairs.*
- The base pairs can be modeled in one of two ways:
  - either as a *list* of *pairs* or DNA characters (see below), where the lists are of length around 200 million (written out, this representation looks like `[(A,T),(G,C),(T,A),(C,G),...]`); or
  - as a *single* pair of *strings*, with each string being about 200 million DNA characters in length (written out, this representation looks like `("AGTC","TCAG")`).
- Each DNA character is a member of the *set* $\{A, C, T, G\}$ representing bases.

Life (represented) is, to a first approximation, a sequence (list) of pairs of sequences (base strings) such as this:

```
[ ("AGTCGA","TCAGCT"), ("AGTCGA","TCAGCT"),
  ("AGTCGA","TCAGCT"), ("AGTCGA","TCAGCT") ]
```

No wonder biologists love Python as a language to study and manipulate DNA, because Python has excellent data structures for representing the above types of mathematical objects (or programming language data structures). There is a package called *BioPython* (`http://www.biopython.org`) that is extremely popular in studying genetics, and is based on Python.

2.1.3 Venn Diagrams

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 [10], a web article: The distinction The distinction between *"Tiffany likes 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 2.1 on Page 12. 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*

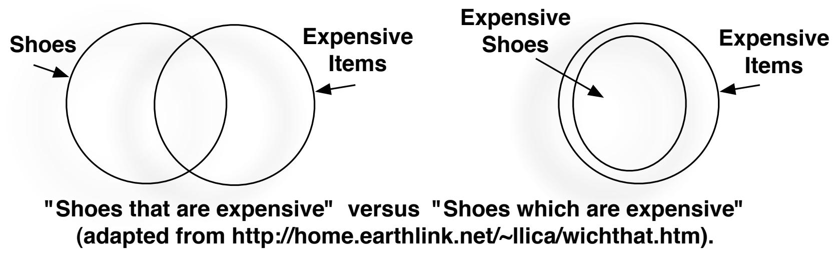

Shoes
Expensive Items
Expensive Shoes
Expensive Items
"Shoes that are expensive" versus "Shoes which are expensive"
(adapted from http://home.earthlink.net/~llica/wichthat.htm).

Figure 2.1: "That" versus "Which" in English usage

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.

2.1.4 Lambdas

Lambdas were formulated as a mathematical concept for *anonymous functions* by Alonzo Church in the 1930s [9]. It now forms part of mainstream programming languages. A C++ example involving Lambdas (from [13]) is given in Figure 2.2 on Page 13. In this example, the Lambda syntax looks like `[&evenCount] (int n) { ..lambda body.. }`.

2.1.5 The Python Language

According to [17], the Python language was developed by Guido van Rossum as a "hobby programming language" beginning 1989. In 1999, van Rossum submitted a funding proposal to DARPA called *Computer Programming for Everybody*, in which he further defined his goals for Python: *an easy and intuitive language just as powerful as major competitors open source, so anyone can contribute to its development code that is as understandable as plain English suitability for everyday tasks, allowing for short development times.* The wide popularity of Python is reflected in its wide international adoption. It is now one of the principal teaching languages at major universities, and dozens of widely used packages are written in it.

```
// even_lambda.cpp
// compile with: cl /EHsc /nologo /W4 /MTd
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   // Create a vector object that contains 10 elements.
   vector<int> v;
   for (int i = 0; i < 10; ++i) {
      v.push_back(i);
   }

   // Count the number of even numbers in the vector by
   // using the for_each function and a lambda.
   int evenCount = 0;
   for_each(v.begin(), v.end(), [&evenCount] (int n) {
      cout << n;

      if (n % 2 == 0) {
         cout << " is even " << endl;
         ++evenCount;
      } else {
         cout << " is odd " << endl;
      }
   });

   // Print the count of even numbers to the console.
   cout << "There are " << evenCount
        << " even numbers in the vector." << endl;
}
--
0 is even
1 is odd
2 is even
...
```

Figure 2.2: The use of Lambdas in C++

2.2 The Baby Python of Sets, Tuples, Lists

Figure 2.3: Baby Python, courtesy of the National Zoo of Washington D.C. Image taken from `http://www.zooborns.com/zooborns/2009/06/national-zoo-python-baby-.html`.

We will now take a second pass over the concepts we have seen so far, and reinforce our understanding by studying some Python programming.

2.2.1 A Tour of the Python Subset of Interest

We will be running Python 2.7 on Windows or Unix. If you are unfamiliar with Unix, please take a refresher (*e.g.*, `http://www.ee.surrey.ac.uk/Teaching/Unix/`). Then go try out Python Version 2.7 (available as `python3` on our CADE machines).

An extremely good set of tutorials on Python (with plenty of good exercises) are available from `http://www.python-course.eu/index.php`. Some additional details are at `http://www.python.org` and `http://docs.python.org/py3k/`. For more online courseware, see for example [12].

2.2.2 Indentation

Indentation rules of Python are very important to understand. You may read about Python's recommended indentation styles (and coding-styles in general) at `http://www.python.org/doc/essays/styleguide.html`. In the sequel below, we present examples in multiple columns, to conserve space.

2.2.3 Characters

Computer science is often concerned with recognizing the patterns in *strings* which are comprised of *characters*. Characters may also be viewed as strings of length 1. Characters are entered and manipulated as follows against the Python prompt `>>>`:

```
>>> 'a'
'a'              >>> '"'
                 '"'
>>> "a"
'a'              >>> ord('a')
                 97
>>> """a"""
'a'              >>> chr(97)
                 'a'
>>> "'"
"'"              >>> chr(ord('a')+1)
                 'b'
```

2.2.4 Strings

Strings are arrays of characters, and can be entered and manipulated as shown below. Note how "raw" strings are entered and the convention pertaining to \, esp. at the end of a string.[3]

```
>>> t='abcdef'                 >>> r"a\a"
                               'a\\a'               >>> for x in "abcd":
>>> t                                                    print(x)
'abcdef'                       >>> s=r"a\a"         .. prints a, b, c, d
                               >>> s
>>> t[::2]                     'a\\a'               >>> len("abcd")
'ace'                                               4
                               >>> s=="a\a"
>>> t[::-1]                    False
'fedcba'                                            s= """hello, this is
                               >>> s==r"a\a"        ... a multi-line sentence.
>>> t='abcdef'                 True                 ... """

>>> t[2:]                      >>> s[0]             >>> s
```

```
'cdef'

>>> t[:2]
'ab'

>>> t[:2] + t[2:]
'abcdef'

>>> t[:]
'abcdef'

>>> t[:2] + 'z' + t[3:]
'abzdef'

>>> "aaa"
'aaa'

>>> "a\a"
'a\x07'

>>> "a\a"
'a\x07'

>>> "a"*5
"aaaaa"
```

```
'a'

>>> s[1]
'\\'

>>> s[2]
'a'

>>> s[0:]
'a\\a'

>>> s[1:]
'\\a'

>>> s[2:]
'a'

>>> t=r"a\b"

>>> t
'a\\b'

>>> t[::2]
'ab'
```

```
s
'hello, this is \na multi-line sentence
```

2.2.5 Lists

Lists are versatile data structures capable of representing sequences and trees of items. Python supports a rich collection of list manipulation functions, and of course type-conversion functions that allow data structures to be interconverted. We will introduce `lambda`s or anonymous functions also below. This topic will receive a more in-depth discussion later in Chapter 4; however, some familiarity gained here cannot hurt.

```
>>> lst = ['a', 'b']

>>> lst[0]
'a'
```

```
>>> alias = n09

>>> alias
```

[3]About raw strings ending in backslash, see `http://stackoverflow.com/questions/647769/why-cant-pythons-raw-string-literals-end-with-a-single-backslash`

```
>>> [x for x in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> n09 = [x for x in range(10)]

>>> n09
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> n09[0]=100

>>> n09
[100, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> n09 = [x for x in range(10)]

>>> n09
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

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

>>> n09[0] = 100

>>> alias
[100, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> n09 = [x for x in range(10)]

>>> n09[2:]
[2, 3, 4, 5, 6, 7, 8, 9]

>>> acopy = n09[:]  # Make a deep copy

>>> acopy
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> n09[0] = 100
>>> acopy
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> f = lambda x: x+1

>>> f
f
<function <lambda> at 0x1005eb5a0>

>>> list(map(f, acopy))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> ["a"*i + "b"*j + "c"*k for i in range(3) for j in range(3)
        for k in range(3) if ((i != 1) or (j == k)) ]

[’’, ’c’, ’cc’, ’b’, ’bc’, ’bcc’, ’bb’, ’bbc’, ’bbcc’, ’a’, ’abc’, ’abbcc’, ’aa’,
   ’aac’, ’aacc’, ’aab’, ’aabc’, ’aabcc’, ’aabb’, ’aabbc’, ’aabbcc’]

>>> f = lambda x, y: x+y

>>> acopy
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> import functools

>>> functools.reduce(f, acopy)
functools.reduce(f, acopy)
```

```
45

>>> odd = lambda x: x%2 == 1

>>> odd(2)
False

>>> odd(3)
True

>>> list(filter(odd,acopy))
[1, 3, 5, 7, 9]

>>> [[1,2],3][0]
[1,2]

>>> [[1,2],3][0][1]
2

>>> [a,b] = [1,2]

>>> a
1

>>> b
2
```

Arguments of Lambda: Please note that Lambdas by default take a *single* argument. Thus, in the definition below, notice that `foo` takes a *single* pair `(1,2)`, and not the two separate arguments of `1` and `2`:

```
>>> foo = lambda(x,y): x

>>> foo(1,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: <lambda>() takes exactly 1 argument (2 given)

>>> foo((1,2))
1
```

More examples are provided in §2.7. Keep this in mind when you read Chapter 4 where we will define a `mux41` in terms of three `mux21`s.

2.2.6 Sequences

Sequences (or tuples) are simpler data structures than lists, and enjoy certain handy properties, including being hashable. They are usually fixed-length sequences where specific positions have a prescribed meaning. Note that in mathematics, we often write $\langle a, b\rangle$ and even sometimes (a, b)—both mean the same.

```
(1,2)
(1, 2)                   >>> ((1,2),3)[1]
                         3
>>> (1,2)[0]
1                        >>> (a,b) = (1,2)

>>> ((1,2),3)[0][0]      >>> a
1                        1

>>> ((1,2),3)[0][1]      >>> b
2                        2

>>> mixList = [ [], (1,2), (), "aa", ("aa", "bb", 2), 12, [1, (1,2)]]

>>> list(zip([1,2], [4,5]))
[(1, 4), (2, 5)]
```

2.3 Sets

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 $Nat = \{0, 1, 2, \ldots\}$). *By convention, domains are non-empty, but can be infinite (*e.g.*, Nat is infinite).* Now we can define sets *over* D. Each set contains some, none, or all of the members of D. For example, $s_1 = \{1, 2\}$, $s_2 = \{1, 3, 2\}$, $s_3 = Nat$, $s_4 = \{0, 2, 4, 6 \ldots\}$, $s_5 = \{1, 3, 5, 7 \ldots\}$, and $s_6 = \{\}$ are all sets of natural numbers (sets over Nat). Note that s_3, s_4, and s_5 are infinite while s_6 is empty.

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!)

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!
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} | {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 $\triangle$. $s_1 \triangle s_2$ stands for $(s_1 \setminus s_2) \cup (s_2 \setminus s_1)$.
  Example: $\{1,2\} \triangle \{1,3\}$ or `{1,2} ^ {1,3}`
  resulting in $\{2,3\}$ or `{2,3}`.
  Example: $\{1,2\} \triangle \{4,3\}$ or `{1,2} ^ {4,3}`
  resulting in $\{1,4,2,3\}$ or `{2,1,3,4}`.
- The *complement* of a set is defined with respect to a domain. Its mathematical operator is written as an "overbar." For instance, with respect to $D = Nat$, $\overline{s_4} = s_5$. 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*.[4]

- 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 $\subset$ (`>`).
  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 thatb by default, `range()` creates a list):

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

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

[4]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."

```
True

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

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

2.4 Reviewing Basics Using Python

We are soon to embark on many concepts you will need (most ought to be a review from high-school). In this section, we will help you review these with the help of Python.

2.4.1 Arithmetic Progressions

$$\sum_{k=1}^{N} k = N(N+1)/2$$

Proof:

- *Case N is even:* We notice that there are $N/2$ additions we can perform in an "outside-inside" fashion, where the sums are:

  $$(1+N), (2+(N-1)), (3+(N-2)), \ldots, (N/2+(N/2+1))$$

  Thus, the total value is $N(N+1)/2$.
- *Case N is odd:* Call the middle element of $1, \ldots, N$ "p." It is guaranteed to be $p = (N+1)/2$. Now write the summation as $1 + \ldots + p/2 + p/2 + \ldots + N$. There are now $N/2$ additions, where the sums are:

  $$(1+N), (2+(N-1)), \ldots, (p+p).$$

  This case also gives rise to $N(N+1)/2$.

Review Using Python:

```
>>> help(range)
Help on built-in function range in module __builtin__:

range(...)
```

```
    range([start,] stop[, step]) -> list of integers

    Return a list containing an arithmetic progression of integers.
    range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to 0.
    When step is given, it specifies the increment (or decrement).
    For example, range(4) returns [0, 1, 2, 3].  The end point is omitted!
    These are exactly the valid indices for a list of 4 elements.

>>> from functools import reduce

>>> help(reduce)
Help on built-in function reduce in module _functools:

reduce(...)
    reduce(function, sequence[, initial]) -> value

    Apply a function of two arguments cumulatively to the items of a sequence,
    from left to right, so as to reduce the sequence to a single value.
    For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates
    ((((1+2)+3)+4)+5).  If initial is present, it is placed before the items
    of the sequence in the calculation, and serves as a default when the
    sequence is empty.

>>> help(filter)
Help on built-in function filter in module __builtin__:

filter(...)
    filter(function or None, sequence) -> list, tuple, or string

    Return those items of sequence for which function(item) is true.  If
    function is None, return the items that are true.  If sequence is a tuple
    or string, return the same type, else return a list.

>>> range(1,11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> addf = lambda x,y:x+y

>>> reduce(addf,range(1,11))
55

>>> (10*(10+1))/2
55
```

2.4.2 Permutations

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
```

2.4.3 Combinations

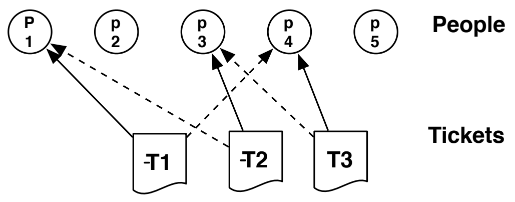


Figure 2.4: With five people and three tickets, 3! ticket swaps are possible for each assignment of people chosen to have tickets in the first place

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 $N.(N-1).(N-2).\ldots.(N-k+1)$ 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 2.4 for a situation where for some choice of ticket holders, the ticket swaps are shown. In other words, if we go by the formula

$$N.(N-1).(N-2).\ldots.(N-k+1),$$

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 $N.(N-1).\ldots.(N-k+1)\ /\ (k!)$, or the same as $N!/(k!).(N-k)!$. In other words, reading

$$\binom{N}{k} \ = \ N.(N-1).\ldots.(N-k+1)\ /\ (k!) \ = \ N!/(k!).(N-k)!$$

In Figure 2.4, 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
```

2.5 Venn Diagrams

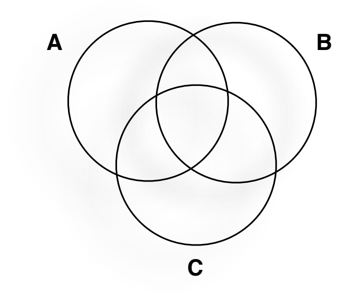


Figure 2.5: The Familiar Venn Diagram of 3 sets

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 2.5 on Page 26) 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 [16, 19]. More formally, an order-N Venn diagram is a collection of simple closed curves in the plane such that

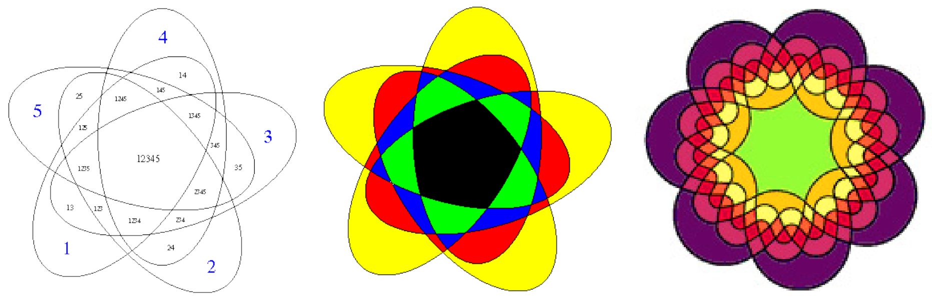


Figure 2.6: Venn Diagrams of order 5 (left); of order 5 with regions colorized (middle); and order 7 (right). Images courtesy of `http://mathworld.wolfram.com/VennDiagram.html` and `http://www.theory.csc.uvic.ca/~cos/inf/comb/SubsetInfo.html\#Venn`.

1. The curves partition the plane into connected regions, and
2. Each subset S of $\{1, 2, \ldots, N\}$ corresponds to a unique region formed by the intersection of the interiors of the curves in S [14].

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 2.6 on Page 27, where the colors represent the number of regions included inside the closed curves.

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). This dervation comes from the use of the Binomial theorem:

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

Alternate Derivation of the Number of Venn Diagram Spaces: 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
```

2.6 Induction/Recursion: Sides of Same Coin

Mathematical induction lets you prove facts over infinite sets. Suppose you are asked to sum $1 \ldots N$. In high-school you have probably learned this trick:

- There are N numbers in the sequence.
- Adding from either end — 1 with N, then 2 with $N - 1$, etc., results in the same sum. In this addition:
  - If N is even, there are $N/2$ pairs being added.
  - If N is odd, represent it as given *except* in place of the middle number m, write $m/2$ twice (thus making the sequence even in length). Then also, there are $N/2$ pairs being added.
  - Each pair adds up to $(N + 1)$.
- Thus the sum is $N(N + 1)/2$.

Suppose we want to state this as a general theorem. Then, *induction* is a style of proof that allows you to prove it. We will see much more about induction later. But let us get warmed up with a simple inductive proof for this summation.

- *Write the basis case*: If $N = 1$, we are summing 1 through 1, which is 1 which $(1 + 1)/2$.
- *Write the induction hypothesis*: Suppose true of summation 1 through N.
- *Show the induction step*: Summation 1 through $(N + 1)$ is summation 1 through N is added to $N + 1$.

$$(N(N + 1)/2) + (N + 1) = (N + 1)(N + 2)/2.$$

- Thus proved.

2.6.1 Recursion

The structure of the above inductive argument is reflected in a recursive program that sums sequences of natural numbers starting at 1.Recursion "works" because (i) all the recursive calls "work,", and (ii) the basis cases also "work." This reasoning style is what inductive proofs also follow.

```
def sum(N):
  if (N <=1):
    return 1
  else:
    return N + sum(N-1)
```

2.6.2 Polynomial versus Exponential Growth

It is important that you learn to recognize exponential growth versus polynomial growth. To give you some grounding on this important topic, here are some Python routines that help you practice. In general, functions of the kind k^N are said to grow exponentially (where k is a constant and N is the growth parameter, or "input"). For example, think of 2^N. Similarly, functions of the kind N^k are said to grow polynomially (where k is a constant and N is the growth parameter, or "input"). For example, think of N^2. Here, now, are some examples:

```
# Poly growth
>>> [n**2 for n in range(1,33)]

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289,
 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 841, 900, 961, 1024]

>>> [2**n for n in range(1,11)]
[2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384,
 32768, 65536, 131072, 262144, 524288, 1048576, 2097152, 4194304, 8388608,
 16777216, 33554432, 67108864, 134217728, 268435456, 536870912, 1073741824,
 2147483648, 4294967296]

# Even a small "k" causes a huge growth with "exponential" growth
>>> [1.01 ** n for n in range(1,10002)][10000]
1.6521869983009303e+43

# ...whereas with polynomial growth, things are nicely bounded
>>> [n ** 1.01 for n in range(1,10002)][10000]
10965.889404963658
```

2.6.3 Sets and Tuples in Mathematics

Sets: Sets are crucially important: in fact, *all* of mathematics can be derived from set theory (more precisely, the Zermelo-Frankel Set theory with the axiom of Choice – often abbreviated ZFC). Even numbers can be viewed as sets. For example, consider natural numbers (the set $\{0, 1, 2, \ldots\}$):

- 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 $\{\{\}, \{\{\}\}, \{\{\}, \{\{\}\}\}\}$.

Later in this course when we deal with sets of natural numbers such as $\{0, 2, 7\}$, we won't be concerned about the fact the numbers 0, 2, and 7 themselves can be viewed as sets. We will be viewing things at a *higher level of abstraction* as *sets of numbers.*

Tuples: Tuples are fixed-length sequences, and written for example as $\langle 0, 1\rangle$, $\langle 2, 0, 1\rangle$ or equivalently as $(0, 1)$ and $(2, 0, 1)$ (respectively). The number of things in a tuple is its *arity* A 2-ary tuple (arity of 2) is called a *pair.* In mathematics, tuples of higher arity than 2 are really represented using pairs; for example, $(2, 0, 1)$ is represented as $(2,(0, 1))$.

Tuples as Sets: There are many ways to represent tuples as sets. According to Naïve Set Theory [6], one can view a tuple $\langle a, b\rangle$ as $\{a, \{a, b\}\}$. But as [15] presents, the accepted modern definition by Kuratowski apparently is $\langle a, b\rangle$ as $\{\{a\}, \{a, b\}\}$.

The Axiom of Extensionality: Given two sets S_1 and S_2, according to the **axiom of extensionality** — or the idea of **extensional equality** of sets — $S_1 = S_2$ exactly when $S_1 \subseteq S_2$ and $S_2 \subseteq S_1$. Here, $\subseteq$ means "subset"— defined on Page 20. The same idea applies to tuples also: given two tuples T_1 and T_2, they are considered to be equal if their respective positions are equal. This is one reason why tuples are represented as $\{a, \{a, b\}\}$ or $\{\{a\}, \{a, b\}\}$ (to force position-wise equality).

2.7 *Review and Labs*

In this section, we will go through the contents of Chapter 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.

1. How does "recursion work?" In other words, how do people go about writing recursive programs?

   Answer: Recursion relies on explicitly solving one problem and solving the simpler problems by calling the same function as being defined. Suppose someone tells you "the N-th Fibonacci number is the sum of the (N-1)st and (N-2)nd Fibonacci numbers" and wants you to write a function `fib` to calculate the Nth Fibonacci number. Then you must ask *Wait a minute, what if* N=0 *and* N=1*, because for them the above definition is not well defined.* If they say "for N=0, `fib(N)=0` , and for `N=1` , `fib(N)=1`, then you are in business!" You can write the desired function as follows, exactly as told in English, and test it out:

   ```
   def fib(N):
       return 0 if (N==0) else 1 if (N==1) else fib(N-1)+fib(N-2)

   >> [fib(i) for i in range(12)]
   [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
   ```

   In recursive programming, you often express the problem clearly as stated. *Later*, you may optimize for speed. We will show how the above `fib` can be optimized for larger values.

2. How is recursion related to induction?

   Recursion and induction are two sides of the same coin. To see this more clearly, let us take a function that has an *independent* specification. Consider a recursive program that sums numbers from `1` through `N`:

   ```
   def sum(N):
       """ Sum numbers from 1 through N. """
       return 1 if N == 1 else N + sum(N-1)

   >>> sum(10)
   55
   ```

   Now, we know *independently* that `sum(N)` must evaluate to $N(N+1)/2$. How do we prove this? This can be proven by tracing the structure of recursion!

There are two cases in the recursive definition:

- *Basis case of recursion:* `return 1 if N == 1 else ...`
  Corresponding to this, we can introduce a *basis case* for an inductive proof.
  - Does $1(1+1)/2 = 1$? Yes.
- *Inductive case of recursion:* `return ...  else N + sum(N-1)`
  Corresponding to this, we can introduce an *induction step* for an inductive proof.
  - Assume that the `sum(N-1)` part is working correctly, *i.e.*, it returns $(N-1).N/2$.
  - Now prove about the main recursive call that it returns $N(N+1)/2$.
  - But, we can see that the main recursive call returns $N+(N-1).N/2$.
  - This simplifies to $N.(N+1)/2$.

This example illustrates that one can *go by the structure of recursion* in proving properties by induction.

3. What are the main differences between sets and lists? Between strings and lists? Between sets and tuples?

   Sets record occurrence, not the count or order. Lists record both. Strings are sequences of characters while lists are sequences of things (usually of the same type).

   One cannot concatenate or reverse sets; these operations make sense for strings and lists. While concatenation of tuples is not a standard idea in mathematics, it is allowed by Python.

   One can select specific positions of lists and tuples by indexing them starting from 0, as in `((a,b),(c,d))[0][1]`. One cannot index sets.

4. For practice, do these problems: For each expression below that is incorrect in some way, identify why it is so; else, write down the value of the expression:

   (a) `(3)[0]`

   (b) `(3)[1]`

   (c) `((3,(2,4)),(5,6)) [0] [1] [0]`

(d) `[[3,[2,4]],[5,6]] [0] [1] [0]`

(e) `{2,4}+{4,3}`

(f) `[2,4]+[4,3]`

(g) `(2,4)+(4,3)`

5. What is the difference (if any) between a pair and a tuple?

   A pair is a two-tuple. The general word is a tuple. A three-ary tuple is called a triple. The number of positions in a tuple is its *arity*.

6. How do Venn diagrams help understand the relationship between various sets?

   Venn diagrams are mainly able to geometrically depict the overlap between two sets; inclusion (one set including the other); and disjointness (the sets in question having no common point). Venn diagrams can also help us visualize the notion of "$\binom{N}{k}$" as illustrated in Figure 2.6.

7. What are Lambda expressions and how are they used?

   Lambda expressions are basically anonymous ("nameless") functions. Languages like Python allow these functions to be passed around as if they are mere values (such as numbers).

   A simple use of Lambdas is shown below. In this usage, the function takes a single argument `x`. We "bind" 2 to `x` and then return the body of the Lambda (which is what appears after `:`. In this example, we return 2+1 as a result of substituting 2 for `x`.

```
>>> (lambda x: x+1)(2)
3
```

   The following is a usage where the Lambda takes *two* arguments `x` and `y`. Thus, to feed the two arguments, we write `(2,3)`. All the arguments of a function are provided within the parentheses `(` and `)`.

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

   Now suppose you want to write your Lambda to take a *single pair*. Then, the usage is `foo( ...  )` where the `...` is a single pair `(1,2)`. Why? Because we put a "pair template" in the `lambda` that defines `foo`.

```
>>> foo = lambda(x,y): x

>>> foo(1,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: <lambda>() takes exactly 1 argument (2 given)

>>> foo((1,2))
1
```

The simple illustration below shows that Lambdas can be used to define specialized functions. Here, we are stamping out `plus2` to be `f(2)`, which is really the value `lambda y: 2+y`. This is because `f` is a function that takes a single argument `x` and returns `lambda y: x+y` with `x` plugged in.

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

>>> plus2 = f(2)

>>> plus2(10)
12
```

A more elaborate illustration of Lambdas is in the following illustration of how to model `Sigma` and `Pi`. Again notice that `SigmaPi` returns a `lambda`, i.e. a function. Now to use `SigmaPi`, we make first of all an add function `addf`, and then feed it to `SigmaPi` in place of `addOrMult`. The returned result is

```
lambda low, high, step=1:
       reduce(addOrMult, range(low, high+1, step))
```

Let us see all of this below:

```
def SigmaPi(addOrMult):
    return (lambda low, high, step=1:
                reduce(addOrMult, range(low, high+1, step)))

addf = lambda x,y: x+y

mulf = lambda x,y: x*y

Sigma2 = SigmaPi(addf)

Pi2    = SigmaPi(mulf)
```

Notice that `SigmaPi` is a function that takes one input. If `SigmaPi` is applied to `addf`, we get the following lambda as the result:

```
    lambda low, high, step=1:
        reduce(addf, range(low, high+1, step))
```

You can now see that `Sigma2` is bound to the right "add from low to high range" function. Similarly, `Pi2` becomes the standard "series multiplication" function.

Finally, to use Sigma2, we just do this:

```
>>> Sigma2(1,10)
55
```

8. How do you reverse a string in Python? Provide an example.

   Python allows strings to be stepped through from specific positions in specific increments. Using the same method, we can allow the string to be stepped through in increments of -1. This allows Python to reverse

```
>>> 'abcdefgh'[::2]
'aceg'

>>> 'abcdefgh'[2:7:2]
'ceg'

>>> 'abcdef'[::-1]
'fedcba'

>>> [1,2,3,4][::-1]  # works for lists also
[4, 3, 2, 1]
```

9. Show how to reverse the concatenation of two strings by reversing its parts.

```
>>> ('abc' + 'def')[::-1] == ('def'[::-1] + 'abc'[::-1])
True
```

   Both the left-hand and right-hand sides return `'fedcba'` and hence the equality is true.

10. How do the <= operation differ for sets and lists? Can we use any of the set operation on lists also, with the same meaning?

    Answer: In general, it is not wise to treat lists as sets. The only exception is when sets of sets are to be represented (this is not supported by Python); in that case, one can use list of lists.

    Lists do behave like sets as far as the use of `in` goes. Otherwise, convert a list to a set before you attempt operations that only make sense for sets.

    So notice that it makes sense to ask whether a set is a subset of another set. `{1,2,3} <= {3,4,5}`.

    Whereas for lists, the notation <= is a *less than* relation between lists, as defined in [11]. It has nothing to do with subsets.

11. A note on parsing

    (a) What does `{1,2,5} & {3,4,2} | {5}` return?

        Answer: Notice that & binds tighter, and so the answer is `{2,5}`

    (b) What does `{1,2,5} & {3,4,2} < {5} | {2}` return?

        It returns `True`.

        Notice that it makes no sense to perform the evaluation of

        `{1,2,5} & {3,4,2} < {5}`

        as

        `(({1,2,5} & {3,4,2}) < {5})`

        and then perform a | ("or") with `{2}`. This is because the first part (namely, `{1,2,5} & {3,4,2} < {5}`) gives us a Boolean, and we can't | ("or" or "union") it with a set.

    (c) The answer is False because `{2,5}` is not a proper subset of `{2,5}`.

    (d) The example above resembles `4+5 <= 4+6` where we evaluate the addition before comparing according to <=.

12. Practice solving the following queries on sets. In case there are inclusions, point out where *proper* inclusions (*e.g.*, proper subset and proper superset) are involved.

    (a) `{1,2} ^ {3,4}  <  {1,2} | {3,4}`

        In math, the same query is: $\{1,2\} \triangle \{3,4\} \subset \{1,2\} \cup \{3,4\}$

(b) `{1,2} ^ {3,4}  <  {1,2} & {3,4}`

In math, the same query is: $\{1,2\} \triangle \{3,4\} \subset \{1,2\} \cap \{3,4\}$

(c) `{1,2} ^ {3,4}  >  {1,2} & {3,4}`

In math, the same query is: $\{1,2\} \triangle \{3,4\} \supset \{1,2\} \cap \{3,4\}$

13. What will this query return in Python?

```
map(lambda x: x+1 if (x%2 == 0) else x-1,  range(10))
```

The answer is `[1, 0, 3, 2, 5, 4, 7, 6, 9, 8]`.

14. What will this query return in Python?

```
map(lambda x: range(x),  range(10))
```

The answer is as shown below. The last list is `[0, 1, 2, 3, 4, 5, 6, 7, 8]` because `range(10)` produces `[0..9]` and when we map the Lambda over this, we will ask it to produce `range(9)` at the very end.

```
[[], [0], [0, 1], [0, 1, 2], [0, 1, 2, 3], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4, 5],
 [0, 1, 2, 3, 4, 5, 6], [0, 1, 2, 3, 4, 5, 6, 7], [0, 1, 2, 3, 4, 5, 6, 7, 8]]
```

15. How do `reduce` and `map` work? `map` applies a given function over *all* the elements of a list or string. Reduce takes a function of two arguments (a *two-ary function*) and a list, and builds a *reduction tree*, composing all the elements of the list.
Here are three examples (and notice that with triple-quoting, we can embed newlines within strings):

```
>>> sentence = """a, b, c.
... d e .. ,,,foo? no it, is, bar!"""

>>> map(lambda x: 1 if x==’,’ else 0, sentence)
[0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0]
```

Notice how each comma turns into a 1 and others into a 0.

```
>>> map(lambda x: ’,’ if x==’,’ else ’’, sentence)
[’’, ’,’, ’’, ’’, ’,’, ’’, ’’, ’’, ’’, ’’, ’’, ’’, ’’, ’’, ’’, ’’, ’’,
 ’’, ’’, ’’, ’,’, ’,’, ’,’, ’’, ’’, ’’, ’’, ’’, ’’, ’’, ’’, ’’, ’’, ’,’,
 ’’, ’’, ’’, ’,’, ’’, ’’, ’’, ’’, ’’]
```

Notice how each comma turns into a ',' and others into a " (empty string)!.

Now let us define a suitable function for use in reduction. Call it `redFn`.

```
redFn = lambda x,y: x+y
```

One nice thing in Python is *overloading*! For instance, `1+2 = 3` while `"hello " + "there!" = "hello there!"`. We will exploit it in the illustrations below:

```
reduce(redFn, map(lambda x: 1 if x==',' else 0, sentence))
7
```

In this illustration, `redFn` added all the elements of this list, giving `7`:

```
[0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0]
```

Now see what happens with `redFn` applied to strings:

```
>>> reduce(redFn, map(lambda x: ',' if x==',' else '', sentence))
reduce(redFn, map(lambda x: ',' if x==',' else '', sentence))
',,,,,,,'
```

It basically concatenated all the `,` and `''` into a string of seven `,` !

16. How many vowels were there in the Gettysburg address? Answer: 449, as obtained below.

```
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))
```

17. How do we prove by induction? Illustrate on balanced binary trees!
    Answer: Imagine a balanced binary tree. A tree of height 0 has 1 leaf and 1 total node. A tree of height 1 has 2 leaves and 3 nodes. A tree of height 2 has 4 leaves and 7 nodes.

    (draw these please!)

    Induction Hypothesis: Suppose true for trees of height upto and including N : that they have 2^N leaves and $2^{N+1} - 1$ nodes.

    Induction Step: Show that trees of height $N + 1$ have 2^{N+1} leaves and $2^{N+2} - 1$ nodes.

    Observation: We grow the tree by a height of one by adding *two* new leaves per original leaf. This immediately gives 2^{N+1} leaves. The total nodes are $2^{N+1} - 1 \; + \; 2.2^N \; = \; 2^{N+2} - 1$. Hence proved!

18. Show that $\binom{N}{k} = \binom{N-1}{k} \; + \; \binom{N-1}{k-1}$.

    Answer: Suppose instead we are faced with playing with N electric switches numbered 1 through N. We have to turn k of these N switches on (say, to launch k "lucky" people from k circus cannons with a boom, onto a waiting net!) There are two cases, directly giving the formula above!

    (a) We spare the first person. Then we have to launch from the $N - 1$ remaining folks the k "lucky" guys.

    (b) We choose the first person as one of the lucky guys (he assures he will fit into the cannon). Then we need to choose only $k - 1$ lucky individuals from the $N - 1$ lucky individuals.

19. How does the above formula turn directly into a program? Show this program with all bells and whistles (design it as a script, and invoke the pretty printer. Also handle the boundary cases properly.

```
#! /usr/bin/env python

import pprint
pp = pprint.PrettyPrinter(indent=4)

def chooseRec(N=5, k=2):
    """Recursive formulation of choose using
```

```
       N choose k = N-1 choose k + N-1 choose k-1."""
    if (N < 0 | k < 0):
        print "ERROR: N and k must be >= 0"
        return -1
    elif (k > N):
        print "ERROR: N must be >= k"
        return -1
    elif ((N == k) | (k == 0)):
        return 1  # these are trivial choices!
    else:
        return chooseRec(N-1, k) + chooseRec(N-1, k-1)

def doubChooseRec(N, k):
    """ This illustrates that chooseRec can also be called from 'inside'"""
    return 2 * chooseRec(N, k)

if __name__ == '__main__':
    N, k = input("Please provide N, k: ")
    print "chooseRec(" + str(N) + ", " + str(k) + ")= " + str(chooseRec(N, k))
    pp.pprint("... and now to call it directly after doubling")
    print "doubChooseRec returns ", doubChooseRec(N, k)

# Now to test it, generate one of the Pascal's triangle rows!

linux prompt> chmod a+x chooseRec.py

linux prompt> chooseRec.py

Please provide N, k: 5,2
5,2
chooseRec(5, 2)= 10
'... and now to call it directly after doubling'
doubChooseRec returns  20

>>> from chooseRec import *

>>> # This is an extremely neat Pascal's Triangle program !!
>>> # These are also Binomial coefficients. Please read on Wikipedia (e.g.)

>>> for N in range(8):
      print [chooseRec(N,k) for k in range(N+1)]
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
[1, 6, 15, 20, 15, 6, 1]
[1, 7, 21, 35, 35, 21, 7, 1]
```

Chapter 3

Boolean Basics and Propositional Potpourri

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. As we have done before, we will start with a historical perspective of the notions to be introduced in this chapter. We will reap the reward of having learnt Python programming—especially Lambdas.

We have already employed truth values in our programs so far. Whenever we write conditional expressions of the kind

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

we are dealing with Boolean functions. It is important for the programmer to understand when the "`then`" part will be evaluated — and more importantly when the "`else`" part will be evaluated, because the "`else`" condition is the negation of the "`then`" condition. It turns out that the "`else`" part will be executed when the following *conditional expression* is true:

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

The questions are: *Is this what the programmer intended? How can they be sure?* Convoluted conditional expressions result in programs that are difficult to debug. In this chapter, we will study the theory of Boolean functions that is a prerequisite to handling such questions. In particular, in our present example, *De Morgan's Laws* of §3.5 help us obtain the `else` condition from the `then` condition.

To gain further insights about Boolean functions, we are going to learn how truth values can be represented in hardware design. For this purpose, we will employ a digital circuit simulator called `TkGate`. Building simple circuits and experimenting with an actual simulator will prove to be a bedrock of understanding to later build upon. We will then abstract the behavior of Boolean gates into tabular presentations called *Truth Tables*. We will also show you how to use hash tables for representing Boolean functions.

We will discuss functions and Boolean functions. To document what functions accept as well as produce, we will use the notation of *signatures*. We will discuss the curried style of writing functions and also elaborate on the signature of curried functions.

We will count the number of possible truth-tables over N inputs and be amazed at how fast this number grows! Counting will be done in two ways: (1) counting truth-tables, and (2) counting the number of ways to "program" a multiplexor-based universal gate. In the process, we will gain further understanding of Boolean functions by designing Boolean functions using `Lambdas`. We will show that gate composition and Lambda composition are very similar activities!

3.1 Chapter Material: History/Applications

3.1.1 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 of the University of Cork, Ireland, published a book called *The Laws of Thought.*[1] 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

[1]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."

(shorting relays causing copious sparks). Again, despite the brilliance[2] of Shannon's work, the world largely yawned, content with its *ad hoc* ways of designing relays[3] Fortunately, the design of digital computers in the 1940s sparked (in a good way) interest in Boole's calculus, and the rest is history.

3.1.2 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.[4] 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.

3.1.3 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[5]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.[6] 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*.[7]

[2]pun intended...

[3]...and staying sufficiently far from the relays.

[4]To gain better appreciation for the scale of this number, a human lives around 3 billion seconds.

[5]Did the term pupil come from the fact that it is they who let the teacher clearly see?

[6]Microprocessors do have published errata ranging over several pages; but we have learned how to avoid fatal mistakes.

[7]A company in Austin called Centaur Inc. builds x86 processors where the register-

- 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`:

```
4195835.0 - 3145727.0*(4195835.0/3145727.0) = 0    (Correct value)
4195835.0 - 3145727.0*(4195835.0/3145727.0) = 256  (Flawed Pentium)
```

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![8] Using 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.

3.2 Truth Values and Functions

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 3.1 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`.[9]

transfer logic of the entire microprocessor is represented in a language called `ACL2`, and daily verification regressions are run on the whole design.

[8]This calculation was performed using Wolfram Alpha available at `http://www.wolframalpha.com`.

[9]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

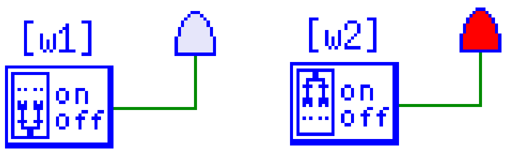


Figure 3.1: A switch and LED represented in TkGate

Be at a computer and start up `TkGate`, version 2.0-b10 (beta). The welcome screen actually contains several circuits, and is a tutorial! You can immediately do these tasks:

- Click on the Simulate tab
- Click on ▷ (to start the simulation)
- "Play" with the switches to move them from off to on, and vice versa
- You will see the lights changing!

Now, using the `File` menu, create a file with some appropriate name (say, `switches.v` and keep the top-level module of the design you are about to create as `main`. In the canvas (drawing space), type '`s`' to obtain a switch, and '`l`' to obtain an LED; see Figure 3.1. Connect the wires by clicking the left mouse button (a soldering iron pops up, and you can finish the wiring by dragging and mating the wires). Make sure that the whole wire appears slightly thicker (sign of good connection). Select the whole circuit using the mouse, and make a copy of this circuit through copy/paste. Now hit `Simulate`, and now you can simulate the circuit. The simulation caterpillar keeps walking in the top right window so long as the simulation is running and the combinational stages have not stabilized. In case of this simple circuit, each time you move around the switch by clicking it with the mouse, the circuit stabilizes in a short while, and the LED state changes.

3.2.1 Functions, and Boolean Functions

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

fairly current.

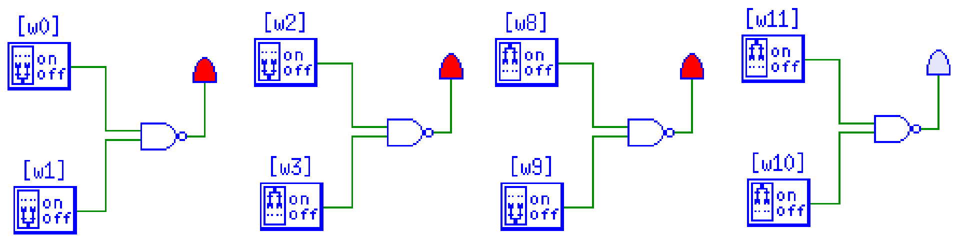


Figure 3.2: The Nand function illustrated over 00, 01, 10, and 11

x	y	z
0	0	1
0	1	1
1	0	1
1	1	0

```
def nand(x,y):
    return 0 if (x & y) else 1

>> nand(0,0)
1
>> nand(1,1)
0
>> nand(0,1)
1
>> nand(1,0)
1
```

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

>> nand(0,0)
1
>> nand(1,1)
0
>> nand(0,1)
1
>> nand(1,0)
1
```

Figure 3.3: The Nand Truth-table, Lambda Representation, and Hash-table Representation

Figures 3.2, 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 produce 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 3.3. 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.

3.2.2 Mathematics of Functions, Signature

We need some concepts to talk about functions more abstractly. The first of these concepts is called the *cartesian product*. The second concept is that of *function signatures*. Let us see these concepts now.

The notion of 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([])
```

It is immediately clear that Python supports this *set comprehension* style definition of cartesian product. Notice that 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).
Terminology: In a set comprehension expression of the form `(x,y) for x in 1,2,3 for y in "he", "she"` call `(x,y)` a template expression, and the part `for x in 1,2,3 for y in "he", "she"` the quantifier expression. Armed with this notation, we can now define the notion of function *signatures*.

Functions and Signature

A function is a mathematical object that expresses how items called "inputs" can be turned into other items called "outputs." 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 "$f : T_D \mapsto T_R$" 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

$$Nand : Bool \times Bool \mapsto Bool$$

if we assume that $Bool = \{0, 1\}$. (Sometimes we employ `True`/`False` in lieu of 1/0.) The signature of $Nand$ is evident from Figure 3.3 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, $+ : Nat \times Nat \mapsto Nat$ denotes the signature of natural number ($\{0, 1, \ldots\}$) addition.

There is one more detail about signatures that we need to understand when functions return functions. This is our next topic.

Signatures When Functions Return Functions

Suppose someone decides to write `nand` differently as follows:

```
def nand(x):
 return lambda y: 0 if (x & y) else 1

>>> nand(0)
<function <lambda> at 0x100499c80>

>>> nand(1)(1)
0

>>> nand(1)(0)
```

```
1

>>> nandOf1 = nand(1)
>>> nandOf1(1)
0
```

For this Nand function, we write the signature as

$$Bool \mapsto (Bool \mapsto Bool)$$

In other words, this signature says:

- "Gimme a Boolean"
- "I'll return a function of type $Bool \mapsto Bool$"

It is evident that we can bind this function to a variable (`nandOf1` in our example) and later feed it one more Boolean that it is expecting. Or, we can feed the arguments one after the other, as the syntax `nand(1)(1)` suggests (we avoid binding the function returned after feeding the first argument to a variable; we just continue to feed another value that is being expected).

Currying: Functions with signatures of the form

$$Bool \mapsto (Bool \mapsto Bool)$$

are called *Curried* functions.[10].

Curried functions have the benefit that one can feed them values one at a time, and store away the intermediate functions. Non-curried functions on the other hand expect to be fed all values in one fell swoop.

An Example: Signature of `SigmaPi` of §2.7: Now, we can refer back to §2.7 and make more sense of the function

```
def SigmaPi(addOrMult):
    return (lambda low, high, step=1:
                reduce(addOrMult, range(low, high+1, step)))
```

[10]Please don't imagine curried chicken and start salivating; it's a different curry here! The word "Curry" stands for the famous logician Haskell B. Curry. Apparently it was his student Moses Schönfinkel who came up with this idea. As was traditional, students never used to give themselves credit; they used to deflect the credit to their advisor. It is however also rumored (not sure if entirely fabricated) that when people started calling this activity "Schönfinkeling," one wasn't sure whether one was supposed to do that.

- Its signature is

$$(Int \times Int \mapsto Int) \mapsto ((Int \times Int \times Int) \mapsto Int)$$

- The reason for this complicated signature is as follows:
  - It expects to be given a function called `addOrMult`. This function has signature $Int \times Int \mapsto Int$
  - When given such a function, it produces a function that has signature $Int \times Int \times Int \mapsto Int$
  - It is this function that is then fed `low`, `high`, and `step` (the three `Ints` in the above signature) to finally obtain a single `Int` as the final answer.

Having studied cartesian products and function signatures, we are back to studying Nand, as well as more functions similar to it.

3.2.3 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

$$Bool \mapsto Bool$$

Let us try and crank out *all* of those functions that take two inputs. Figure 3.4 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.

3.2.4 How Many Functions over Two Inputs Exist?

It is clear that $Nand$ is only *one of the possible functions* of signature $Bool \times Bool \mapsto Bool$. Let us try and crank out *all* of those functions that take two inputs. Figure 3.5 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.

$Constant$ 0

x	z
0	0
1	0

$Constant$ 1

x	z
0	1
1	1

$z = !x$
Inverter

x	z
0	1
1	0

$z = x$
Identity

x	z
0	0
1	1

Figure 3.4: All possible 1-input Boolean Functions

$Constant$
0

x	y	z
0	0	0
0	1	0
1	0	0
1	1	0

$z = x.y$
AND

x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

$z = x.\,!y$

x	y	z
0	0	0
0	1	0
1	0	1
1	1	0

$z = x$

x	y	z
0	0	0
0	1	0
1	0	1
1	1	1

$z = !x\ .y$

x	y	z
0	0	0
0	1	1
1	0	0
1	1	0

$z = y$

x	y	z
0	0	0
0	1	1
1	0	0
1	1	1

$z = !x.y + x.!y$
XOR

x	y	z
0	0	0
0	1	1
1	0	1
1	1	0

$z = x + y$
OR

x	y	z
0	0	0
0	1	1
1	0	1
1	1	1

z = !(x+y)
NOR

x	y	z
0	0	1
0	1	0
1	0	0
1	1	0

$z = xy{+}!x.!y$
XNOR or =

x	y	z
0	0	1
0	1	0
1	0	0
1	1	1

$z = !y$

x	y	z
0	0	1
0	1	0
1	0	1
1	1	0

$z = x{+}!y$

x	y	z
0	0	1
0	1	0
1	0	1
1	1	1

$z = !x$

x	y	z
0	0	1
0	1	1
1	0	0
1	1	0

$z = !x + y$
IMPLICATION

x	y	z
0	0	1
0	1	1
1	0	0
1	1	1

$z = !(x.y)$
NAND

x	y	z
0	0	1
0	1	1
1	0	1
1	1	0

$Constant$
1

x	y	z
0	0	1
0	1	1
1	0	1
1	1	0

Figure 3.5: All possible 2-input Boolean Functions

3.2.5 Counting All Possible N-input Truth Tables

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`):

- 16 – 2-input gate types (of the kind shown in Figure 3.5) are possible.
- 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!)
- $1.8.10^{19}$ 6-input gate types
- 3.10^{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 Functions: The point is this: no purveyor of possible gate types ought to have dreams of supplying one custom-made gate for *every* possible function type! The way around this conundrum is simple: provide *one* gate type (or a few gate types) using which we can make all other gate types. These are termed *universal gate sets*.

- Here are some universal gate sets (we have *not* included all possibilities): $\{Nand\}$, $\{Nor\}$, $\{And, Not\}$, $\{Or, Not\}$, $\{Implication\}$, and $\{And, XOR\}$.
- This means that using merely Nand gates, Nor gates, Implication gates, one can make every other gate type.

- *And* by itself is not universal; similarly, *Or* by itself is not universal. Even $\{And, Or\}$ is not universal. Also, XOR by itself is not universal.
- However, notice that $\{And, Not\}$ is universal, and so is $\{And, XOR\}$.

In §3.6, we will show you how to prove that a set of gate types is (or isn't) universal.

3.2.6 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 3.6 summarizes our common usages, and also shows commonly used gate icons.

3.3 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 §3.3.1, an Implication gate can be realized using Nand §3.3.2, and a 2-to-1 mux can also be realized using Nand §3.3.3.

3.3.1 Realizing Inclusive and Exclusive Or using Nands

An *inclusive* Or (or simply "or") has the truth table discussed in Figure 3.5. The Or function can be realized using the Nand function as follows:

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

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

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

We leave it to the reader to draw and simulate this circuit using TkGate. Figure 3.13 shows how to realize the Exclusive OR functionality using Nands.

3.3.2 An "Implication Gate"

The Implication function has truth-table described in Figure 3.5. It can be realized using Nands as follows:

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

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

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	$!x$, $\neg y$
Function	"Implication"	$\Rightarrow$	`If-Then`		$x \Rightarrow y$, `if x then y`
Function	"XOR"	$\oplus$	$\neq$	"Inequality"	$x \oplus y$, $x \neq y$
Function	"XNOR"	$\overline{\oplus}$	=	"Equality"	$x \overline{\oplus} y$, $x = y$, $x \Leftrightarrow y$

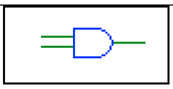
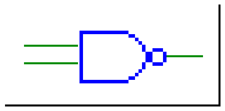
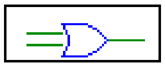
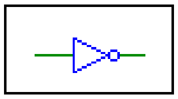
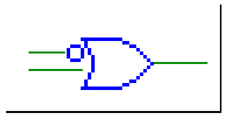
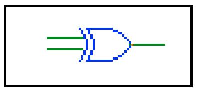
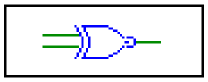
Function	Gate Icon	inputs
"And"		on the left
"Nand"		on the left
"Or"		on the left
"Not"		on the left
"Implication"		`i` on left `s` is beneath
"XOR"		on the left
"XNOR"		on the left

Figure 3.6: Different Syntaxes as well as Gate Icons for Boolean Functions

3.3.3 A 2-to-1 Mux Using Nand Gates

A 2-to-1 multiplexor is shown in Figure 3.8. This multiplexor can be realized as follows:

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

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

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

That is, "IF s THEN b ELSE a." This is why in §3.4.1, we call the gate an `if-then-else` gate.

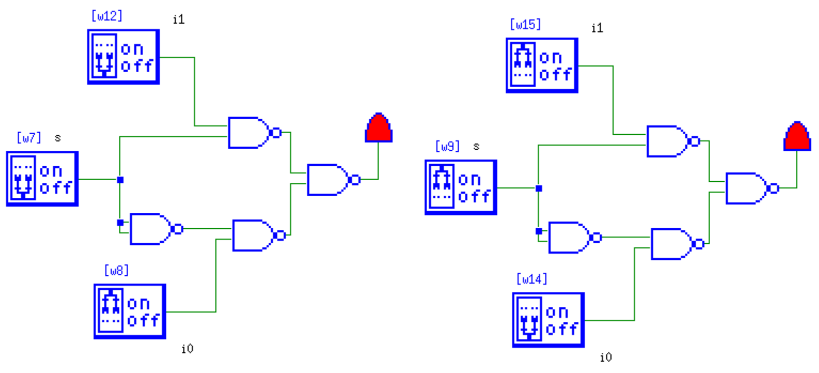


Figure 3.7: A 2-to-1 multiplexor Realized Using Nands

3.4 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 3.5, 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.

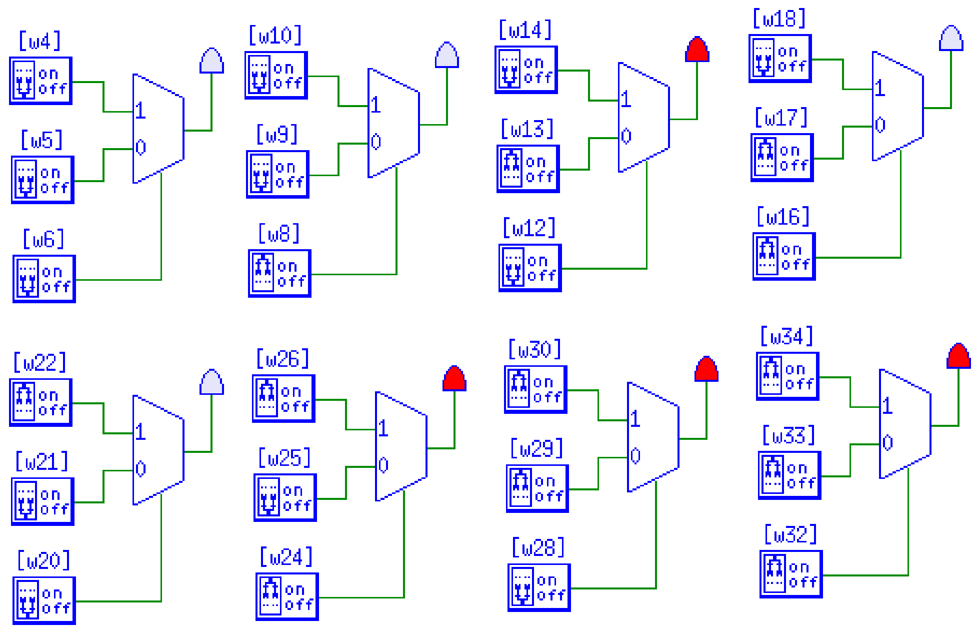


Figure 3.8: A 2-to-1 multiplexor

3.4.1 A 2-to-1 Multiplexor or an `if-then-else` gate

A 2-to-1 mux is illustrated in Figure 3.8. If we take the inputs of this circuit to be $i1$ and $i0$ (for "inputs") and s (for "select"), the functionality of this circuit is described by the following lambda expression:

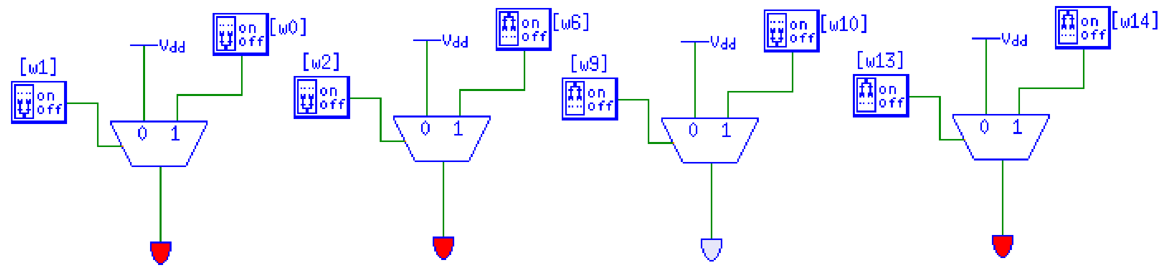
```
>>> mux21 = lambda(i1,i0): lambda(s): i0 if not(s) else i1

>>> 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 3.8 is being captured by the above lambda expression. It is basically an `if-then-else` (or `if-else`) gate: it copies `i0` to the output if `not(s)`; else, it copies `i1` to the output. Look at Figure 3.8 again: the `i0` input is labeled `0` and the `i1` 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 `i0` 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 `i0` is high, and hence the LED on the output port is lit.

It is also clear that a `mux21` is one of the 2^{2^3} possible three-input Boolean functions. This is because `mux21` is sensitive to all of its three inputs, and maps the inputs as described by Figure 3.8. Let us now see how an `if-then` gate can be obtained from `mux21`.

An `if-then` Gate



```
>>> Implication = lambda i1: lambda s: i1 if s else true
```

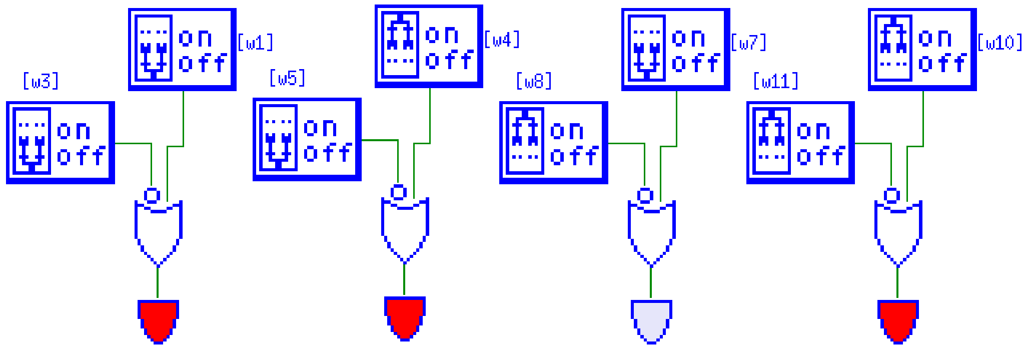


Figure 3.9: An Implication Gate: $s \Rightarrow i1$, shown using two different icons. The "select" input `s` is to the left-hand side, while the input `i1` is on top.

The Implication gate in Figure 3.5 can be called an `if-then` gate. One can use the following lambda to describe it, and realize it as shown in Figure 3.9. The symbol $\Rightarrow$ will be used to denote implication. Thus, one can express the functionality of this gate also as $s \Rightarrow i1$. 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: $(1 = 1) \Rightarrow (2 = 2)$

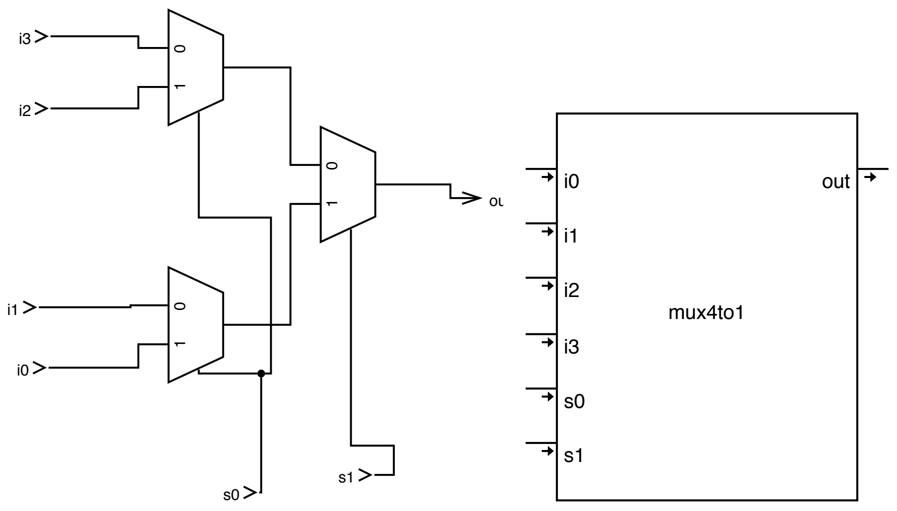
It is clear that S0, S1, and S3 (likewise T0, T1, and T3) must be true, and both S2 and T2 are false. This is because

- 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. As will be further elaborated in §3.7, it would be catastrophic to conclude "false" as a theorem because it would then allow us to prove *anything* at all.

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 $!P$—something no logical system wants to admit.

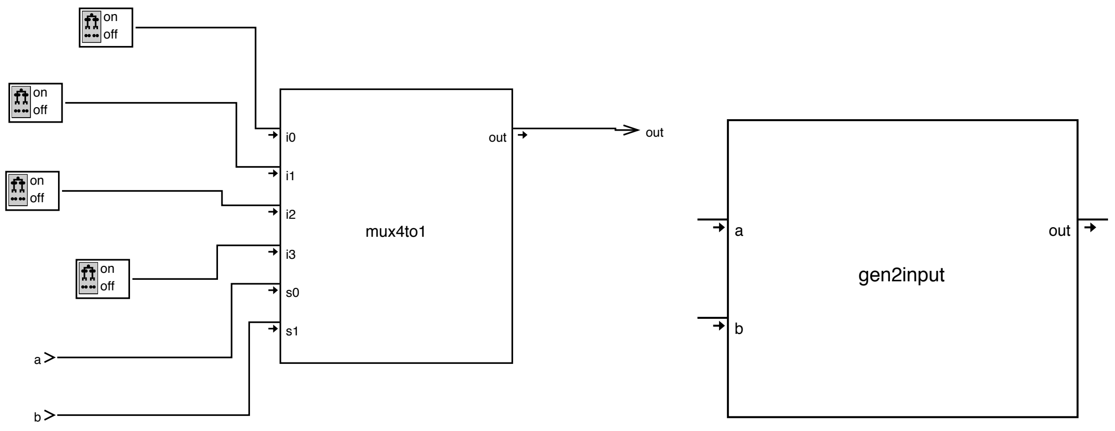
3.4.2 Creation of a general 2-input function module

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-**



```
>>> 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 3.10: A 4-to-1 mux: Before and After Interface Generation. You can verify that depending on the input bit combination at inputs s0 and s1, one of the four inputs (i0 through i3) is copied to the output (out). We also provide ways to define these functions using Lambdas



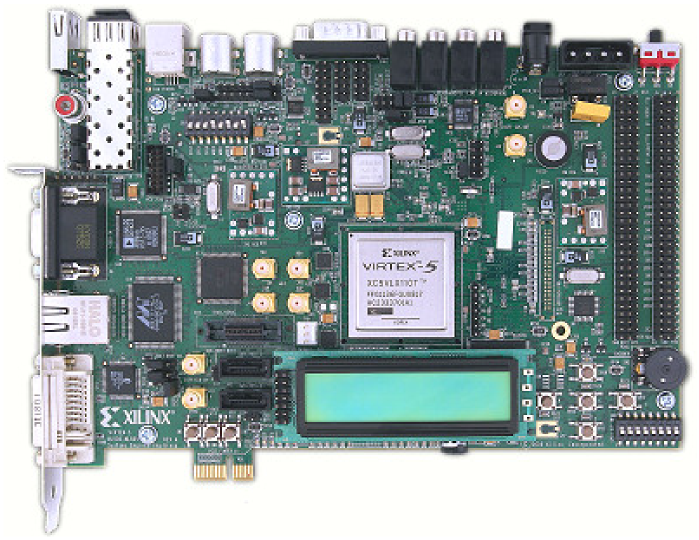


Figure 3.11: A General 2-input Function Module and its Symbol (Interface). This is a "general" builder because by setting the switches, we can realize *any* of the 16 functions in Figure 3.5 by downloading a bit-pattern into their storage units that then set the "input switches." A prototyping board 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.

to-1 multiplexors.

Before we do that, we must get better at using TkGate. In practice, when using TkGate, cutting and pasting circuits *ad nauseam* is not a good way to build hierarchical circuits. We will show you how to build hierarchical modules and thus realize a 4-to-1 multiplexor that is "programmed" to realize an XOR gate. In other words, let us build an XOR gate using three 2-to-1 multiplexors and a few switches.

Open a new file gen2input.v. In here, we shall create two modules as described now. The first is called mux4to1 (the creation of which is described in detail) and the second is called gen2input (the creation of which is described briefly).

Creating a 4-to-1 mux

Let us follow along the construction shown in Figure 3.10. Let us attempt to create a module called mux4to1 as shown on the right-hand side of this figure.

Create a new module mux4to1 by selecting Module and New. Uncheck the button Prevent edit mode interface changes. This will allow us to resize the module. The module mux4to1 is created under Unused.

Now double-click on mux4to1 and this puts you in a position to create the circuit that makes up this module. Proceed to create this circuit as follows.

- Enter three 2-to-1 multiplexors by clicking their positions and typing m.
- Rotate the muxes to face right.
- Wire-up the muxes to form a 4-to-1 mux.
- Place ports to export the relevant signals to the interface. The ports might be called i3, i2, i1, and i0. The creation of these ports is accomplished by clicking the left mouse, and then typing Shift-}, and typing the port name into the pop-up.
- Now type more input ports, namely s0 and s1, and one output port, namely out.
- Now hit save. The result is as in Figure 3.10.
- Now you can generate the interface properties of this module by selecting Interface and selecting Auto Generate. Select Use ports used in module only. Presto! Module mux4to1 would now be created with the ports introduced above forming part of the interface.

Testing the Lambda Expressions of the Muxes

The encoding of the functions defined in Figure 3.10 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 3.11, arriving its module definition in a new module called `gen2input`.

Editing `gen2input` to become an `XOR`

Notice that we left the "switches" inside Figure 3.11(left) at "all off".[11] 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. Editing the switches is accomplished by double-clicking on the switches Then auto-generate its interface also. You can now see two simulation states of this component. Figure 3.12 shows all the pertinent details. We can also simulate these XOR functions similar to how we tested the muxes in Figure 3.10. Here are some details:

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

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

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

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

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

[11]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".

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

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

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

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

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

3.4.3 Counting Boolean Functions: the Mux Approach

Figure 3.12 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 3.11 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 digital system. In our research group, we have realized an entire multiprocessor of 8 cores that fits comfortably into such a board.

3.5 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 "=":

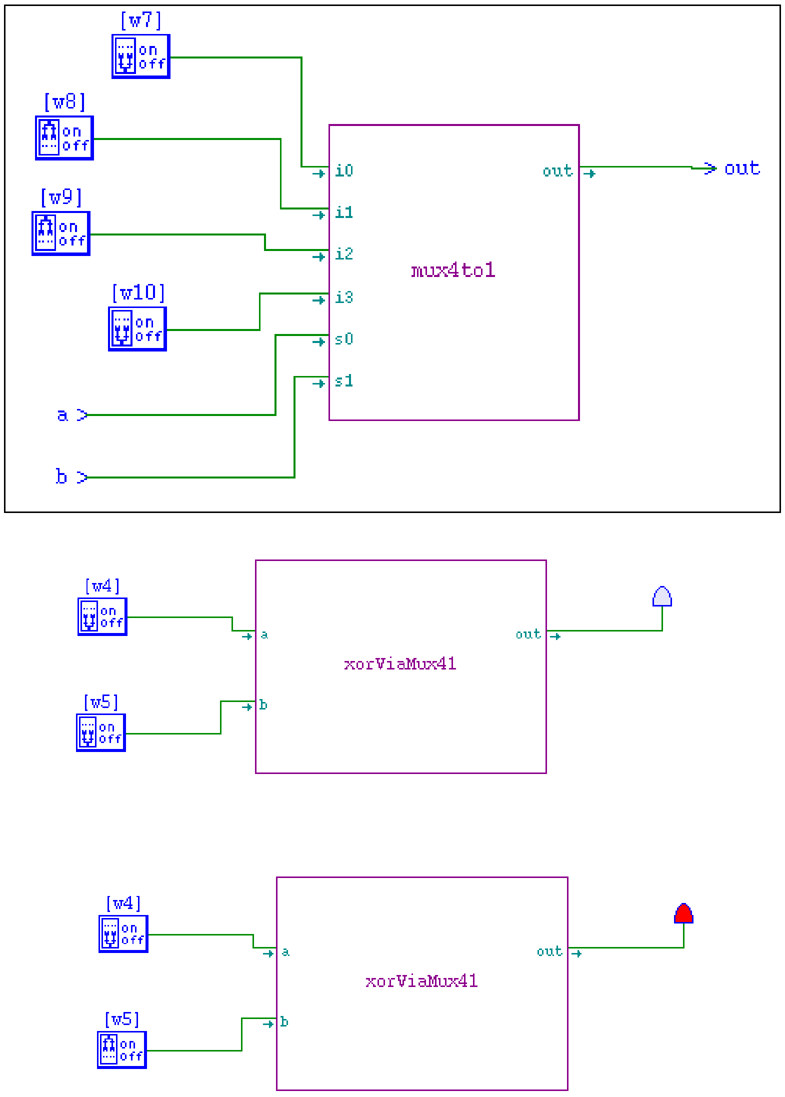


Figure 3.12: An XOR realization (top) and its simulation states (middle, bottom)

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

3.6 Proof of Universality of Gates

In this section we prove a powerful result in the space of 2-input Boolean functions. As described in §3.2.5, a set of gates $G = \{g_1, \ldots, g_N\}$ is a universal set *if and only if* it can realize *any* Boolean function. In this book, we are not going to delve too deeply into building Boolean function networks. However, to illustrate the fact that Nand is universal, in Figure 3.13, we show how to build an XOR gate using four Nands: The fact this is XOR can be

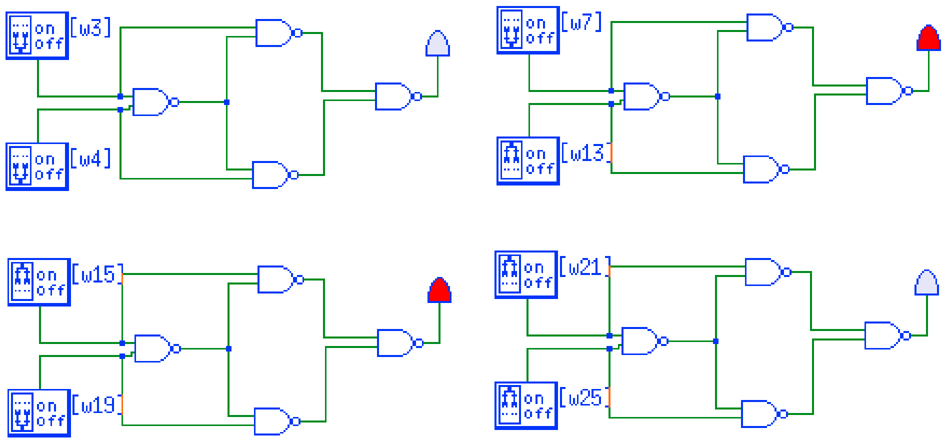


Figure 3.13: XOR via Nand

derived using Boolean identities in §3.5:

- Let $P = !(x.(!(x.y)))$
- Let $Q = !(y.(!(x.y)))$
- $Out = !(P.Q) = !P + !Q$
- $!P = (x.(!(x.y)))$
- $!Q = (y.(!(x.y)))$
- $!P + !Q = (!(x.y)).(x + y)$
- $= (!x + !y).(x + y)$
- $= !x.(x + y) + !y.(x + y)$
- $= (!x.x) + (!x.y) + (!y.x) + (!y.y)$
- $= (!x.y) + (!y.x)$

3.6.1 Universality Condition

Universality Condition: A set of gates G is universal if and only if it can be used to realize a 2-to-1 mux.
Proof of the Necessary part: If a set of gates G is unable to realize a 2-to-1 mux, well then it is not universal. There is at least one well-defined Boolean function that it cannot generate.
Proof of the Sufficient part: Suppose G can realize a 2-to-1 mux. Then it is clear that we can build a N-to-1 mux for any N by connecting these muxes in a tree-like fashion, as in Figure 3.10.

3.6.2 Showing that XOR is *Not* Universal

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

- We can take an XOR gate and feed it 1, 0, x, or y. The resulting set of functions are
  - $x \oplus y$
  - $x \oplus 1 = !x$
  - $x \oplus 0 = x$
  - $y \oplus 1 = !y$
  - $y \oplus 0 = y$
- Adding one more XOR gate in all possible ways, we can obtain
  - $(x \oplus y) \oplus x = y$
  - $(x \oplus y) \oplus y = x$
  - $!x \oplus x = 1$
  - $!y \oplus y = 1$
  - $(x \oplus y) \oplus !y = !x$
  - $(x \oplus y) \oplus !x = !y$
  - $(!x \oplus y)$
  - $(!y \oplus x)$
- We find that we never exit the set; we *saturate* (or close off) into this set.
$$XOR_{realized} = \{(x \oplus y), (!x \oplus y), (x \oplus !y), !x, !y, 0, 1\}$$
- Since none of these are a 2-to-1 mux, XOR is not universal.

This is an inductive-style proof because we assume by induction hypothesis that the set of XOR-realized functions is $XOR_{realized}$. The induction step shows that we never leave this set.

3.7 Propositional Reasoning

Propositional logic or Propositional calculus is a formal system for reasoning. The idea behind propositional calculus is to set up

- *axioms* that are true, and
- *rules of inference* that are truth-preserving

The idea in propositional reasoning is to deduce, purely through syntactic manipulations, new Boolean facts from existing facts. In the next section, we give a taste of propositional logic.

3.7.1 Axiomatization of Propositional Logic

Following Church [2], we now present one axiomatization of propositional calculus following the so called "Hilbert style." Let the syntax of propositional formulas (wff) be as follows:

$$Fmla ::= Pvar \mid \neg Fmla \mid Fmla \Rightarrow Fmla \mid (Fmla).$$

In other words, a propositional formula is a propositional variable (a *Boolean* variable), the negation of a propositional formula, an implication formula, or a parenthesized formula. In relationship with the earlier sections on Boolean reasoning, we have trimmed down the set of "propositions" and formulae we deal with. You may wonder what happened to $\wedge$, $\vee$, $True$, and $False$; well, they are all obtainable from the given set of "formula builders." Here are some examples:

- Take $True$ to be $x \Rightarrow x$ for some propositional variable (Boolean variable). This works because no matter what x is, the formula $x \Rightarrow x$ is $True$. Thus, one can use $x \Rightarrow x$ in place of $True$.
- By a similar reasoning, one can use $x \Rightarrow !x$ in place of $False$.
- Instead of $x \vee y$, one can use $!x \Rightarrow y$.
- Instead of $x \wedge y$, one can use $!(x \Rightarrow !y)$. From §3.5, using the expansion for $\Rightarrow$, this can be rewritten as $!(!x \vee !y)$. Then using De Morgan's law, we obtain $(x \wedge y)$.

Now we present three axioms (A1-A3) and two rules of inference (R1-R2) shown below. Here, axioms are taken to be *theorems* – they are $True$ for all values to their variables. Here, p and q stand for propositional formulas.

A1: $p \Rightarrow (q \Rightarrow p)$

A2: $(s \Rightarrow (p \Rightarrow q)) \quad \Rightarrow \quad (s \Rightarrow p) \Rightarrow (s \Rightarrow q)$

A3: $(\neg q \Rightarrow \neg p) \quad \Rightarrow \quad (p \Rightarrow q)$

R1 (Modus Ponens): If P is a theorem and $P \Rightarrow Q$ is a theorem, conclude that Q is a theorem.

Example of Modus Ponens: From p and $p \Rightarrow (p \Rightarrow q)$, we can infer $p \Rightarrow q$.

R2 (Substitution): The substitution of wffs for propositional variables in a theorem results in a theorem. A substitution is a "parallel assignment" in the sense that the newly introduced formulas themselves are not affected by the substitution (as would happen if, for instance, the substitutions are made serially).

Example of substitution: Substituting $(p \Rightarrow q)$ for p and $(r \Rightarrow p)$ for q in formula $p \Rightarrow q$, results in $(p \Rightarrow q) \Rightarrow (r \Rightarrow p)$. It is as if p and q are replaced by fresh and distinct variables first, which, in turn, are replaced by $(p \Rightarrow q)$ and $(r \Rightarrow p)$ respectively.

We do *not* perform the substitution of $r \Rightarrow p$ for q first, and then affect the p introduced in this process by the substitution of $(p \Rightarrow q)$ for p.

An Example "Proof": Given all this, a proof for a simple theorem such as $p \Rightarrow p$ can be carried out – but it can be quite involved:

P1: From A1, through substitution of $p \Rightarrow p$ for q, we obtain

$$p \Rightarrow ((p \Rightarrow p) \Rightarrow p).$$

P2: From A2, substituting p for s, $p \Rightarrow p$ for p, and p for q, we obtain

$$(p \Rightarrow ((p \Rightarrow p) \Rightarrow p)) \quad \Rightarrow \quad (p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p).$$

P3: Modus ponens between **P1** and **P2** yields

$$(p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p).$$

P4: From A1, substituting p for q, we obtain

$$p \Rightarrow (p \Rightarrow p).$$

Modus ponens between **P4** and **P3** results in $(p \Rightarrow p)$. □

The General Idea of Designing a Propositional Calculus

In a well-designed and useful propositional calculus axiomatization, one must begin with a small set of axioms that are $True$. Then one must provide a small set of rules of inference that allow us to infer "new truths from existing truths." The rules of inference must *only* allow us to prove things that evaluate to $True$. For instance, one must not be able to prove $p \wedge !p$ (always false) or $p \vee q$ (can be set to $False$ by picking $p = q = False$). This attribute of an axiomatization is called *soundness*.

Finally (and most importantly), *every formula that happens to be true must be derivable using only the axioms and the rules of inference.* Then the logical axiomatization is said to be *complete*.

3.7.2 Validity, Satisfiability, Contradiction

A Boolean (propositional) formula is

- Valid, if it is true for every assignment of variables. *Examples:*
  - De Morgan's laws are valid
  - The axioms A1 through A3 given above are valid
- Satisfiable, if it can be satisfied for some assignment of variables. *Examples:*
  - $x + y$ is satisfiable, and so is $x \Rightarrow y$
- Contradiction, if it remains false for every assignment of variables. *Examples:*
  - $x.!x$ is unsatisfiable
  - The negation of every valid formula is unsatisfiable

Valid propositional formulae are equivalent to 1 (or "True") and unsatisfiable propositional formulae are equivalent to 0 ("False"). A truth-table for a valid formula will have 1 in every output position.

The idea behind a proof system (or deductive system) is to start with valid axioms, and using inference rules that are *sound* (truth-preserving), derive new truths from existing truths. We saw a proof of $p \Rightarrow p$ above.
Example: A simple way to check the validity of

$$(s \Rightarrow (p \Rightarrow q)) \quad \Rightarrow \quad (s \Rightarrow p) \Rightarrow (s \Rightarrow q)$$

is to build a truth-table shown in Figure 3.14: In practice, we can use any of the Boolean identities in §3.5 as axioms. We don't need to follow the above "Hilbert style." For example, in §3.6, we proved that the four Nand gates are

s	p	q	$s \Rightarrow p$	$s \Rightarrow q$	$s \Rightarrow (p \Rightarrow q)$	$(s \Rightarrow p) \Rightarrow (s \Rightarrow q)$	$(s \Rightarrow (p \Rightarrow q)) \Rightarrow ((s \Rightarrow p) \Rightarrow (s \Rightarrow q))$
0	0	0	1	1	1	1	1
0	0	1	1	1	1	1	1
0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	1
1	0	0	0	0	1	1	1
1	0	1	0	1	1	1	1
1	1	0	1	0	0	0	1
1	1	1	1	1	1	1	1

Figure 3.14: Checking Validity (*Tautology*) Through a Truth Table. For every setting of s, p, and q, the whole given formula is $True$ (1).

obtaining the functionality of an XOR gate; we used standard mathematical practices of using propositional identities in arriving at our proof.

3.8 Sets Using Characteristic Predicates

The last topic in this chapter is to define sets using characteristic predicates. In particular, we can now derive the operations defined in §2.3 now using mathematical logic:

- Union, written $s_1 \cup s_2$ can be defined as follows:

$$s_1 \cup s_2 = \{x \mid x \in s_1 \ \vee \ x \in s_2\}$$

- Intersection, written $s_1 \cap s_2$ can be defined as follows:

$$s_1 \cap s_2 = \{x \mid x \in s_1 \ \wedge \ x \in s_2\}$$

- 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 $s_1 \triangle s_2$ can be defined as follows:

$$s_1 \triangle s_2 = \{x \mid (x \in s_1 \ \wedge \ x \notin s_2) \vee (x \in s_2 \ \wedge \ x \notin s_1)\}$$

- $s_1 \subseteq s_2$ can be defined as follows:

$$s_1 \subseteq s_2 \text{ exactly when for all } x : \ (x \in s_1 \Rightarrow x \in s_2)$$

- $s_1 \subset s_2$ can be defined as follows:

$$s_1 \subset s_2 \text{ exactly when } (s_1 \subseteq s_2) \text{ and } (s_1 \neq s_2)$$

When we introduce quantification (first order logic), we will introduce $\forall x$ as a symbol that denotes "forall x". This finishes our preliminary discussion of the connection between sets and propositions. Now, let us deepen our understanding of Boolean reasoning and propositional logic by doing the exercises in §3.9.

3.9 *Review and Labs*

In this section, we will go through the contents of Chapter 3 by posing problems and solving them. If you have any doubts about Python expressions given here, please evaluate them and thus confirm your understanding.

1. 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()`.

2. 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. On Page 41, 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.)

4. Argue that whenver you find a combination of values for `x`, `y`, and `z` that makes the expression `((x != 0) or (y >= 0)) and (z <= w)` true, expression `((x == 0) and (y < 0)) or (z > w)` will be false.

5. 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.)

6. 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.

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

8. 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.

   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.

x	y	z
0	0	0
0	1	1
1	0	1
1	1	0

```
def xor(x,y):
    return 0 if (x == y) else 1

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

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

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

Figure 3.15: The XOR Truth-table, Lambda Representation, and Hash-table Representation

9. Construct a series of diagrams as in Figure 3.2 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 3.3 also for these gates. For achieving this, you must write down a truth-table (as on the left-hand side of Figure 3.3), 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 3.15.

10. 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.

    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 $\overline{a} + \overline{b}$. 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 3.16.

11. 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.

12. Using Python's set comprehension notation and the template expression `(x,y)`, write down, in Python, a set comprehension expression for the cartesian product of odd numbers below 50 and even numbers be-

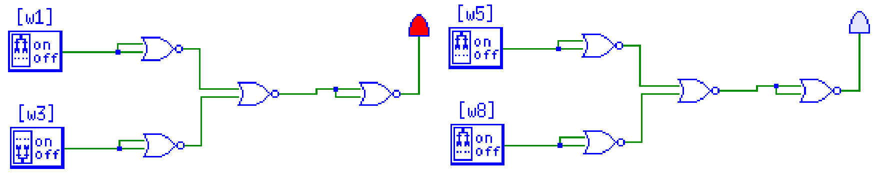


Figure 3.16: NAND using NOR

low 40. Execute this cartesian product experssion in Python 2.7 and also print its length.

```
>>> { (x,y) for x in range(50) for y in range(40) if (x%2 == 1) and (y%2 == 0)}

>>> len({ (x,y) for x in range(50) for y in range(40) if (x%2 == 1) and (y%2 == 0)}) = 500
```

13. Write down the signature of the function `reduce` where the third argument of `reduce` is also supplied.

    The usage is `reduce(func, list, initial)`. For instance

    ```
    reduce(lambda x,y: x*y, [1], 0) generates 0

    reduce(lambda x,y: x*y, [1]) generates 1

    reduce(lambda x,y: x*y, [1,2,3], 0) generates 0

    reduce(lambda x,y: x*y, [1,2,3]) generates 6
    ```

    Thus, the signature of `reduce` is

    $((\alpha \times \alpha \mapsto \alpha) \ \times \ \alpha \text{ list} \ \times \ \alpha) \ \mapsto \ \alpha$

    Here, α is some type such as `int`.

    This says that `reduce` takes a binary function of type $(\alpha \times \alpha) \mapsto \alpha$, a list of αs, and a single α (the init value), and produces a single α.

14. 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.

15. 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 3.17

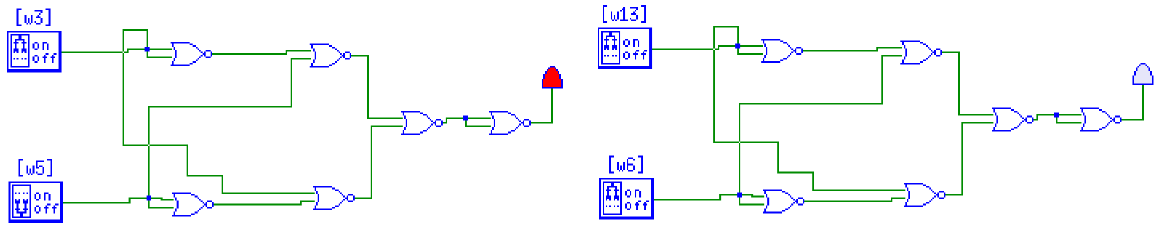


Figure 3.17: XOR using NOR

16. 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.

17. Show that the Implication gate described in Figure 3.5 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 3.18 for details of this circuit.

18. Show that the Implication gate described in Figure 3.5 is universal (alternate proof):

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

    - $nand(P, Q) = negPandQ$
    - $negPandQ = PandQ \Rightarrow 0$
    - $PandQ = ((P \Rightarrow negQ) \Rightarrow 0)$
    - $newQ = (Q \Rightarrow 0)$

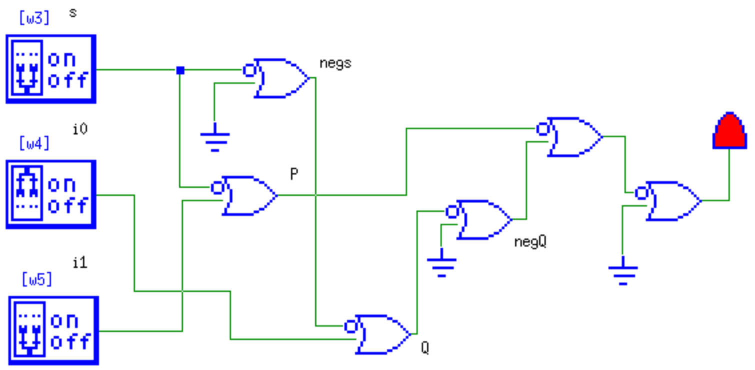


Figure 3.18: mux21 using Imp

Once a NAND is realized, we have a universal gate. The NAND realized using IMP gates is shown in Figure 3.19.

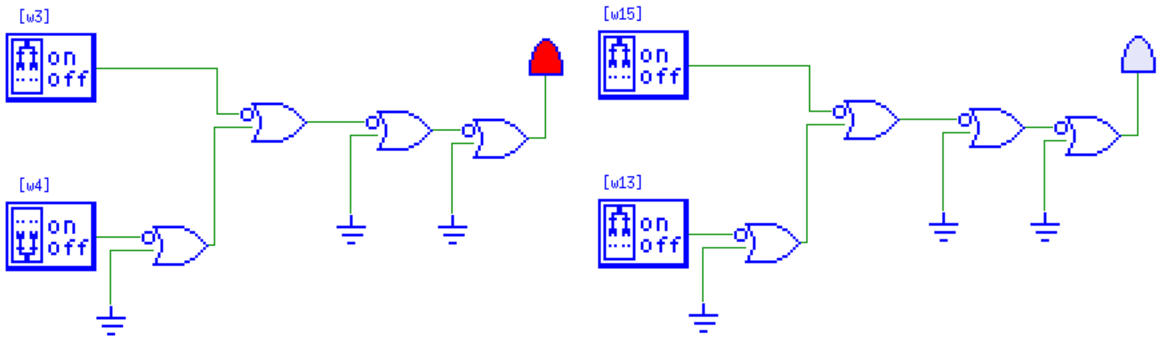


Figure 3.19: XOR using NOR

19. Establish the identity $x+!x.y \; = \; x+y$.

Answer:

$(x+!x.y)$
$= (x+!x).(x+y)$
$= 1.(x+y)$
$= x+y$

20. 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:

$!(x \Rightarrow y) + (x.y)$
$=!(!x + y) + (x.y)$
$= (x.!y) + (x.y)$
$= x.(!y + y)$
$= x$

21. Show that $(\neg q \Rightarrow \neg p) \quad \Rightarrow \quad (p \Rightarrow q)$ is valid.

Answer: Let us use ! uniformly in place of $\neg$:

The first approach is to use propositional identities:
$(!q \Rightarrow !p) \quad \Rightarrow \quad (p \Rightarrow q)$
$=!(!q \Rightarrow !p) \ + \ (p \Rightarrow q)$
$= (!q.p) \ + \ (!p + q)$
$= (!p + q + !q).(p + !p + q)$
$= (1).(1)$
$= 1.$

Answer: The second approach is by building truth tables:

p	q	$\neg q$	$\neg p$	$\neg q \Rightarrow \neg p$	$p \Rightarrow q$	$(\neg q \Rightarrow \neg p) \Rightarrow (p \Rightarrow q)$
0	0	1	1	1	1	1
0	1	0	1	1	1	1
1	0	1	0	0	0	1
1	1	0	0	1	1	1

Figure 3.20: Checking Validity (*Tautology*) Through a Truth Table. For every setting of p and q, the whole given formula is $True$ (1).

Answer: The third approach is by negating the given formula and showing that it is not satisfiable.

$!((!q \Rightarrow !p) \quad \Rightarrow \quad (p \Rightarrow q))$
$= ((!q \Rightarrow !p) \ . \ (p.!q))$
$= ((q + !p) \ . \ (p.!q))$
$= ((q.p.!q) + (!p.p.!q))$
$= (0 + 0)$
$= 0$, which is not satisfiable. Hence the given formula is valid.

22. Show that $p \Rightarrow q$ is not valid.

    Answer: to show that a formula is not valid, take its negation and find a way to satisfy it. The negation of $p \Rightarrow q$ is $p.!q$. This can be satisfied by choosing $p = 1$ and $q = 0$.

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

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

    Answer: $\{(2,0),(2,1),(2,2),(3,4),(7,8),\ldots\}$

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

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, True]

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

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

>>> len([(x,y) for x in range(20) for y in range(20)
      if ((x%2 == 0) or (isPrime(x) and isComposite(y)))])
284

>>> len([(x,y) for x in range(20) for y in range(20)])
400
```

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

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

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

Some pairs not in the set are $\{(1,100), (2,100), (1,1000), \ldots\}$. 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
```

25. 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 ^:

```
>>> {(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

>>>
```

Chapter 4

The Higher Order of Programming

Chapter 5

Putative Proper Proof Principles

Chapter 6

Googols of Graphs

Chapter 7

Boosted Boolean Brawn

Chapter 8

Combinatorial Cornucopia

Chapter 9

Rocking Recursion and Impressive Induction

Chapter 10

Probable Certainties and Certain Probabilities

Bibliography

[1] http://en.wikipedia.org/wiki/Boolean_algebra.

[2] Alonzo Church. *Introduction to Mathematical Logic*. Princeton University Press, October 1996.

[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] Allen Downey. Think stats. http://www.greenteapress.com/thinkstats/thinkstats.pdf.

[5] Sumit Gulwani. Flashfill. See the Youtube video http://www.youtube.com/watch?v=qHkgJFJR5cM and other details at http://research.microsoft.com/en-us/um/people/sumitg/flashfill.html.

[6] Paul Halmos. Naïve set theory, 1960.

[7] http://en.wikipedia.org/wiki/Human_genome.

[8] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodov, 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] http://en.wikipedia.org/wiki/Lambda_calculus.

[10] Lorraine Lica. http://home.earthlink.net/~llica/wichthat.htm.

[11] http://docs.python.org/release/2.3.5/ref/comparisons.html.

[12] http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/
6-01sc-introduction-to-electrical-engineering-and-computer-science-i-spri
python-tutorial/.

[13] http://msdn.microsoft.com/en-us/library/dd293608.aspx.

[14] F. Ruskey, C. D. Savage, and S. Wagon. The search for simple symmetric venn diagrams. *Notth Amer. Math. Soc.*, 53:1304–1311, 2006.

[15] http://en.wikipedia.org/wiki/Ordered_pair\#Kuratowski_definition.

[16] The University of Victoria website http://www.theory.csc.uvic.ca/~cos/inf/comb/SubsetInfo.html\#Venn.

[17] Guido van Rossum. http://en.wikipedia.org/wiki/Guido_van_Rossum.

[18] Mitchell Wand. *Induction, Recursion, and Programming.* Elsevier Science, August 1980.

[19] The Wolfram website http://mathworld.wolfram.com/VennDiagram.html.