Formal Methods for Linguistics

Bruce Tesar

January 2007
Contents

0.1 Preface ................................ vii
0.2 Further Reading ............................. vii

1 Mathematical Fundamentals ........................ 1
  1.1 Collections and Mappings ....................... 1
    1.1.1 Sets and Propositions .................... 1
    1.1.2 Types of Mathematical Objects ............ 2
    1.1.3 Types of Propositions .................... 5
    1.1.4 Exercises ................................ 6
  1.2 Algebras ................................ 7
    1.2.1 What is an Algebra? ....................... 7
    1.2.2 Boolean Algebra .......................... 8
    1.2.3 Exercises ................................ 9
  1.3 Order ...................................... 10
    1.3.1 Partial Ordering .......................... 10
    1.3.2 Total Ordering ........................... 10
    1.3.3 Lattices ................................ 10
    1.3.4 The Power Set Lattice .................... 11
    1.3.5 Exercises ................................ 12
  1.4 More About Order ................................ 13
    1.4.1 Complete Lattices ......................... 13
    1.4.2 Composite Orders .......................... 14
    1.4.3 Exercises ................................ 14
  1.5 The Semantics of Plurals ....................... 14
    1.5.1 The Phenomena of Interest ................. 14
    1.5.2 Free Join Semilattices .................... 16
    1.5.3 Linguistic Analysis ....................... 16
    1.5.4 Pulling it all together: An illustration ... 18
    1.5.5 Exercises ................................ 21

2 Cardinality ..................................... 23
  2.1 Finite Combinatorics .......................... 23
    2.1.1 The Sum Rule ............................. 23
    2.1.2 The Product Rule .......................... 23
    2.1.3 Permutations ............................. 23
## CONTENTS

2.1.4 Combinations ........................................... 24
2.1.5 The Binomial Theorem ................................. 24
2.1.6 Exercises ................................................ 25

2.2 Countable Sets ............................................ 26
2.2.1 Equivalence ........................................... 26
2.2.2 The Integers ........................................... 26
2.2.3 The Rational Numbers ................................. 27
2.2.4 Exercises ................................................ 27

2.3 Uncountable Sets .......................................... 28
2.3.1 The Real Numbers ..................................... 28
2.3.2 The Unboundedness of Cardinality .................... 28
2.3.3 Exercises ................................................ 29

3 Computation .................................................. 31
3.1 Formal Languages ......................................... 31
3.1.1 Strings ............................................... 31
3.1.2 Languages ............................................ 32
3.1.3 Exercises .............................................. 32

3.2 Formal Rewrite Grammars .................................. 33
3.2.1 Exercises .............................................. 35

3.3 Regular Grammars ......................................... 36
3.3.1 Definitions ........................................... 36
3.3.2 Regular Languages ................................... 36
3.3.3 Recursion ............................................. 38
3.3.4 Limitations of Regular Grammars .................... 39
3.3.5 Exercises .............................................. 41

3.4 Context-Free Grammars ..................................... 41
3.4.1 Exercises .............................................. 43

3.5 Unrestricted Grammars .................................... 43
3.5.1 Exercises .............................................. 44

3.6 Finite State Automata ..................................... 44
3.6.1 Finite State Acceptors ................................ 44
3.6.2 Finite State Transducers ............................. 46
3.6.3 Exercises .............................................. 48

3.7 Pushdown Automata ........................................ 49
3.7.1 Exercises .............................................. 50

3.8 Turing Machines ........................................... 51
3.8.1 Turing Machine Definitions and Conventions ....... 51
3.8.2 Example Machines .................................... 53
3.8.3 Universal Turing Machines ............................ 56
3.8.4 Exercises .............................................. 56

3.9 Computability Theory ..................................... 57
3.9.1 The Chomsky Hierarchy ............................... 57
3.9.2 Uncomputability ...................................... 58
4 Proof ..................................................... 61
  4.1 Introduction ..................................... 61
  4.2 Formal Systems of Logic .............................. 61
  4.2.1 What is a Logic? ................................ 61
  4.2.2 First Order Logic (FOL) ......................... 62
  4.2.3 Deduction ...................................... 64
  4.2.4 Inconsistency .................................. 64
  4.3 Resolution ......................................... 65
  4.3.1 Horn Clauses .................................. 65
  4.3.2 Implications as Horn Clauses ....................... 66
  4.3.3 Resolution with Horn Clauses ....................... 67
  4.4 Prolog Basics ...................................... 69
  4.4.1 Notation ...................................... 69
  4.4.2 Databases and Queries ......................... 69
  4.4.3 Search in Prolog ................................ 70
  4.4.4 Exercises ..................................... 73
  4.5 Database Prolog .................................... 74
  4.5.1 Variables ..................................... 74
  4.5.2 Search with Variables ....................... 74
  4.5.3 Recursive Rules ............................. 76
  4.5.4 Exercises ..................................... 76
  4.6 Pure Prolog ....................................... 77
  4.6.1 Unification .................................. 77
  4.6.2 Lists ......................................... 78
  4.6.3 Soundness and Completeness of Inference ............ 79
  4.6.4 Exercises ..................................... 79
  4.7 Parsing with Prolog ................................ 80
  4.7.1 Representing String Positions with Difference Lists 80
  4.7.2 Grammar Rules as Prolog Rules ................... 80
  4.7.3 Constructing Tree Representations ................. 81
  4.8 Mathematical Induction ............................. 82
  4.8.1 Summing Numbers (Again) ....................... 82
  4.8.2 Induction over Grammar Derivations ............... 83
  4.8.3 Exercises ..................................... 83

5 Mathematical Analysis ................................... 85
  5.1 Sequences and Series ............................... 85
  5.1.1 Infinite Sequences ............................ 85
  5.1.2 Exercises ..................................... 87
  5.2 Infinite Series .................................... 87
  5.2.1 Sequences of Partial Sums ....................... 87
  5.2.2 Series with Positive and Negative Terms .......... 88
  5.3 Derivatives ....................................... 89
  5.3.1 Rate of Change ................................ 89
  5.3.2 Limits of Functions ............................ 91
  5.3.3 Rates of Change Provide Useful Information ....... 94
CONTENTS

5.3.4 Derivatives Define Functions .................................. 96
5.4 Integration ..................................................................... 97
5.4.1 Going the other way: Adding up rates of change .......... 97
5.4.2 Constantly Changing Rates ....................................... 98
5.4.3 Definite Integrals .................................................... 100
5.4.4 Improper Integrals ................................................... 102

6 Probability and Statistics ................................................. 105
6.1 The Nature of the Beast(s) ............................................. 105
6.2 Basic Probability .......................................................... 106
6.2.1 Outcomes, Sample Spaces, and Events .................... 106
6.2.2 Probability Distributions ........................................ 106
6.2.3 Bayes Theorem ....................................................... 107
6.2.4 Exercises .............................................................. 110
6.3 Random Variables ........................................................ 111
6.3.1 Properties of Numeric Random Variables ............... 112
6.3.2 Combining Random Variables ................................. 113
6.3.3 Binomial Random Variables ..................................... 113
6.3.4 Exercises .............................................................. 115
6.4 Continuous Random Variables ................................. 116
6.4.1 Probability Density Functions ................................. 116
6.4.2 Mean and Variance .................................................. 118
6.4.3 Interpreting Probabilities ........................................ 118
6.4.4 Normal Random Variables ...................................... 118
6.4.5 Exercises .............................................................. 121
6.5 Sample Distributions ...................................................... 122
6.5.1 Random Samples ..................................................... 122
6.5.2 Binomial Sample Distribution .................................. 122
6.5.3 The Central Limit Theorem ..................................... 123
6.6 Point Estimators and Confidence Intervals .................... 124
6.6.1 Descriptive Statistics .............................................. 124
6.6.2 Estimating \( p \) ......................................................... 125
6.6.3 Estimating Means ................................................... 126
6.6.4 Adjusting Confidence Levels ................................... 126
6.6.5 Exercises .............................................................. 127
6.7 Hypothesis Testing ......................................................... 127
6.7.1 Null and Alternative Hypotheses ............................. 127
6.7.2 Large Sample Test for the Population Mean ............. 128
6.7.3 Alternative Hypotheses ........................................... 129
6.7.4 Exercises .............................................................. 130
0.1 Preface

These are the course notes for Linguistics 610, Formal Methods for Linguistics. They constitute the main text for the course.

Some of the material on prolog in the chapter on proof and provability was directly influenced by my past use of Fernando Pereira and Stuart Shieber’s *Prolog and Natural-Language Analysis* as a secondary text. Similarly, parts of the chapter on probability and statistics were influenced by my past use of Larry Gonick and Woollcott Smith’s *The Cartoon Guide to Statistics* as a “gentle” introduction to the subject matter. Students are warmly encouraged to consult these books for further material on their respective topics.

0.2 Further Reading

Chapter 1

Mathematical Fundamentals

1.1 Collections and Mappings

1.1.1 Sets and Propositions

A set $S$ is a collection of stuff. The only significant property distinguishing a set (for now) is that membership in a set is a binary affair: an object either is an element of a set ($x \in S$) or it is not ($x \notin S$). An object cannot be in a set “three times” or “partly a member” of a set, nor can it be “neither a member nor not a member of $S$”.

A proposition $P$ is a statement that is either true or false (but not both). A proposition cannot be “true three times” or “partly true”, nor can it be “neither true nor false.” Notice that this distinguishes our concept of proposition from some concepts of truth values in semantics, where sentences like “The king of France is bald” are regarded as neither true nor false.

None of these are really definitions; they are just descriptions of concepts that should be familiar. Many things can be defined in terms of the above concepts. But first, it can be interesting to see how sets and propositions are related.

A set can be viewed as consisting of propositions of the form $x \in S$, each of which is true if $x$ is in fact an element of $S$, and false otherwise. In fact, the propositions can be bundled into a predicate $\text{InS} [x]$, which is true precisely when its argument $x$ is an element of the set. In general, a predicate is an entity which combines with one or more arguments to form a proposition. Going the other way, a one-argument predicate $P$ can be viewed as a set $S_P$ containing precisely those objects $x$ for which $P[x]$ is true.

Two simple operations that can be applied to groups of sets are union and intersection. The union of a group of sets is the set of all objects that are an element of at least one of the sets in the group. The intersection of a group of
sets is the set of all objects that are an element of every one of the sets in the group. Logic has directly analogous operations. The logical *disjunction* ("OR") of a group of propositions is a proposition which is true if at least one of the propositions in the group is true. The logical *conjunction* ("AND") of a group of propositions is a proposition which is true only if every one of the propositions in the group is true.

**Example 1.1** Consider collections defined over the universe of objects \{1, 2, 3, 4, 5, 6, 7, 8, 9\}.

- \( S_1 = \{1, 2, 3, 4\} \)
- \( S_2 = \{8, 2, 3, 7\} \)
- \( P_1[x] \) is true when \( x \) is 1, 2, 3, or 4, and false when \( x \) is 5, 6, 7, 8, or 9.
- \( P_2[x] \) is true when \( x \) is 8, 2, 3, or 7, and false when \( x \) is 1, 4, 5, 6, or 9.
- **Union:** \( S_1 \cup S_2 = \{1, 2, 3, 4, 7, 8\} \)
- **Disjunction:** \( P_1[x] \lor P_2[x] \) is true when \( x \) is 1, 2, 3, 4, 7, or 8, and false when \( x \) is 5, 6, or 9.
- **Intersection:** \( S_1 \cap S_2 = \{2, 3\} \)
- **Conjunction:** \( P_1[x] \land P_2[x] \) is true when \( x \) is 2 or 3, and false when \( x \) is 1, 4, 5, 6, 7, 8, or 9.

The correspondence between the set theoretic view and the logical view is more apparent when the logical case is cast in terms of one-argument predicates. The logical disjunction of a group of predicates is a predicate which is true of an object if at least one of the disjoined predicates is true of that object. The logical conjunction of a group of predicates is a predicate which is true of an object only if every one of the conjoined predicates is true of that object.

Being aware of and comfortable with these different views is quite useful. For instance, if you have a linguistic problem that you understand in terms of sets, and a computational technique defined over predicates, being able to translate easily between sets and predicates may allow you to apply the computational technique to the linguistic problem (whether or not doing so is actually a good idea would depend, of course, on many other details).

### 1.1.2 Types of Mathematical Objects

In mathematics, one can construct composite types of objects by combining other types in various ways. This section presents a few ways of constructing composite mathematical objects. For illustration purposes, types of objects such as the natural numbers \( \mathbb{N} = \{0, 1, 2, 3, \ldots\} \), the integers \( \mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} \), and the English alphabet \{a, b, c, \ldots\}, will be treated as ‘basic’ types, even though it is possible (and sometimes desirable) to construct entities like the natural numbers out of other types.
Sets and Power Sets

The letter ‘a’ is a letter. The set \{a\} is a set of letters (it happens to contain only one). These two objects are of different types: a set of letters is not itself a letter. Another type of object is a set of sets of letters; an example would be \{\{a,b,c\}, \{e\}, \{b,y\}, \{y,z\}\}.

Call the set of all letters \(A\). The set of all objects of type “set of letters” is the set of all subsets of \(A\), often called the power set of \(A\), denoted \(\mathcal{P}(A)\). Both \(A\) and \{z\} are elements of \(\mathcal{P}(A)\); both are “sets of letters”.

Example 1.2 The power set of \(S = \{x, y, z\}\) is
\[
\mathcal{P}(S) = \{\emptyset, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}
\]

But now, one can ask, what type of object is \(\mathcal{P}(A)\)? It obviously isn’t a letter. It isn’t an element of itself, so it isn’t a set of letters. It is a set of “sets of letters”. The set of all sets of sets of letters is \(\mathcal{P}(\mathcal{P}(A))\), that is, the power set of the power set of the letters. The power set operator \(\mathcal{P}\) can be applied an unbounded number of times to define objects that are sets of sets of sets of...sets of letters. Even with only one basic type (letters), combined with the conceptual ability to create sets of objects, an infinite number of types of mathematical objects can be defined.

Cartesian Products

A Cartesian product \(T_1 \times T_2\) of sets \(T_1\) and \(T_2\) is the set of all ordered pairs \((t_1, t_2)\) such that \(t_1 \in T_1\) and \(t_2 \in T_2\). Ordered pairs easily generalize to ordered n-tuples, and so do the corresponding Cartesian products. An ordered 5-tuple \((t_1, t_2, t_3, t_4, t_5)\) is an element of some five-way Cartesian product \(T_1 \times T_2 \times T_3 \times T_4 \times T_5\). The types being combined in a Cartesian product need not be distinct: an ordered pair of integers is a member of \(\mathbb{Z} \times \mathbb{Z}\). In fact, Cartesian products of a single set are quite common. The n-way Cartesian product of a single set will sometimes be denoted \(S^n\).

A better term for some ordered n-tuples might be “labeled n-tuple”. Consider a product that represents a person’s name, department and phone number. An example of such a 3-tuple might be (Bruce, Linguistics, 2-6933). But this really should be written as (Name:Bruce, Department:Linguistics, Phone:2-6933). Notice that this could just as reasonably be written (Department:Linguistics, Name:Bruce, Phone:2-6933). An ordered n-tuple simply uses notational order as a kind of label. Notational order is convenient because there are (usually) no a priori semantics assigned to them. This is like using completely generic labels, such as L_1, L_2, and so forth.

An ordered n-tuple is distinguished from a set in several respects. A particular object can appear more than once in an ordered n-tuple, for example, \(2,7,2,1\). Changing the order in which the objects appear in an ordered n-tuple changes the identity of the object: \(2,1\) is not the same object as \(1,2\).
Relations

A relation \( R \) for a sequence of sets \( S_1, S_2, \ldots, S_n \) is a subset of the Cartesian product \( S_1 \times S_2 \times \ldots \times S_n \). Each element of a relation is thus an ordered \( n \)-tuple. A binary relation is a relation on two sets \( S_1 \) and \( S_2 \), a subset of \( S_1 \times S_2 \).

One special kind of relation is an \( n \)-ary relation on a single set. A 3-ary relation on a set \( S \) is a subset of the Cartesian product \( S^3 = S \times S \times S \). The significant characteristic is that the same set is used for all the factors of the Cartesian product.

Among binary relations on (single) sets, there are further properties of interest. Consider a binary relation \( R \) on the set \( S \) (meaning \( R \subseteq S \times S \)).

- The relation is reflexive if \( (x, x) \in R \) for each \( x \in S \).
- The relation is symmetric if for each \( (x, y) \in R \), \( (y, x) \in R \) also.
- The relation is antisymmetric if, whenever \( (x, y) \in R \) and \( (y, x) \in R \), then \( x = y \).
- The relation is transitive if, whenever \( (x, y) \in R \) and \( (y, z) \in R \), then \( (x, z) \in R \) also.

If a binary relation on a set is reflexive, symmetric, and transitive, it is an equivalence relation.

Example 1.3 The odd/even parity relation is a relation on the natural numbers. Two numbers \( x \) and \( y \) have the same odd/even parity, that is, \( (x, y) \) is an element of the odd/even parity relation, if and only if \( x \) and \( y \) are both even, or are both odd.

Because a relation on a sequence of sets \( \{S_1, \ldots, S_n\} \) is a subset of the Cartesian product of the sets, the set of all possible relations on the sequence of sets is the set of all possible subsets of the Cartesian product, i.e., the power set \( \mathcal{P}(S_1 \times \ldots \times S_n) \).

Functions

A function \( f : D \to C \) is a map from one set, the domain \( D \) of the function, to another, the codomain \( C \) of the function. A function must map each member of the domain to exactly one member of the codomain. It need not map to every member of the codomain, and a member of the codomain can have more than one domain member mapped to it.

A function \( f \) can be viewed as a set of ordered pairs \( S = \{(d, c)\} \) such that \( f(d) = c \). All such functions are binary relations on \( D \times C \). The set \( F \) of all possible functions from \( D \) to \( C \) is thus a specific subset of the set of all possible relations between \( D \) and \( C \): \( F \subseteq \mathcal{P}(D \times C) \).
Example 1.4 The function $\text{PlusOne}(x)$ maps each natural number to the number one greater (it maps $x$ to $x+1$). Thus, $\text{PlusOne}(0) = 1$, $\text{PlusOne}(5) = 6$, and so forth. This function can be viewed as a set of ordered pairs:
\[
\{(0, 1), (1, 2), (2, 3), \ldots \}
\]

Example 1.5 General addition on the natural numbers can be viewed as a function that maps an ordered pair of natural numbers to a single natural number. Thus, $+(1, 2) = 3$, and $+(5, 5) = 10$. This function can be viewed as a set of ordered pairs:
\[
\{((0, 0), 0), ((0, 1), 1), ((0, 2), 2), \ldots , ((1, 0), 1), ((1, 1), 2), ((1, 2), 3), \ldots \}
\]

The function in Example 1.5 can be thought of as applying to two arguments, in a sense, the two numbers being added together. Our definition of a function as mapping a single object to a single other object can be preserved by taking the collection of objects we want to apply the function to, and grouping them into a single object, such as an ordered n-tuple.

A function can be viewed as a predicate by viewing the set $S$ of ordered pairs just described as a predicate $\text{pred}([d, c])$ which is true if and only if the function $f$ maps $d$ to $c$ (or, equivalently, if the ordered pair $(d, c) \in S$).

Functions can be combined just like other mathematical objects. A Cartesian product of sets of functions $F_1 \times F_2$ can be realized, where each member of the space is an ordered pair of functions $(f_1, f_2)$. It is easier to understand how this could be done once you realize that a function is itself just a set.

There are certain classifications of functions that prove particularly useful. A function is called an **injection** if no two members of the domain are mapped to the same member of the codomain. Injections are said to be one-to-one functions. A function is called a **surjection** if every member of the codomain has at least one member of the domain mapped to it. Surjections are said to be onto functions (“onto” is here being used as an adjective). A function is a **bijection** if it is both an injection and a surjection (that is, if it is both one-to-one and onto). With a bijection, every member of the codomain is mapped to by exactly one member of the domain.

### 1.1.3 Types of Propositions

In logic, one can construct composite propositions by combining other propositions. One can do the same with predicates.

A **logical operator** combines propositions to form a new, composite, proposition. In the process, it **defines** the truth value of the composite proposition in terms of the truth values of its arguments.

It is natural to talk about logical operators using function-like terminology, describing the propositions being defined as arguments to the logical operator.

**Example 1.6** The logical operator $\neg$ (“NOT”) is a one-argument logical operator resulting in a proposition that is false whenever its argument is true, and true whenever its argument is false.
Example 1.7 The implication operator $\rightarrow$ is a two-argument operator creating the proposition $(\alpha \rightarrow \beta)$. The relationship between the truth values of the propositions can be summarized with a truth table:

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\beta$</th>
<th>$(\alpha \rightarrow \beta)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td>$F$</td>
<td>$T$</td>
</tr>
<tr>
<td>$F$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$</td>
<td>$F$</td>
<td>$F$</td>
</tr>
<tr>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
</tbody>
</table>

Familiar operators like conjunction and disjunction are logical operators.

1.1.4 Exercises

For the following exercises, assume a universe of objects $\mathbb{U} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, and the sets $S_1 = \{1, 2, 3, 4, 5\}$ and $S_2 = \{2, 4, 6, 8\}$. The predicate $\text{even}[x]$ is true if $x$ is even; the predicate $\text{GT3}[x]$ is true if $x$ is greater than 3.

Exercise 1.1.1 Write $S_1 \cup S_2$ as a set.

Exercise 1.1.2 Write the truth value of the predicate $\text{even}[x] \lor \text{GT3}[x]$ for each number in $\mathbb{U}$.

Exercise 1.1.3 Write $S_1$ and $S_2$ as predicates, by listing the truth values of each for each number in $\mathbb{U}$. Then:

(a) Write the truth values for the predicate $S_1[x] \land S_2[x]$.
(b) Write the truth values for $S_1[x] \lor \text{even}[x]$.

Exercise 1.1.4 Write $\text{even}[x]$ and $\text{GT3}[x]$ as sets containing the numbers in $\mathbb{U}$ for which the predicate is true. Then:

(a) Write the set $\text{even} \cap \text{GT3}$.
(b) Write the set $\text{GT3} \cup S_2$.

Exercise 1.1.5 List all of the possible functions $\{a, b\} \rightarrow \{1, 2, 3\}$. Give each possible function in the form of a set.

Exercise 1.1.6 Write the power set of $S_1 \cap S_2$.

Exercise 1.1.7 Could $\{\{} , \{1\} , \{2\} , \{3\} , \{1, 2, 3\}\}$ be the power set of some other source set? If so, give the source set; if not, explain why.

Exercise 1.1.8 Could $\{\{} , \{a, b\} , \{c\} , \{\{a, b\} , c\}\}$ be the power set of some other source set? If so, give the source set; if not, explain why.
1.2 Algebras

1.2.1 What is an Algebra?

An algebraic structure is a set $S$, called the carrier, with one or more operations on $S$. An operation on a carrier set $S$ is a function of the form $S^n \to S$. As before, we can switch back and forth between viewing the operation as mapping from a single entity which is an n-tuple, and viewing the operation as mapping from n arguments (get used to it; we will be switching back and forth like this a lot).

Example 1.8 An algebraic structure known as $\mathbb{Z}_3$ is defined on the carrier set $\{0, 1, 2\}$ with an addition operation $+$ and a multiplication operation $\ast$. The addition operation is like normal addition, except that instead of reaching 3, the numbers start over at 0: $1 + 2 = 0$, $2 + 2 = 1$. Multiplication is similar: $1 \ast 2 = 2$, $2 \ast 2 = 1$.

Algebraic structures are themselves mathematical objects. $\mathbb{Z}_3$ is a 3-tuple $(S, +, \ast)$, where $S$ is the carrier set, $+$ is the set of ordered pairs constituting the addition operation, and $\ast$ is the set of ordered pairs constituting the multiplication operation.

Example 1.9 The multiplication operation for $\mathbb{Z}_3$ is the following set.

$$\ast = \{(0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2)\}$$

An algebraic structure can have an infinite carrier. Consider normal arithmetic on the integers, with addition, multiplication, and negation defined in the normal way. Addition is a 2-argument operation, mapping each pair of integers to a single integer; the same is true of multiplication. Negation is a 1-argument operation.

Algebras are often classified according to the properties of their operations, and properties of special elements contained in the carrier. Three properties of particular interest all involve 2-argument operations.

- An operation $\Box$ is commutative if $x \Box y = y \Box x$ for all $x$ and $y$ in the carrier.
- An operation $\Box$ is associative if $x \Box (y \Box z) = (x \Box y) \Box z$ for all $x$, $y$, and $z$.
- A particular element $e$ is an identity under $\Box$ if $x \Box e = x = e \Box x$ for all $x$.
- If there exists an identity element $e$ such that $x \Box y = e = y \Box x$, then $x$ is an inverse of $y$ under $\Box$. 
1.2.2 Boolean Algebra

Distributivity

Let $\square$ and $\triangledown$ be commutative binary operations on $X$. The operation $\square$ is distributive over $\triangledown$ if $x \square (y \triangledown z) = (x \square y) \triangledown (x \square z)$ for all $x, y, z \in X$.

Definition of a Boolean Algebra

Let $S$ be a set containing 0, 1, and possibly other elements. Let $\bar{x}$ be a unary (1-argument) operation on $S$, and let $*$ and $+$ be binary operations on $S$. Further, let 1 be an identity for $*$ and 0 be an identity for $+$. The algebraic structure formed by the carrier $S$ and these operations is a boolean algebra if it satisfies the following four axioms:

1. $*$ and $+$ are both commutative and associative on $S$.
2. Each of $*$ and $+$ are distributive over the other.
3. $0 \neq 1$.
4. $x * \bar{x} = 0$ and $x + \bar{x} = 1$ for each $x \in S$.

Watch the definitions carefully; not all of your arithmetic intuitions will work.

Some theorems follow from these axioms. For example, the values of $x$ with respect to the identities, $\bar{0} = 1$ and $\bar{1} = 0$, follow as consequences of the axioms.

**Theorem 1.1** In a boolean algebra, $\bar{0} = 1$ and $\bar{1} = 0$.

**Proof.** Consider the axiom $x * \bar{x} = 0$ with $x = 1$. This results in $1 * \bar{1} = 0$. Because 1 is the multiplicative identity, it follows that $\bar{1} = 0$.

The proof that $\bar{0} = 1$ is left as an exercise. ■

The most commonly used boolean algebra uses as its carrier the set $\mathbb{B} = \{0, 1\}$. But that is not the only possible boolean algebra; there exist structures with carriers containing more than two elements that are nevertheless boolean algebras as defined above.

The Relation to Logic

Boolean algebra with carrier $\mathbb{B} = \{0, 1\}$ can be interpreted in terms of familiar notions of logic. Label 0 as false, 1 as true, $\bar{x}$ as logical negation, $+$ as disjunction, and $*$ as conjunction.
1.2. ALGEBRAS

Boolean expressions

A boolean expression is an expression constructed from instances of the values \( \mathbb{B} = \{0, 1\} \) using only the operations \( \land, +, \text{ and } \ast \). Notice that this definition is restricted to the boolean algebra with only 0 and 1 in the carrier. A boolean operation is a function from a Cartesian product of the carrier to the carrier, that is, \( \mathbb{B}^n \to \mathbb{B} \). Boolean operations are often referred to as boolean functions.

Example 1.10 The exclusive OR operation can be defined with a boolean expression: \( \text{XOR}(x, y) = (x \land \bar{y}) \lor (\bar{x} \land y) \). XOR is a binary operation, mapping \( \mathbb{B} \times \mathbb{B} \to \mathbb{B} \).

A common application of boolean algebra is to simplify logical expressions. There are several properties of boolean algebra that are particularly useful in simplifying expressions.

Theorem 1.2 \( x \land (x \lor y) = x \), and \( x \lor (x \land y) = x \) (Absorption Laws).

Theorem 1.3 \( x \lor y = \bar{x} \land \bar{y} \), and \( x \land y = \overline{x \lor y} \) (De Morgan’s Laws).

1.2.3 Exercises

Exercise 1.2.1 In \( \mathbb{Z}_3 \) of example 1.8, what is \( 2 * (1 + 1) \)?

Exercise 1.2.2 In \( \mathbb{Z}_3 \), is it true that \( 2 * (1 + 2) = (2 * 1) + (2 * 2) \)? Explain why or why not.

Exercise 1.2.3 Write out the addition operation \( + \) for \( \mathbb{Z}_3 \). Your answer should take a form analogous to that given for the multiplication operation in 1.9.

Exercise 1.2.4 Add a one-argument complement operator to \( \mathbb{Z}_3 \), where the complement of 1 is denoted \( -1 \). Adding a complemented number is like adding a positive number, only you move in the reverse direction: \( 2 + (-1) = 1 \). Give the value of the complement operation for each of the numbers \( \{0, 1, 2\} \). REMEMBER: the result value of the complement operation, like any operation on \( \{0, 1, 2\} \), will be an element of \( \{0, 1, 2\} \); \( -1 \) equals some value contained in \( \{0, 1, 2\} \).

Exercise 1.2.5 For a boolean algebra, simplify \( (x \lor z) \land (x \lor w) \land (y \lor z) \land (y \lor w) \). Justify each step of the simplification.

Exercise 1.2.6 For a boolean algebra, prove that \( \bar{0} = 1 \) (see Theorem 1.1).

Exercise 1.2.7 Is the operation \( \bar{x} \) in a boolean algebra the same as the negation operation in ordinary integer arithmetic (where the negation of 5, \( \bar{5} \), would be taken to be -5)? Justify your answer.
1.3 Order

1.3.1 Partial Ordering

If \( X \) is a set, and \( \leq \) is a reflexive, antisymmetric, transitive, binary relation on \( X \), then \( \leq \) is a partial ordering, and the ordered pair \((X, \leq)\) is a partially ordered set.

Example 1.11 Let \( X = \{\alpha, \beta, \gamma, \delta\} \). Let \( \leq \) be the relation \[(\alpha, \alpha), (\beta, \beta), (\gamma, \gamma), (\delta, \delta), (\alpha, \beta), (\alpha, \gamma), (\alpha, \delta), (\beta, \delta)\] Then \((X, \leq)\) is a partially ordered set. Observe that \(\alpha \leq \beta\) and \(\alpha \leq \gamma\). Note also that \(\beta\) and \(\gamma\) are not ordered with respect to each other.

Example 1.12 Let \( X = \{\{a\}, \{c\}, \{a, b\}, \{b, c\}, \{a, b, c\}\} \) and consider the imposition of the subset relation \(\subseteq\) on \(X\). \(\{a\} \subseteq \{a, b\}\) and \(\{a, b\} \subseteq \{a, b, c\}\). But the relation is not defined on every pair of elements: \(\{c\} \not\subseteq \{a, b\}\) and \(\{a, b\} \not\subseteq \{c\}\). This possibility is what puts the “partial” in partial order. \((X, \subseteq)\) is a partially ordered set.

A least element of a partial ordering is an element \(l \in X\) such that, for every \(x \in X\), \(l \leq x\). The least element of a partial ordering, when one exists, is often called the bottom, denoted as \(\bot\). Symmetrically, a greatest element is a \(g \in X\) such that, for every \(x \in X\), \(x \leq g\). The greatest element, when one exists, is often called the top, denoted as \(\top\). It is possible for a partial ordering to have one, both, or neither of these. The partial ordering described in example 1.12 has a greatest element, \(\{a, b, c\}\), but it has no least element.

1.3.2 Total Ordering

A total ordering is a partial ordering where every pair of elements is comparable: for each \(x, y \in X\), at least one of \(x \leq y\) and \(y \leq x\) holds.

Example 1.13 The natural numbers \(\mathbb{N} = \{0, 1, 2, \ldots\}\) with the order relation \(\leq\) (less than or equal to) is a totally ordered set. For any pair of numbers in \(\mathbb{N}\), one is less than or equal to the other.

The requirement that every pair of elements be comparable does not automatically entail the existence of a least element or of a greatest element. The natural numbers \(\mathbb{N}\) have a least element, 0, but no greatest element. The integers \(\mathbb{Z}\) with \(\leq\) are a totally ordered set with neither a least element nor a greatest element. Notice that both of these totally ordered sets are infinite.

1.3.3 Lattices

Let \((X, \preceq)\) be a partially ordered set, with \(x, y \in X\). In \((X, \preceq)\), a lower bound for a pair \((x, y)\) is an element \(m \in X\) such that \(m \preceq x\) and \(m \preceq y\). An upper
1.3. ORDER

bound for \((x, y)\) is an element \(u \in X\) such that \(x \leq u\) and \(y \leq u\). Note that the lower bound for a pair is not necessarily one of the elements of the pair, but is necessarily contained in \(X\).

Recall the partially ordered set in example 1.12. For the pair of elements \(\{a, b\}, \{b, c\}\), the element \(\{a, b, c\}\) is an upper bound. The pair has no lower bound; there is no element in \(X\) that is a subset of both \(\{a, b\}\) and \(\{b, c\}\). A lower bound for the pair \(\{a\}, \{a, b\}\) is \(\{a\}\), and an upper bound is \(\{a, b\}\).

Example 1.14 Consider the natural numbers \(\mathbb{N}\) with the normal ordering \(\leq\). The pair \((5, 9)\) has a set of lower bounds \(\{0, 1, 2, 3, 4, 5\}\). It has an infinite number of upper bounds: \(\{9, 10, 11, \ldots\}\).

If the set of lower bounds for \((x, y)\) is nonempty and has a greatest element \(g\), then \(g\) is the greatest lower bound for \((x, y)\) in \((X, \leq)\). The greatest lower bound of \((x, y)\) is sometimes called the meet of \(x\) and \(y\), denoted by \(x \cap y\). If the set of upper bounds for \((x, y)\) is nonempty and has a least element \(k\), then \(k\) is the least upper bound for \((x, y)\) in \((X, \leq)\). The least upper bound of \((x, y)\) is sometimes called the join of \(x\) and \(y\), denoted by \(x \sqcup y\).

Example 1.15 For \((\mathbb{N}, \leq)\), the set of lower bounds for \((5, 9)\) is \(\{0, 1, 2, 3, 4, 5\}\), as shown in example 1.14. The greatest element of \(\{0, 1, 2, 3, 4, 5\}\) is 5, so \(5 \cap 9 = 5\). The least upper bound \(5 \sqcup 9 = 9\). The pair \((0, 6)\) has a single lower bound, 0, which is also its greatest lower bound.

If every pair of elements of \(X\) has both a meet and a join, then the ordered set is a lattice.

Example 1.16 Consider the partially ordered set \((X, \leq)\) where \(X = \{a, b, c, d\}\) and \(d \leq c \leq a\) and \(d \leq b \leq a\). This ordered set has a top, \(a\), and a bottom, \(d\). Every pair of elements has a meet and a join. The pair \((b, c)\) is not comparable with respect to the ordering, but it has a meet and a join: \(b \cap c = d\) and \(b \sqcup c = a\).

1.3.4 The Power Set Lattice

Recall that the power set of a set \(S\) is the set of all subsets of \(S\). Consider the power set of \(S = \{x, y, z\}\), as presented in example 1.2. A partially ordered set is induced by the relation “subset of”, \(\subseteq\), on \(\mathcal{P}(S)\). Observe some interesting properties of this order. The order has a least element, the empty set \(\emptyset\), and a greatest element, \(\{x, y, z\}\). This means that every pair of elements has at least one lower bound and one upper bound, because \(\emptyset\) is a lower bound for any pair, and \(\{x, y, z\}\) is an upper bound for any pair.

Consider the pair \((\{x\}, \{y\})\). It has one lower bound, \(\emptyset\), which is therefore the greatest lower bound; \(\{x\} \cap \{y\} = \emptyset\). It has two upper bounds: \(\{x, y\}\) and \(\{x, y, z\}\). \(\{x, y\} \subseteq \{x, y\}\) and \(\{x, y\} \subseteq \{x, y, z\}\), so \(\{x, y\}\) is the least element of the upper bounds, and is thus the least upper bound; \(\{x\} \sqcup \{y\} = \{x, y\}\). This establishes that the pair \((\{x\}, \{y\})\) has a meet and a join. In fact, every pair of elements in the ordered set has a meet and a join.
CHAPTER 1. MATHEMATICAL FUNDAMENTALS

Theorem 1.4 A partially ordered set induced by the relation \( \subseteq \) on a power set \( \mathcal{P}(S) \) is a lattice.

Proof. Consider an arbitrary pair \( a, b \in \mathcal{P}(S) \). We wish to prove that \( a \cap b \) exists. Consider the set \( a \cap b \). \( a \cap b \subseteq a \) and \( a \cap b \subseteq b \), so \( a \cap b \) is a lower bound on \( (a, b) \). Now, consider some arbitrary lower bound \( m \) on \( (a, b) \). Because \( m \) is a lower bound, \( m \subseteq a \) and \( m \subseteq b \). Therefore, for any \( w \in m \), \( w \in a \) and \( w \in b \). That implies that \( w \in a \cap b \). Therefore, \( m \subseteq a \cap b \). This shows that any lower bound on \( (a, b) \) is a subset of \( a \cap b \), and thus that \( a \cap b \) is the greatest lower bound on \( (a, b) \). Therefore, \( a \cap b \) is \( a \cap b \). It follows directly that \( a \cap b \) exists (this is called “proof by construction”).

The proof that \( a \cup b \) exists is left as an exercise. The existence of \( a \cap b \) and \( a \cup b \) imply that \( \mathcal{P}(S), \subseteq \) is a lattice. ■

1.3.5 Exercises

Exercise 1.3.1 Using the set \( X = \{ w, x, y, z \} \), give examples of partial orders that: (a) have both a least element and a greatest element; (b) neither a least element nor a greatest element; (c) a least element but not a greatest element.

Exercise 1.3.2 Is the relation “strictly less than”, denoted \( < \), a partial ordering on \( \mathbb{N} \)?

Exercise 1.3.3 Consider the following Optimality theoretic constraint ranking:

\[
\text{Onset} \gg \text{Max} \gg \text{NoCoda} \gg \text{Dep}
\]
1.4. MORE ABOUT ORDER

Construct a partial ordering capturing the ranking (expressed as a set of ordered pairs).

**Exercise 1.3.4** Must a finite totally ordered set have a least element and a greatest element?

**Exercise 1.3.5** Consider the following Optimality theoretic constraint ranking with tied constraints:

\[
\text{Onset} \gg \{\text{Max,Dep}\} \gg \text{NoCoda}
\]

Construct a proper partial ordering (make sure it meets all the requirements) capturing the ranking as well as possible (expressed as a set of ordered pairs). Does this partial ordering express everything that the ranking does?

**Exercise 1.3.6** Prove that, in \((\mathcal{P}(S), \subseteq)\), for arbitrary \(a, b \in \mathcal{P}(S)\), \(a \sqcup b\) exists (see Theorem 1.4).

**Exercise 1.3.7** Is a finite totally ordered set necessarily a lattice?

1.4 More About Order

1.4.1 Complete Lattices

The operations meet and join can be generalized from binary operations to operations on sets. Let \((S, \leq)\) be a partially ordered set, with \(X \subseteq S\). The meet of \(X\), \(\cap X\), is that element (if it exists) that is the greatest lower bound for \(X\). The join of \(X\), \(\sqcup X\), is the least upper bound for \(X\). Put in more basic terms:

- \(\cap X \in S\), such that
  1. \(\forall x \in X, x \leq \cap X\)
  2. \(\forall s \in S, (\forall x, s \leq x) \rightarrow (s \leq \cap X)\)

- \(\sqcup X \in S\), such that
  1. \(\forall x \in X, x \leq \sqcup X\)
  2. \(\forall s \in S, (\forall x, x \leq s) \rightarrow (\sqcup X \leq s)\)

The existence of a meet and a join for every pair of elements does not guarantee the existence of a meet and join for every subset. The natural numbers with the usual order \((\mathbb{N}, \leq)\) is a lattice, because every pair of numbers has a least upper bound and a greatest lower bound. But the even natural numbers, a subset of \(\mathbb{N}\), has no least upper bound.

A complete lattice is a lattice for which \(\cap X\) and \(\sqcup X\) exist for every nonempty subset \(X\) of \(S\).
1.4.2 Composite Orders

The Cartesian product on sets can be extended to Cartesian products on ordered sets. Given ordered sets \((X, \preceq_X)\) and \((Y, \preceq_Y)\), the Cartesian product ordered set \((X \times Y, \preceq_{X \times Y})\) is the normal Cartesian product \(X \times Y\), with the order relation \(\preceq_{X \times Y}\) defined so that \((a, b) \preceq_{X \times Y} (c, d)\) if and only if \(a \preceq_X c\) and \(b \preceq_Y d\).

**Example 1.17** Let \(S = \{a, b\}\) and \((\mathcal{P}(S), \subseteq)\) be the standard power set lattice. The Cartesian product ordered set \((\mathcal{P}(S) \times \mathcal{P}(S), \preceq)\) is an ordered set of ordered pairs of subsets. The members of the set \(\mathcal{P}(S) \times \mathcal{P}(S)\) include \((\{a\}, \{b\})\), \((\{a, b\}, \{b\})\), and \((\emptyset, \{a\})\). The ordering relations include \((\emptyset, \emptyset) \preceq (\{a\}, \emptyset) \preceq (\{a\}, \{a\}) \preceq (\{a, b\}, \{a, b\})\).

1.4.3 Exercises

**Exercise 1.4.1** Consider the set of numbers \(\{1, \frac{1}{2}, \frac{1}{4}, \ldots\}\) with the usual ordering \(\leq\). Is this partially ordered set a lattice? Is it a complete lattice?

**Exercise 1.4.2** Consider the set of numbers \(\{1, \frac{1}{2}, \frac{1}{4}, \ldots\} \cup \{0\}\) with the usual ordering \(\leq\). Is this partially ordered set a lattice? Is it a complete lattice?

**Exercise 1.4.3** Draw a picture of the composite lattice described in example 1.17.

**Exercise 1.4.4** Let \(S = \{a, b\}\) and \(T = \{b, c\}\). Consider the power sets of \(S\) and \(T\) with the usual subset orderings for each: \((\mathcal{P}(S), \subseteq_S)\) and \((\mathcal{P}(T), \subseteq_T)\). Now we want to consider the mathematical object resulting from a direct union of the carrier sets of these two partial orders, paired with a direct union of the ordering relations of these two partial orders: \((\mathcal{P}(S) \cup \mathcal{P}(T), \subseteq_S \cup \subseteq_T)\). Is this mathematical object a partial ordering? Is it a lattice?

**Exercise 1.4.5** Let \(S = \{a, b\}\) and \(T = \{b, c\}\). Consider the power sets of \(S\) and \(T\) with the usual subset orderings for each: \((\mathcal{P}(S), \subseteq_S)\) and \((\mathcal{P}(T), \subseteq_T)\). Now we want to consider the mathematical object resulting from a direct intersection of the carrier sets of these two partial orders, paired with a direct intersection of the ordering relations of these two partial orders: \((\mathcal{P}(S) \cap \mathcal{P}(T), \subseteq_S \cap \subseteq_T)\). Is this mathematical object a partial ordering? Is it a lattice?

1.5 The Semantics of Plurals

This goes with pp. 303-312 of Fred Landman, *Structures for Semantics*.

1.5.1 The Phenomena of Interest

The definite article requires that the situation of interpretation provide a unique entity satisfying its noun phrase. If the phrase is singular, the satisfying entity must be an individual. For the phrase “the student” to be defined, there must
1.5. THE SEMANTICS OF PLURALS

be a single, identifiable student in the situation (what constitutes ‘identifiable’ and ‘situation’ won’t concern us here). If the phrase is plural then the entity can consist of more than one individual; however, it must contain all individuals satisfying the noun phrase. “The students” is defined if there is at least one student in the situation, in which case it is defined as the entity containing each of the individuals in the situation that qualifies as a student. There is a notion of uniqueness carried by the definite article: in the singular it is a unique individual, in the plural it is a unique group.

Predicates are also sensitive to singularity/plurality. Interestingly, predicates can differ in how they relate to singularity/plurality.

- “The students walk.”

This implies that each student walks by themself; “walk” distributes over its arguments. It is an individual predicate.

- “The students meet.”

This does not imply that each student meets by themself; “meet” does not distribute over its arguments. It is a collective predicate.

- “The students carry tables.”

This does not imply that each student carries a table by themself, but allows for that possibility; “carry tables” may or may not distribute over its arguments, yielding different readings (interpretations). It is a mixed predicate. In the collective plural reading, “carry tables” does not distribute, and all of the students are understood to be working together to carry each table. In the distributed plural reading, “carry tables” does distribute, and each student is understood to carry a table on their own.

An interesting question arises regarding the denotation of predicates. Consider an individual predicate like “walk”. Any individual walking in the situation constitutes a singular entity that is a member of the predicate: the truth of \( \text{walk}(\{\text{bob}\}) \) implies that \( \{\text{bob}\} \in \text{walk} \). The combination of plurals with individual predicates makes it tempting to include plural entities in the denotation of the predicates. A simple analysis of “Bob and Jane are walking” might appear to be \( \text{walk}(\{\text{bob}, \text{jane}\}) \). However, this follows from the separate facts of \( \text{walk}(\{\text{bob}\}) \) and \( \text{walk}(\{\text{jane}\}) \), and it would be nice to be able to capture the systematic nature of these relationships, rather than stipulating that they coincidentally occur in all individual predicates. By contrast, for collective predicates it is quite important that the denotation of the predicate include plural entities and not include singular ones.

The challenge is to capture all of these interactions with plurality within a single coherent, compositional semantics.
1.5.2 Free Join Semilattices

A join semilattice is a partial order in which the join operation is defined over every pair of elements. You could think of it as like a lattice, but defined with only the join operator; a meet operator is not defined (which is important to this analysis).

A free join semilattice is formed by taking a set of lowest-level entities, and generating the semilattice by taking joins for each set of lowest-level entities. The free part requires that each distinct collection of lowest-level entities has a separate join: \( x \sqcup y \neq x \sqcup z \) if \( y \neq z \). It is like a full lattice without the bottom.

These properties are a good match to our intuitions about plurals. A plural entity is a set of more than one individual. A singular entity is a set of a single individual. The order relation is subset, the join operation is set union, and the lowest-level entities generating the free join semilattice are the singular entities. Thus, plural entities are sums of individuals (defined by the join operator, specifically join as set union). Sums of distinct sets of individuals are themselves distinct sums (this is the “free” part).

1.5.3 Linguistic Analysis

Types of predicates

Natural language predicates are understood to be sorted into three groups: individual (IND), collective (COL), and mixed (MIX). Individual predicates are predicates in which every member is an atom (a set containing exactly one individual). Collective predicates have no members that are atoms. With mixed predicates, some members may be atoms, but at least one is not.

For our purposes, the division is really two-way. Individual predicates are considered to be inherently singular predicates. Collective and mixed predicates are considered to be inherently plural predicates. The significance comes when we map natural language expressions into semantic expressions.

- The interpretation of a linguistically singular expression must be with a singular predicate.
- The interpretation of a linguistically plural expression must be with a plural predicate.

If a linguistically plural expression involves, on the surface, an inherently singular predicate, as in the expression “students”, something must be done to resolve the situation.

Operations on predicates: Singularization and pluralization

Singularization is the operation denoted \( \downarrow \). It creates one predicate from another (the source predicate) by selecting, from the extension of the source predicate, those entities that denote individuals (sets with only one member).
Pluralization is the operation denoted \( \uparrow \). This operation creates a predicate with an extension containing all entities of the join semilattice (with join being the set union operator) generated by the entities in the extension of the source predicate.

The operation of singularization should only be applied to a plural predicate, forming a singular predicate as a result. The operation of pluralization should only be applied to a singular predicate, forming a plural predicate.

This provides the needed resolution for predicate types. An interpretation can be provided for the linguistic expression “students” by applying pluralization to the inherently singular predicate \( \text{student} \), yielding an appropriate plural predicate \( \uparrow \text{student} \).

The possible singular and plural readings

The trick used to obtain the possible readings is the following (p. 306-308 of Landman):

1. Grant, as possible readings of a predicate, both the predicate unaltered, and all alternating compositional applications of singularization and pluralization to the predicate (singularization is only applied to plural predicates, and pluralization is only applied to singular predicates).

   (a) A reading that is to be combined with an argument that is singular must either
       i. be an inherently singular predicate;
       ii. have had singularization applied to it last (after any other singularizations and pluralizations).

   (b) A reading that is to be combined with an argument that is plural must either
       i. be an inherently plural predicate;
       ii. have had pluralization applied to it last (after any other singularizations and pluralizations).

2. Sounds like a lot of different possible readings, right? Wrong.

   (a) If the predicate is individual, then each entity of the predicate is atomic (a single individual). The predicate is inherently singular. The pluralization is the join semilattice generated by those individuals. The singularization of the pluralization gives you just the individuals, which is equivalent to the original predicate again. Any further singularization and/or pluralization just gives you one of these two. Thus, for an individual predicate \( I \):
       i. The possible readings for singular are: \( \{ I \} \)
       ii. The possible readings for plural are: \( \uparrow I \)
(b) If the predicate is collective, then each entity of the predicate is not atomic (each contains more than one individual). The predicate is inherently plural. The singularization is the empty set, which is equivalent to no reading at all. The pluralization of the empty set is empty, as is any further singularization and/or pluralization. Thus, for a collective predicate $C$:
  i. The possible readings for singular are: $\emptyset$
  ii. The possible readings for plural are: $\{C\}$

(c) If the predicate is mixed, then some entities are atomic and others aren’t. The predicate is inherently plural. The singularization of the predicate picks out just those entities that are atomic. The pluralization of the singularization then gives something like a distributed reading for those entities that appear as atoms in the original predicate. This possibility adds a second, distributed plural reading for mixed predicates. Any further singularization and pluralization just alternates back and forth between just the atoms, and the pluralization of the atoms. Thus, for a mixed predicate $M$:
  i. The possible readings for singular are: $\{^1M\}$
  ii. The possible readings for plural are: $\{M,^1^1M\}$

  $M$ is the collective plural reading, $^1^1M$ is the distributed plural reading.

The definite article

The definite article is applied to a predicate $P$ by applying a special version of set abstraction to the predicate:

$$\sigma x.P(x)$$

This receives a conditional interpretation $[\sigma x.P(x)]$:

- $\sqcup([P])$ if $\sqcup([P]) \in [P]$; if the join of all members in the extension of the predicate is itself in the extension of the predicate, then the interpretation is the join of all the members;
- $*$ otherwise: if the join of all the members is not in the extension of the predicate, then the form is undefined.

In other words, the definite article is only defined if the predicate it is applied to contains a maximal element $\top = \sqcup([P])$.

1.5.4 Pulling it all together: An illustration

The situation (as we see it)

$[\text{semanticist}] = \{\{Veneeta\}, \{Roger\}, \{Maria\}\}$
1.5. THE SEMANTICS OF PLURALS

\[
\begin{align*}
[\text{woman}] &= \{\{\text{Venecta}\}, \{\text{Maria}\}\} \\
[\text{chair}] &= \{\{\text{Akin}\}\} \\
[\text{walk}] &= \{\{\text{Maria}\}, \{\text{Roger}\}, \{\text{Venecta}\}\} \\
[\text{meet}] &= \{\{\text{Venecta, Maria}\}, \{\text{Roger, Akin}\}\} \\
[\text{eat}] &= \{\{\text{Maria}\}, \{\text{Akin}\}\}
\end{align*}
\]

Singular Individual Predicates

“The chair is eating.”

\[
eat(\sigma x.\text{chair}(x))
\]

1. The singular form of the predicate chair is taken:

\[
[\text{chair}] = \{\{\text{Akin}\}\}
\]

2. The definite article is applied:

\[
[\sigma x.\text{chair}(x)] = \{\text{Akin}\}
\]

This is defined, because the join of all the members of the predicate,

\[
\sqcup[\text{chair}] = \{\text{Akin}\}
\]

is a member of the predicate chair itself.

3. The singular reading of the predicate eat is taken:

\[
[\text{eat}] = \{\{\text{Maria}\}, \{\text{Akin}\}\}
\]

4. The two predicates are composed:

\[
[[\text{eat}(\sigma x.\text{chair}(x))] = [\sigma x.\text{chair}(x)] \in [\text{eat}] = \text{true}
\]

“The semanticist is walking.”

\[
\text{walk}(\sigma x.\text{semanticist}(x))
\]

1. The singular form of the predicate semanticist is taken:

\[
[\text{semanticist}] = \{\{\text{Veneeta}\}, \{\text{Roger}\}, \{\text{Maria}\}\}
\]

2. The definite article is applied:

\[
[\sigma x.\text{semanticist}(x)] = *
\]

This is undefined, because the join of all the members of the predicate,

\[
\sqcup[\text{semanticist}] = \{\text{Veneeta, Roger, Maria}\}
\]

is not a member of the predicate semanticist itself. In other words, the single definite plural expression “The semanticist” cannot be satisfied, because there is not a unique semanticist in the situation.
Plural Individual Predicates

“The semanticists are walking.”

\[ \updownarrow \text{walk} (\sigma x. \updownarrow \text{semanticist} (x)) \]

1. The plural form of the predicate \( \text{semanticist} \) is taken:

\[
\begin{align*}
\updownarrow \text{semanticist} & = \{ \{ V \}, \{ R \}, \{ M \}, \{ V, R \}, \{ V, M \}, \{ R, M \}, \{ V, R, M \} \} \\
\end{align*}
\]

2. The definite article is applied:

\[
\begin{align*}
\llbracket \sigma x. \updownarrow \text{semanticist} (x) \rrbracket & = \{ \text{Venecta, Roger, Maria} \}
\end{align*}
\]

3. The plural reading of the predicate \( \text{walk} \) is taken:

\[
\begin{align*}
\updownarrow \text{walk} & = \{ \{ V \}, \{ R \}, \{ M \}, \{ V, R \}, \{ V, M \}, \{ R, M \}, \{ V, R, M \} \}
\end{align*}
\]

4. The two predicates are composed:

\[
\begin{align*}
\llbracket \updownarrow \text{walk} (\sigma x. \updownarrow \text{semanticist} (x)) \rrbracket & = \text{true}
\end{align*}
\]

5. Because \( \text{walk} \) is an individual predicate, the truth of “the semanticists are walking” implies the the truth of “a semanticist is walking”, because the truth of \( \updownarrow \text{walk} (\{ \text{Venecta, Roger, Maria} \}) \) implies the truth of \( \updownarrow \text{walk} (\{ \text{Venecta} \}) \).

Singular Collective Predicates

“The chair is meeting.”

\[ \downarrow \text{meet} (\sigma x. \downarrow \text{chair} (x)) \]

1. The singular form of the predicate \( \text{chair} \) is taken:

\[
\begin{align*}
\llbracket \text{chair} \rrbracket & = \{ \{ \text{Akin} \} \}
\end{align*}
\]

2. The definite article is applied:

\[
\begin{align*}
\llbracket \sigma x. \text{chair} (x) \rrbracket & = \{ \text{Akin} \}
\end{align*}
\]

3. The singular form of the predicate \( \text{meet} \) is taken:

\[
\begin{align*}
\llbracket \downarrow \text{meet} \rrbracket & = \emptyset
\end{align*}
\]

This is the empty set, because there are no atoms in the extension of \( \text{meet} \).
Plural Collective Predicates

“The women are meeting.”

\( \text{meet} (\sigma x. \downarrow \text{woman}(x)) \)

1. The plural form of the predicate \( \text{woman} \) is taken:

\[ [\downarrow \text{woman}] = \{\{\text{Veneeta}\}, \{\text{Maria}\}, \{\text{Veneeta, Maria}\}\} \]

2. The definite article is applied:

\[ [\sigma x. \downarrow \text{woman}(x)] = \{\text{Veneeta, Maria}\} \]

3. The collective plural reading of the predicate \( \text{meet} \) is taken:

\[ \llbracket \text{meet} \rrbracket = \{\{\text{Veneeta, Maria}\}, \{\text{Akin, Roger}\}\} \]

4. The two are composed, yielding a satisfiable expression:

\[ [\text{meet} (\sigma x. \downarrow \text{woman}(x))] = \text{true} \]

5. You can attempt the distributive plural reading of \( \text{meet} \), but the result is empty:

\[ [\downarrow \uparrow \text{meet}] = \emptyset \]

Thus, there is no distributive reading available.

1.5.5 Exercises

Exercise 1.5.1 Using the situation provided in section 1.5.4, give the interpretation of the sentence “the women are eating.”

Exercise 1.5.2 Using the situation provided in section 1.5.4, give the interpretation of the sentence “the chairs are eating.”
Chapter 2

Cardinality

Cardinality is the mathematical study of size. Much of the basic mathematical theory of infinite sets was originally developed by Georg Cantor. The work was conducted over many years, but most of the results were first published between 1895 and 1897.

2.1 Finite Combinatorics

Combinatorics is the study of counting. In particular, it studies the sizes of mathematical objects, especially relationships between the sizes of objects and the mathematical operations used to define them. This section presents a few of the best known (and most useful) combinatoric principles for finite sets.

2.1.1 The Sum Rule

If two sets $A$ and $B$ have $|A|$ and $|B|$ elements, respectively, then $A \cup B$ has $|A| + |B| - |A \cap B|$ elements.

2.1.2 The Product Rule

If two sets $A$ and $B$ have $|A|$ and $|B|$ elements, respectively, then there are $|A| \cdot |B|$ independent pairs of elements consisting of one element from each set.

An $n$-way Cartesian product of a set $S$ contains $|S|^n$ elements (ordered $n$-tuples).

2.1.3 Permutations

A permutation of a set of $n$ elements is an arrangement of the elements in the set into some total order. A set of $n$ elements has $n!$ permutations.

The expression $n!$ is pronounced “$n$ factorial”, and is the product of the integers $1 \ldots n$. By convention, $0! = 1$. 

23
CHAPTER 2. CARDINALITY

2.1.4 Combinations

A combination of \( n \) elements taken \( r \) at a time is an \( r \)-element subset of elements taken from an \( n \)-element set. The number of distinct combinations of \( n \) elements taken \( r \) at a time is denoted by \( \binom{n}{r} \).

Theorem 2.1 Let \( n, r \in \mathbb{N} \) and \( r \leq n \). \( \binom{n}{r} = \frac{n!}{r!(n-r)!} \)

Proof. The \( n \) elements have \( n! \) permutations. The set of permutations can also be arrived at in the following way. First, select \( r \) elements, which can be done \( \binom{n}{r} \) ways (by definition). For each such selection, there are \( r! \) permutations of those \( r \) elements, and \( (n-r)! \) permutations of the remaining elements. Thus, the number of permutations of the \( n \) elements is \( n! = \binom{n}{r} \cdot r! \cdot (n-r)! \). It then follows that \( \binom{n}{r} = \frac{n!}{r!(n-r)!} \).

Observe that there are two ways to count all the possible subsets of a set of \( n \) elements. One is to add the number of 0-element subsets \( \binom{n}{0} \) with the number of 1-element subsets \( \binom{n}{1} \) and so on up to the number of \( n \)-element subsets \( \binom{n}{n} \), giving \( \sum_{r=0}^{n} \binom{n}{r} \). The other is to count all subsets by whether the first person is in the subset, whether the second person is in the subset, and so on, yielding \( 2 \cdot 2 \cdot \ldots \cdot 2 = 2^n \). It follows that \( \sum_{r=0}^{n} \binom{n}{r} = 2^n \).

2.1.5 The Binomial Theorem

Consider taking a simple sum of two values, and then raising that sum to successively higher exponents.

\[
(a + b)^1 = a + b \\
(a + b)^2 = a^2 + 2ab + b^2 \\
(a + b)^3 = a^3 + 3a^2b + 3ab^2 + b^3 \\
(a + b)^n = a^n + na^{n-1}b + \ldots + nab^{n-1} + b^n
\]

As the exponent grows, so does the number of terms in the expansion; in fact, the number of terms is always one more than the exponent. The makeup of each term with respect to the exponents of the two values \( a \) and \( b \) is rather obvious. What is not quite as obvious is the pattern to the coefficients of the terms.

The pattern to the coefficients is a famous result, and ties directly to the “\( n \) choose \( r \)” combinatorial concept. There are two standard ways to view it. One way is algebraic, and expresses the result commonly known as the Binomial Theorem:

\[
(a + b)^n = \sum_{r=0}^{n} \binom{n}{r} a^r b^{n-r}
\]
2.1. FINITE COMBINATORICS

The other way to view the pattern is diagrammatic, and is commonly known as Pascal’s Triangle. This shown pictorially in figure 2.1. Each value in the triangle is a value of $\binom{n}{r}$ for some value of $n$ and $r$. It capitalize on the following fact:

$$
\binom{n}{r} = \binom{n - 1}{r - 1} + \binom{n - 1}{r}
$$

for $n > 0$

In the figure, the rows correspond to values of $n$, with the top row being $n = 0$, the next row down $n = 1$, and so forth. In each row, the values across correspond to values of $r$ from 0 to $n$ for that row. Thus, the 2 in the middle of the third row down is the value of $\binom{3}{2}$. The first and last members of each row are always 1; $\binom{n}{0}$ and $\binom{n}{n}$ are always 1, for any value of $n$.

2.1.6 Exercises

For each of the following, assume that there is a vocabulary $V$ of seven words.

**Exercise 2.1.1** How many distinct sentences can be formed from the words in $V$, where each word appears exactly once in any given sentence?

**Exercise 2.1.2** How many distinct sets of three words can be formed from the words in $V$?

**Exercise 2.1.3** How many distinct sentences of length 3 (i.e., have three words) can be formed from $V$, if no word appears more than once in any given sentence?

**Exercise 2.1.4** How many distinct sentences of length 3 (i.e., have three words) can be formed from $V$, if a word can appear more than once in a sentence?
Exercise 2.1.5 *How many distinct sentences with length in the range 0...7 can be formed from V, if no word appears more than once in any given sentence?*

2.2 Countable Sets

There are two intuitions that people frequently have about cardinality. The first is that if $A \subset B$, then $|A| < |B|$. The second is that infinity is a single size: all infinite sets should be the same size. Clearly, these two conditions cannot simultaneously be true. What we are about to discover is that, for infinite sets, neither one is true.

First, a bit of notation: $\mathbb{N} = \{0, 1, 2, \ldots \}$ denotes the natural numbers, while $\mathbb{N}^+ = \{1, 2, 3, \ldots \}$ denotes the positive natural numbers. $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots \}$ denotes the integers. $\mathbb{R}$ denotes the set of all real numbers.

2.2.1 Equivalence

Two sets $A$ and $B$ are equivalent if and only if there exists a bijection between $A$ and $B$. Equivalent sets are often said to be “the same size”. Equivalence is a reflexive, symmetric, transitive relation, denoted as $A \approx B$. Note the distinction between equivalence and equality. If $A = \{1, 2, 3\}$ and $B = \{a, b, c\}$, then $A \approx B$ but $A \neq B$.

Let $\mathbb{N}^+_k$ denote the set $\{1, 2, 3, \ldots, k\}$. A set $S$ is finite if and only if one of the following is true:

- $S = \emptyset$
- $S \approx \mathbb{N}^+_k$ for some $k \in \mathbb{N}^+$

If $S = \emptyset$, $S$ is said to have size 0, denoted as $|S| = 0$. If $S \approx \mathbb{N}^+_k$ for some $k \in \mathbb{N}^+$, then $S$ is said to have size $|S| = k$.

A set $S$ is denumerable if $S \approx \mathbb{N}^+$. A set $S$ is countable if it is either finite or denumerable.

2.2.2 The Integers

**Theorem 2.2** $\mathbb{Z}$ is denumerable.

**Proof.** Define $f : \mathbb{N}^+ \rightarrow \mathbb{Z}$ by

$$f(x) = \begin{cases} \frac{x}{2} & \text{if } x \text{ is even} \\ \frac{x+1}{2} & \text{if } x \text{ is odd} \end{cases}$$

This function, pictorially:

$$\begin{align*} \mathbb{N}^+ &= \{ 1, 2, 3, 4, 5, 6, 7, \ldots \} \\
       &\downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \\
\mathbb{Z} &= \{ 0, 1, -1, 2, -2, 3, -3, \ldots \} \end{align*}$$
2.2. COUNTABLE SETS

This function is a bijection between $\mathbb{N}^+$ and $\mathbb{Z}$. ■

This result may seem a bit counterintuitive at first. After all, $\mathbb{N}^+$ is quite clearly a proper subset of $\mathbb{Z}$, yet they are the same size.

2.2.3 The Rational Numbers

A rational number is a number that is equal to the ratio of two integers, e.g., $\frac{p}{q}$, $p, q \in \mathbb{Z}$. The set of all rational numbers is denoted as $\mathbb{Q}$.

The rationals are interesting to us because of a property they have relative to the normal ordering of them, $\leq$. The rationals are dense in the normal number line: between any two rationals, there is another rational number. In fact, that immediately implies that between any two rational numbers there are an infinite number of others.

**Theorem 2.3** The rationals are dense in the number line.

**Proof.** Consider two distinct rational numbers, $\frac{p}{q} < \frac{r}{s}$. We wish to prove that there is another rational number greater than $\frac{p}{q}$ and less than $\frac{r}{s}$.

Consider the number $\frac{p}{q} + \frac{1}{2} \cdot \left( \frac{r}{s} - \frac{p}{q} \right)$. It is clearly a rational number, because the rationals are closed under addition and multiplication. Because $\frac{p}{q} < \frac{r}{s}$, it follows that $0 < \left( \frac{r}{s} - \frac{p}{q} \right)$, and therefore that $0 < \frac{1}{2} \cdot \left( \frac{r}{s} - \frac{p}{q} \right)$. Therefore,

$$\frac{p}{q} < \frac{p}{q} + \frac{1}{2} \cdot \left( \frac{r}{s} - \frac{p}{q} \right)$$

It should also be clear that

$$\frac{p}{q} + \frac{1}{2} \cdot \left( \frac{r}{s} - \frac{p}{q} \right) < \frac{p}{q} + \left( \frac{r}{s} - \frac{p}{q} \right) = \frac{r}{s}$$

Thus, $\frac{p}{q} + \frac{1}{2} \cdot \left( \frac{r}{s} - \frac{p}{q} \right)$ is a distinct rational in between $\frac{p}{q}$ and $\frac{r}{s}$. ■

The really remarkable thing is that, despite this density, the rationals are nevertheless countable.

**Theorem 2.4** $\mathbb{Q}$ is denumerable.

2.2.4 Exercises

**Exercise 2.2.1** Show that $S \approx S \times \{x\}$.

**Exercise 2.2.2** Prove that the odd natural numbers $\{1, 3, 5, \ldots\}$ are denumerable.

**Exercise 2.2.3** Show that if $S$ is denumerable, then $S \approx S \cup \{x\}$.
2.3 Uncountable Sets

2.3.1 The Real Numbers

The real numbers are representable as a denumerable sequence of digits, with one decimal place.

**Theorem 2.5** The open interval $(0, 1)$ is uncountable.

**Proof.** For convenience and simplicity, consider each real number in $(0, 1)$ represented in decimal digits, i.e., $\frac{1}{2} = 0.5\overline{3}$. Suppose there is a function $f : \mathbb{N}^+ \rightarrow (0, 1)$ that is an injection. We will show that it is not a surjection (not onto), and therefore is not a bijection. Write each pair of the mapping in the following way:

\[
\begin{align*}
  f(1) &= 0.a_{11}a_{12}a_{13}a_{14}... \\
  f(2) &= 0.a_{21}a_{22}a_{23}a_{24}... \\
  f(3) &= 0.a_{31}a_{32}a_{33}a_{34}... \\
  f(4) &= 0.a_{41}a_{42}a_{43}a_{44}... \\
  \vdots \\
  f(n) &= 0.a_{n1}a_{n2}a_{n3}a_{n4}... \\
  \vdots 
\end{align*}
\]

Now let $x$ be the number $x = 0.x_1x_2x_3x_4\ldots$, where

\[
x_i = \begin{cases} 
  3 & \text{if } a_{ii} \neq 3 \\
  7 & \text{if } a_{ii} = 3
\end{cases}
\]

Then $x \neq f(n)$ for any $n \in \mathbb{N}^+$, because it will differ from $f(n)$ on the $n$th digit. However, $x$ is a real number in $(0, 1)$, so $f$ is not onto. Therefore, no bijection between $\mathbb{N}^+$ and $(0, 1)$ exists, so $(0, 1)$ is not countable. $\blacksquare$

This kind of proof is called a *diagonalization argument*.

We have now demonstrated that not all infinite sets are of the same size; the real numbers is a bigger set than the natural numbers.

2.3.2 The Unboundedness of Cardinality

**Theorem 2.6** For every set $S$, $|S| < |\mathcal{P}(S)|$.

**Proof.** First, observe that $f(x) = \{x\}$ is a mapping from $S$ to $\mathcal{P}(S)$ that is an injection, so $|S| \leq |\mathcal{P}(S)|$.

The rest is a proof by contradiction. Suppose, to the contrary, that there exists a bijection $g : S \rightarrow \mathcal{P}(S)$. Let $X = \{x \in S | x \notin g(x)\}$. Since $X \subseteq S$, $X \in \mathcal{P}(S)$, and since $g$ is a surjection onto $\mathcal{P}(S)$ (follows from being a bijection), there exists $z \in S$ such that $g(z) = X$. Now, either $z \in X$ or $z \notin X$. If $z \in X$, then $z \notin g(z) = X$, a contradiction. But if $z \notin X$, then $z \in g(z) = X$, also a contradiction. Therefore, no such bijection $g$ exists, so $|S| < |\mathcal{P}(S)|$. $\blacksquare$
2.3.3 Exercises

Exercise 2.3.1 Is the power set of the set of natural numbers, $\mathcal{P}(\mathbb{N})$, denumerable?

Exercise 2.3.2 Prove, using a diagonalization argument, that the open interval $(4,5)$ on the real numbers is uncountable.

Exercise 2.3.3 Is the set $\mathbb{R}$ of all real numbers uncountable? Why or why not?
Chapter 3

Computation

3.1 Formal Languages

3.1.1 Strings

An alphabet is a finite set of symbols (non-divisible objects).

A string is a sequence of symbols from some alphabet. The length of a string is the number of symbols in the sequence. The empty string, denoted $\epsilon$, is the string containing precisely no symbols, and has length 0.

The fundamental operation on strings is concatenation. The concatenation of two strings $s$ and $t$ is the string resulting from putting $t$ immediately after $s$, sometimes denoted $s \cdot t$ or $st$. The concatenation of “b c b” and “a a” is “b c b a a”.

The notation (“a”)^4 denotes a sequence of concatenations of four occurrences of the string “a”, in other words “a”·“a”·“a”·“a”, resulting in the string “a a a a”. The notation (“b c”)^3 denotes the string “b c b c b c”. The string of n consecutive occurrences of a string $q$ is denoted $q^n$. $q^0 = \epsilon$.

We can extend the notation for concatenation to apply to sets of strings. The concatenation of two sets of strings is a set of strings, each of which results from a concatenation of a string from the first set with a string from the second set. {“a”, “b”}·{“c”, “d”} = {“a c”, “a d”, “b c”, “b d”}.

The exponent notation can also apply to a set of strings, resulting in a set of strings each formed by concatenation. {“a”, “b”}^2 denotes a set of strings: {“a a”, “b b”, “a b”, “b a”}. This is the set of strings that can be formed by taking any two strings from the set (including selecting the same string twice), in any order, and concatenating them together.

Example 3.1 Let $A = \{a, b, c\}$ be the alphabet. Strings defined on that alphabet include: “a”, “b c b c a b”, (“c”)^5, and $\epsilon$.

Another kind of string operation is reversal. Denoted with a superscript R, reversal reverses the order of the string: (“a b c”)^R = “c b a”.

31
3.1.2 Languages

A language is a set of strings on some alphabet.

General operations on sets, such as union and intersection, are also operations on languages.

The set of strings obtained by concatenating together zero or more instances of a string \( s \) is denoted \( \{s\}^* \). This is called the Kleene star closure operation. It is defined more generally as an operation on a language (a set of strings), giving the set of all strings formable by concatenating an arbitrary number of strings in the language.

In a slight abuse of notation, the set of all possible strings that can be formed on an alphabet \( A \) is commonly denoted \( A^* \) (technically speaking, the Kleene star should be applied to a set containing one unitary string for each symbol of the alphabet). Every possible language on \( A \) is a subset of \( A^* \).

**Example 3.2** The language \( \{“a”\}^* = \{\epsilon, “a”, “a a”, “a a a”, … \} \).

**Example 3.3** The language \( \{“a”, “c b”\}^* = \{\epsilon, “a”, “c b”, “a a”, “a c b”, “c b c b”, “c b a”, … \} \).

A regular expression on an alphabet \( A \) is an expression formable from the symbols of the alphabet, plus balanced parentheses, the union symbol \( \cup \), the intersection symbol \( \cap \), the Kleene star, the empty string symbol and the empty set symbol.

There is a variation of Kleene star that removes the empty string possibility. \( \{s\}^+ \) denotes the set of strings obtained by concatenating together one or more instances of the string \( s \).

The empty set is a possible formal language, sometimes called the empty language; it is the language with precisely no strings. Do not confuse the empty language with the empty string; the former is a language, the latter is a string. A language containing only the empty string, \( \{\epsilon\} \), is not empty, it is a language containing precisely one string (which happens to be the empty string).

3.1.3 Exercises

**Exercise 3.1.1** Let \( A = \{“a”\} \) and \( B = \{“b”\} \). Describe the following languages informally.

- \( B^* \)
- \( (A \cup B)^* \)
- \( (A \cap B)^* \)
- \( A \cdot (B^*) \)
- \( \{\epsilon\}^3 \)
- \( \emptyset^3 \)
Exercise 3.1.2 Is concatenation of strings commutative?

Exercise 3.1.3 Let $A = \{a, b\}$. List all strings of length 2 or less than can be formed on this alphabet.

Exercise 3.1.4 Let $A = \{a, b\}$. How many distinct strings of length 5 can be formed on this alphabet?

Exercise 3.1.5 Let $A = \{a, b\}$. What is the cardinality of $A^*$?

Exercise 3.1.6 Let $A = \{a, b\}$. What is the cardinality of the set of all possible languages on $A$?

Exercise 3.1.7 Describe the language defined by the regular expression 
\[
\left(\{a\}^* \cap \{a\} \cdot \{ab\}^*\right) \cup \{c^n \cdot \{b^n\}^2\}.
\]

3.2 Formal Rewrite Grammars

A formal rewrite grammar is a mathematical object $G = (Q, T, P, \Sigma)$ where

- $Q$ is a finite set of non-terminals (the categories of the language)
- $T$ is a finite set of terminals (the alphabet of the language)
- $P$ is a finite set of productions (also referred to as rewrite rules)
- $\Sigma$ is the start symbol

The sets of non-terminals and terminals must be disjoint, that is, $Q \cap T = \emptyset$.

Grammars are used to define formal languages. The terminal symbols of the grammar are precisely the alphabet for the strings in the language defined by the grammar. A step-by-step application of the rules of a grammar, resulting in a particular string, is called a derivation. The strings in the language of a grammar are those strings for which there exists a derivation using the rules of that grammar. Some example derivations for strings using the grammar of example 3.4 are given in example 3.6 and example 3.7.

Example 3.4 The language generated by the regular expression $\{a\}^+ \cdot \{b^n\}^+$ (also expressible as $\{(a)^m \cdot (b)^n \mid m, n > 0\}$) is generated by the grammar with $Q = \{A, B\}$, $T = \{a, b\}$, and the productions

- $\Sigma \rightarrow a A$
- $\Sigma \rightarrow a B$
- $A \rightarrow a A$
- $A \rightarrow a B$
- $B \rightarrow b B$
- $B \rightarrow b$
To save space, productions with the same left-hand side are sometimes grouped together on one line, with the different right-hand sides linked with "|" operators.

**Example 3.5** The productions of the grammar of example 3.4 in abbreviated form:

\[
\begin{align*}
\Sigma & \rightarrow aA \mid aB \\
A & \rightarrow aA \mid aB \\
B & \rightarrow bB \mid b
\end{align*}
\]

**Example 3.6** A derivation for the string “a a b”:

\[
\begin{align*}
\Sigma & \\
aA & \text{(using rule } \Sigma \rightarrow aA) \\
aaB & \text{(using rule } A \rightarrow aB) \\
aab & \text{(using rule } B \rightarrow b)
\end{align*}
\]

**Example 3.7** A derivation for the string “a a a b b”:

\[
\begin{align*}
\Sigma & \\
aA & \text{(using rule } \Sigma \rightarrow aA) \\
aaA & \text{(using rule } A \rightarrow aA) \\
aaB & \text{(using rule } A \rightarrow aB) \\
aaBB & \text{(using rule } B \rightarrow bB) \\
aaab & \text{(using rule } B \rightarrow b)
\end{align*}
\]

Every derivation starts with the start symbol \( \Sigma \). Each step of a derivation involves the application of a production. A production allows for a non-terminal, indicated on the left-hand side of the production, to be replaced, or rewritten, with a string of terminals and non-terminals, indicated on the right-hand side of the rule. Any production indicating rewriting of the start symbol can be chosen for the first step of a derivation. Different strings are derived by choosing different productions for rewriting the same symbol; if there is only one production in a grammar for rewriting the start symbol and for each non-terminal, then the grammar can generate at most one string.

A derivation is, effectively, a sequence of strings, with each string consisting of terminals and non-terminals (we can think of the start symbol as like a non-terminal). A proper derivation always ends in a string consisting only of terminal symbols; such a string is generated by the grammar. Note that a derivation cannot continue once a string of all terminal symbols is reached, because productions can only replace non-terminals, not terminals. The first string in a derivation is always the start symbol by itself. Each subsequent string in a derivation is the result of applying a production of the grammar to the preceding string in the derivation.

Strings with non-terminals in them are not part of the language generated by the grammar. Non-terminals are only used in the process of deriving strings of terminals. The non-terminals help to characterize, or describe, the strings of the language. This is perhaps more easily seen with another perspective on derivations. The derivation of a string can be represented with a tree, in which each
3.2. FORMAL REWRITE GRAMMARS

production application is represented by a parent node and its child nodes. The derivation tree for the step-by-step derivation of example 3.7 is given in figure 3.1. The parent node represents the non-terminal being rewritten (the left-hand side of the production), while the child nodes represent the symbols of the string that replaces (the right-hand side of the production). Every derivational tree has the start symbol at the root, and terminal symbols at the leaves.

The empty string can be generated by a rewrite grammar by including the production \( \Sigma \rightarrow \epsilon \). Grammars which include that production generate the empty string.

The word “finite” appears frequently in the definition of a formal rewrite grammar. This is not a minor fact; one of the major motivations for this definition of formal grammars is that the grammars are finite mathematical objects. Notice that it is the grammars, and not necessarily the generated languages, that are finite. For instance, the grammar in example 3.4 generates an infinite language. The grammar itself is finite. This finiteness condition will be the theme throughout this chapter. One motivation is an interest in the formal properties of finite descriptions of infinite sets. What are the formal consequences of restricting our attention to formal languages allowing finite descriptions? Another motivation is psychological. If we believe that the grammar of a natural language is represented inside the mind of a native speaker, and we agree that human minds are finite objects, then it follows that the grammars of natural languages must be finite.

3.2.1 Exercises

Exercise 3.2.1 Using the grammar in example 3.4, give the step-by-step derivation for the string “a b b”.

Exercise 3.2.2 For the grammar in example 3.4, which productions can possi-
bly be last production applied in a derivation? Explain why.

3.3 Regular Grammars

3.3.1 Definitions

Different classes of grammars are distinguished by the particular forms of production rules that they allow. In a regular grammar, every production has one of the following forms, where \( A \in Q \cup \{\Sigma\} \), \( B \in Q \), \( t \in T \):

- \( A \rightarrow tB \) (right-linear form)
- \( A \rightarrow t \)
- \( \Sigma \rightarrow \epsilon \)

An alternative, equivalent version is to have rules of the left-linear form, \( A \rightarrow Bt \), in place of rules of right-linear form. However, any particular regular grammar can only have rules of one of those types; mixing the two in a single grammar can produce non-regular grammars.

The formal languages described by regular expressions can also be described by regular grammars.

**Example 3.8** The language for the regular expression \( \{a\}^* \cdot \{b\}^* \) is generated by the grammar with \( Q = \{A, B\} \), \( T = \{a, b\} \), and the productions

\[
\begin{align*}
\Sigma & \rightarrow \ a\ A \mid a\ B \mid b\ B \mid a \mid b \mid \epsilon \\
A & \rightarrow \ a\ A \mid a\ B \mid a \\
B & \rightarrow \ b\ B \mid b
\end{align*}
\]

The significance of the term "right-linear form" can be seen by looking at a derivation tree for a right-linear regular grammar, such as the derivation tree in figure 3.1. Observe that the tree gets deeper along the right edge. This is because the right-linear form always puts the non-terminal on the right side. The effect can also be seen in the step-by-step derivations, such as the one in example 3.7. Notice that at each step (except for the final string), there is only one non-terminal, and it is the rightmost symbol in the string. Further, the string is derived left-to-right, one symbol at each step. This is the "linear" in right-linear form.

3.3.2 Regular Languages

A formal language is said to be a regular language if it is generated by at least one regular grammar. Two regular languages of interest are the language containing all possible strings on an alphabet, which we can denote \( A^* \), and the empty language, containing no strings. The empty language is trivially generated by a grammar with no productions.
3.3. REGULAR GRAMMATERS

Example 3.9 For the alphabet $A = \{a, b\}$, the language containing all possible strings on the alphabet is generated with the following productions:

$$
\Sigma \rightarrow aX \mid bX \mid a \mid b \mid \epsilon
$$

$$
X \rightarrow aX \mid bX \mid a \mid b
$$

The grammar generating all possible strings on an alphabet is remarkably simple; it only requires one non-terminal. This reveals that the complexity of a language, in terms of the size of the grammar needed to generate it, does not correlate at all with the size of the language itself. We will demonstrate below that there are languages that cannot be generated by any regular grammar, but that is not because the languages contain strings that cannot be generated by any regular grammar. Any string can be generated by some formal grammar. The complexity of languages resides as much in the strings that are not in the languages as in the strings that are in the language.

Any finite language is a regular language. This is a consequence of the fact that a finite language can be represented by simply listing the strings of the language; because the language is finite, the list of the strings is finite, thus constituting a finite description of the language.

Example 3.10 The finite language \{“a”, “b a”, “b b c”, “a c a b”\} is regular, and generated by the grammar

$$
\Sigma \rightarrow a \mid b A2 \mid b A3 \mid a A4
$$

$$
A2 \rightarrow a
$$

$$
A3 \rightarrow b B3
$$

$$
B3 \rightarrow c
$$

$$
A4 \rightarrow c B4
$$

$$
B4 \rightarrow a C4
$$

$$
C4 \rightarrow b
$$

The recipe for constructing a regular grammar for a finite list is fairly simple. The key is to construct a separate set of productions for each string, using different non-terminals for each, then combine the productions together. In example 3.10, string 1, “a”, only needs the production $\Sigma \rightarrow a$. String 2, “b a”, has length 2, and so needs one additional non-terminal, A2.

Of course, it will often be possible to generate a given finite language with a regular grammar that doesn’t follow this recipe. But this recipe is always a possibility for any finite language. Thus, formal languages that are in principle not regular languages must necessarily be infinite. It is not the case that all infinite languages are not regular: the language containing all possible strings on a (non-empty) alphabet is infinite and regular, as are many other languages. But the formal languages that escape the descriptive power of regular grammars are infinite languages containing some possible strings but not others; it is the complexity of the “some but not others” that eludes the regular grammars.

Example 3.10 reveals a general recipe for encoding fixed strings in a regular grammar. Assign a distinct non-terminal to all but the last symbol of the
string; for instance, the string “a c a b” is given the sequence of non-terminals [A4,B4,C4]. Each non-terminal represents how much of the string has been produced thus far in the derivation, and thus what remains to be produced: the non-terminal B4 indicates that “a c” has been produced thus far, and that “a b” remains to be produced.

Example 3.10 also reveals how, given a regular language which is the union of other regular languages, a regular grammar for the union language can be constructed from the grammars for the other languages. To combine two factor regular grammars so as to generate the union, first change, in one of the factor grammars, any non-terminals that are identical to non-terminals in the first. The result should be two factor grammars that have no non-terminals in common. Then, take the union of each of the components: the resulting set of terminals is the union of the factor sets of terminals, likewise for the non-terminals and the productions. One consequence of this recipe is that the union of any pair of regular languages is necessarily a regular language.

3.3.3 Recursion

A derivation is recursive if a production is used, and then the same production is subsequently used in the derivation rewriting the non-terminals produced by the initial use of the production. A grammar is recursive if it allows recursive derivations. The intuition behind recursion is that things are defined “in terms of themselves.” This intuition is a very important part of linguistic theory. For instance, a verb phrase can be understood as being defined in terms of itself, in that a verb phrase can contain a noun phrase, which in turn contains a relative clause, which in turn contains a verb phrase which is subject to the same definition as the containing verb phrase. Realized in a formal grammar, the production for rewriting the non-terminal for verb phrase creates a noun phrase non-terminal, which leads via further productions to the creation of another verb phrase non-terminal that the same rule can apply to. An example of this is shown in figure 3.2.

Recursion is an essential concept in formal languages and computation. In fact, it is the essential ingredient that allows finite grammars to generate infinite languages. A non-recursive grammar has a necessary limit on the length of strings it can generate. Any grammar has a finite number of non-terminals. A path down a derivation tree moves from non-terminal to non-terminal until a terminal is finally reached. The number of non-terminals in the path cannot exceed the number of non-terminals in the grammar unless a non-terminal is repeated in the path. If a non-terminal is repeated in the path, then we have one instance dominating the other, and thus a recursive grammar. If paths in derivation trees cannot exceed the number of non-terminals in the grammar, then there is a limit on how deep trees can be, which in turn limits how long strings can be.

Once you allow recursion into a grammar, there is usually no way to stop it. Consider the grammar in example 3.9. When the rule “X → a X” applies, it generates an “a” symbol, and produces another X, which the same rule can
3.3. REGULAR GRAMMARS

Figure 3.2: A recursive syntactic structure: the same rule applies to the VP at the top and to the VP below it.

apply to. This rule can then apply an arbitrary number of times before switching to a different rule for X, generating an arbitrary number of “a” symbols in the process. This is how Kleene star patterns in languages can be expressed in regular grammars. If there is a sequence of productions that starts and ends with the same non-terminal, that sequence can be repeated zero or more times before alternative productions are chosen to end the derivation.

3.3.4 Limitations of Regular Grammars

There are formal languages which, while they may appear simple to us, are not describable by regular grammars. A very simple example is a language with strings of matched left and right parentheses. There are two terminal symbols, ‘(’ and ‘)’, and they occur in nested pairs, so that example strings are “()”, “(((())))”, and “(((())()))”. No regular grammar can generate all and only such strings. It is easy to construct a regular grammar that generates strings with some number of left parens, followed by some right parens. What a regular cannot do is keep the numbers of left and right parens the same. The problem can be understood as one of memory during the course of the derivation. At each step, the rest of the derivation is determined solely by the identity of the single non-terminal. That non-terminal is the sole source of information about what has already happened in the derivation, the sole “memory” of what symbols have already been produced. The distinguish the difference between having generated two left parens and five left parens would require different
non-terminal symbols for each case. But there are an infinite number of such cases (0 left perens, 1 left peren, 2 left perens, ...), and only a finite number of non-terminals are allowed.

A simple linguistic structure that cannot be captured by regular grammars is a phrase structure grammar in which there are noun phrases and verb phrases, and a sentence may be either a noun phrase followed by a verb phrase, or a noun phrase followed by a verb phrase followed by a noun phrase.

Example 3.11  An attempt to generate phrase structures with a regular grammar.

\[
\begin{align*}
\Sigma & \rightarrow \text{NP} \\
\text{NP} & \rightarrow \text{mark VP} \mid \text{jane VP} \\
\text{VP} & \rightarrow \text{walks} \mid \text{greets NP}
\end{align*}
\]

The problem with this grammar is that it generates “jane greets mark walks”. But \(\text{NP} \rightarrow \text{jane VP}\) is necessary if sentences are going to have more than just a subject. The only way around this is to use a different non-terminal for the noun before the verb and the noun after, losing a primary motivation for the phrase structure analysis.

Example 3.12  Distinguishing NPs before the verb from NPs after the verb avoids the “too many VP” problem, but requires duplicating all of the information concerning NPs.

\[
\begin{align*}
\Sigma & \rightarrow \text{NPS} \\
\text{NPS} & \rightarrow \text{jane VP} \mid \text{mark VP} \\
\text{NPO} & \rightarrow \text{jane} \mid \text{mark} \\
\text{VP} & \rightarrow \text{walks} \mid \text{greets NPO}
\end{align*}
\]

This is unappealing from the point of view of linguistic theory, because different non-terminals are used to describe subject NPs and object NPs, and any generalizations common to NPs would have to be separately stated for each. Again, we see the “memory problem”: the memory that a verb has already been generated has to be encoded in the identity of the following non-terminal in order to have an effect on the derivation.

These limitations follow from the linear structure of regular grammars. If we view the non-terminals of the grammar as identifying the constituents of the language, as is the norm in linguistic analysis, then regular grammars impose a fixed constituent structure on every string of the language: every substring contiguous from some symbol to the right edge of the string is a constituent. This makes regular grammars very simple to analyze and work with, but runs counter to the voluminous evidence that constituent structure in human natural languages does not follow that pattern.
3.3.5 Exercises

Exercise 3.3.1 Consider the following regular grammar:

\[
\begin{align*}
\Sigma & \rightarrow \ c \ X \\
X & \rightarrow \ a \ X \ | \ b \ Y \\
Y & \rightarrow \ b \ Y \ | \ c \ X \ | \ c
\end{align*}
\]

List three strings generated by this grammar.

Exercise 3.3.2 For the grammar in 3.3.1, is the string “c b a c b c” generated by the grammar?

Exercise 3.3.3 Construct a regular grammar generating the language \{“a”\}^*.

Exercise 3.3.4 Construct a regular grammar generating the language \{“a b a”\} \cup \{“b”\} \cdot \{“c”\}^*.

Exercise 3.3.5 Construct a regular grammar generating the language \{ε\} \cup \{“b”\} \cdot \{“c”\}^*.

3.4 Context-Free Grammars

A context free grammar is a formal grammar where all productions have the following form:

- \( A \rightarrow \omega \) where \( \omega \in (Q \cup T)^* - \{\varepsilon\} \) and \( A \in Q \cup \{\Sigma\} \)
- \( \Sigma \rightarrow \varepsilon \)

Here, we are using string notation to characterize the form of the productions themselves. The right-hand side of a production rewriting a non-terminal is a nonempty string of terminals and non-terminals. The significant change from regular grammar productions is that the right-hand sides are not limited to containing a single non-terminal.

Example 3.13 The following grammar captures the before-V / after-V distinction without duplicate categories.

\[
\begin{align*}
\Sigma & \rightarrow \ NP \ VP \\
NP & \rightarrow \ jane \ | \ mark \\
VP & \rightarrow \ walks \ | \ teaches \ NP
\end{align*}
\]

The possibility of more than one non-terminal on the right-hand side permits dependencies to be captured that are not strictly consecutive in nature.

Example 3.14 This grammar generates matching parentheses, center embedded.

\[
\begin{align*}
\Sigma & \rightarrow \ P \\
P & \rightarrow \ ( \ P \ ) \ | \ ( \ )
\end{align*}
\]
Figure 3.3: A derivation tree for a context-free grammar.

Example 3.15 This grammar generates mirror-image symmetric strings of \{a, b\}, formally described as \{ωω^R | ω ∈ \{“a”, “b”\}^*\}:

\[
\begin{align*}
\Sigma & \rightarrow \ X \mid \epsilon \\
X & \rightarrow \ a \ X \ a \mid b \ X \ b \mid a \ a \mid b \ b
\end{align*}
\]

The class of context-free grammars are a generalization of the class of regular grammars: every regular grammar production meets the requirements for context-free grammar productions, so all regular grammars are also context-free grammars. But the reverse is not true. Of particular significance to natural languages, context-free grammars permit constituent structures where some constituents do not extend to the end of the string. An example is the derivation tree in figure 3.3. The PP constituent, covering the substring “with batteries”, extends to neither edge.

While the constituents of context-free grammars do not necessarily extend to the end of the string, they are necessarily contiguous substrings. This means that in order for a non-terminal to control a relationship between two symbols, that non-terminal must dominate every symbol in between. Further, the constituents must be properly nested: two separate non-terminals in a derivation tree either are disjoint in the sets of symbols they dominate, or else one of the non-terminals dominates the other. In the parentheses grammar of example
3.5. **UNRESTRICTED GRAMMARS**

3.14, it is no accident that the grammar simultaneously generates the first left peren and the last right peren, and then the next pair in, and so forth. It would not be possible for a context-free grammar to generate the strings in a way that paired the first left peren with the first right peren, the second with the second, and so forth. This imposes limitations on the kinds of languages that context-free grammars can generate. For instance, the language \( \omega \cdot \omega | \omega \in \{ “a”, “b” \}^* \)
with two identical substrings of ‘a’s and ‘b’s, is not context-free.

### 3.4.1 Exercises

**Exercise 3.4.1** Give a context-free grammar generating the language
\( \omega \cdot “c” \cdot \omega | \omega \in \{ “a”, “b” \}^* \).

**Exercise 3.4.2** Using your grammar of exercise 1, show the step-by-step derivation of the string “a b c b a a”.

**Exercise 3.4.3** Consider the language \( \{ (“a”)^n \cdot (“b”)^n | n \geq 0 \} \), where some number of occurrences of ‘a’ must be followed by an equal number of occurrences of ‘b’. Can this language be generated by a regular grammar? By a context-free grammar?

### 3.5 Unrestricted Grammars

**Unrestricted grammars** permit the re-writing of non-terminals to be sensitive to the context in which the nonterminals occur. Given a set \( Q \) of non-terminals and a set \( T \) of terminals, the possible forms of productions are

- \( \Sigma \rightarrow \omega \)
- \( \lambda A \rho \rightarrow \lambda \omega \rho \)

where \( A \in Q, \lambda, \omega, \rho \in (Q \cup T)^* \).

This schema permits a couple of interesting types of productions:

- context-sensitive productions: \( b B c \rightarrow b b c \)
- contraction productions: \( x A y \rightarrow x y \)

The class of **context-sensitive grammars** includes grammars with context-sensitive productions but not contraction productions. There are some formal languages that can be generated by grammars in the unrestricted class but cannot be generated by strictly context-sensitive grammars. Context-sensitive productions still re-write single non-terminals. The contexts serve to restrict which occurrences of a non-terminal a production can apply to, based on the context in the derivation. The production \( b B c \rightarrow b b c \) can rewrite the non-terminal ‘B’ as the terminal ‘b’ if the ‘B’ is currently immediately preceded by ‘b’ and immediately followed by ‘c’. This means that the production might
be applicable to some occurrences of ‘B’ and not others. It also means that the production might be applicable to the same non-terminal at some points of the derivation but not others, depending upon what happens to the context during the course of the derivation.

Example 3.16 Here is a grammar generating the language \{("a")^n \cdot ("b")^n \cdot ("c")^n \mid n > 0\}.

\begin{align*}
\Sigma & \to A \\
A & \to a ABC \\
A & \to abC \\
C B & \to B C \\
b B & \to bb \\
b C & \to bc \\
c C & \to cc
\end{align*}

It should be noted that this grammar bends the rules a bit: to be strictly correct, rule $C B \rightarrow B C$ should be replaced with the rules \{ $C B \rightarrow XY$, $XY \rightarrow BY$, $BY \rightarrow BC$ \}. 

3.5.1 Exercises

Consider the following grammar.

\begin{align*}
\Sigma & \rightarrow a X \\
a X & \rightarrow a Y a \\
b X & \rightarrow b Y b \\
Y & \rightarrow a X b \\
Y a & \rightarrow a a \\
Y b & \rightarrow b
\end{align*}

Exercise 3.5.1 Give the derivation and draw the tree for the string “a a a”.

Exercise 3.5.2 Give the derivation and draw the tree for the string “a a a b a”.

Exercise 3.5.3 Does this grammar generate “a b b a a”? 

Exercise 3.5.4 Does this grammar generate “b a a b b”?

3.6 Finite State Automata

3.6.1 Finite State Acceptors

An acceptor is a device which, when presented with a string as input, either accepts or rejects the string. An acceptor is said to accept a language if it accepts all of the strings that are in the language, and rejects all of the strings that are not in the language.

A deterministic finite-state acceptor is an object $M = (Q, T, \mathcal{P}, q_I, \mathcal{R})$ where

- $Q$ is a finite set of internal states
3.6. FINITE STATE AUTOMATA

Figure 3.4: An FSA accepting the language \( \{a\}^+ \cdot \{b\}^+ \).

- \( T \) is a finite input alphabet
- \( \mathcal{P} \) is the state transition function \( \mathcal{P} : Q \times T \to Q \)
- \( q_I \in Q \) is the initial state
- \( R \subseteq Q \) are the accepting states

A finite-state acceptor can be non-deterministic if, instead of a state transition function, \( \mathcal{P} \subseteq Q \times T \times Q \) is a transition relation. This effectively makes it possible for the machine to have more than one possible transition to choose at a given point.

Example 3.17 The language \( \{a\}^+ \cdot \{b\}^+ \) is accepted by an FSA with states \( Q = \{1, 2, 3, 4\} \), with \( q_I = 1 \) the start state, \( R = \{3\} \) the accepting states, an input alphabet \( T = \{a, b\} \), and the following transition function:

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,a)</td>
<td>(2)</td>
<td>read ‘a’ and change to state 2</td>
</tr>
<tr>
<td>(1,b)</td>
<td>(4)</td>
<td>read ‘b’ and change to state 4</td>
</tr>
<tr>
<td>(2,a)</td>
<td>(2)</td>
<td>read ‘a’ and stay in state 2</td>
</tr>
<tr>
<td>(2,b)</td>
<td>(3)</td>
<td>read ‘b’ and change to state 3</td>
</tr>
<tr>
<td>(3,a)</td>
<td>(4)</td>
<td>read ‘a’ and change to state 4</td>
</tr>
<tr>
<td>(3,b)</td>
<td>(3)</td>
<td>read ‘b’ and stay in state 3</td>
</tr>
<tr>
<td>(4,a)</td>
<td>(4)</td>
<td>read ‘a’ and stay in state 4</td>
</tr>
<tr>
<td>(4,b)</td>
<td>(4)</td>
<td>read ‘b’ and stay in state 4</td>
</tr>
</tbody>
</table>

In this example, state 4 is what is known as a trap. Once the machine has reached this state, it will never leave, and the state is not an accepting state. The transitions to the trap are in response to symbols whose occurrence indicates that the string cannot belong to the accepted language (like a \( b \) as the first symbol).
Example 3.18 Here is the step-by-step operation of the FSA in example 3.17 on the input string “a a b b”.

<table>
<thead>
<tr>
<th>State</th>
<th>Unread Input</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a a b b</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>a b b</td>
<td>((1,a),(2))</td>
</tr>
<tr>
<td>2</td>
<td>b b</td>
<td>((2,a),(2))</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>((2,b),(3))</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>((3,b),(b))</td>
</tr>
</tbody>
</table>

The final state, 3, is an accepting state, so the string is accepted.

A finite-state acceptor can be non-deterministic if, instead of a state transition function, \( f = Q \times T \times Q \) is a transition relation. This effectively permits the machine to choose from more than one next state, given a current state and the next input symbol. Another extension is to use empty-string transitions, which permit moving from one state to another without consuming any of the input. However, these variations are mere representational conveniences: they do not change the power of the machines. For every language accepted by a non-deterministic FSA, there is a deterministic FSA that accepts the same language.

The states of an FSA can be thought of as containing information about what has already been seen. Thus, they constitute a kind of memory. The current state of machine is its sole representation of what has been seen thus far in a computation. The term "finite state" means just that: a finite number of states. This effectively means a finite amount of memory. Thus, any given FSA, just like any given regular grammar, cannot keep track of arbitrarily large numbers of occurrences of forms; the storage capacity is bounded by the number of states.

The languages accepted by FSAs are exactly the languages generated by regular grammars. This means, of course, that a language that cannot be generated by a regular grammar cannot be accepted by an FSA.

Example 3.19 The language \( \{ ("a")^n \cdot "c" \cdot ("b")^n \mid n > 0 \} \) is not accepted by any FSA.

Example 3.20 One application of FSAs is as efficient lookup table implementations. Consider the following list of words: [book, bike, boa, boat, table, take, tale, talk, trap]. The machine shown in Figure 3.5 will accept all and only the words on the list. Using this approach generally, the number of steps required to determine if an input word is in the list is at most the number of letters in the input, regardless of the number of words in the list.

3.6.2 Finite State Transducers

A finite-state transducer is an object \( M = (Q, T, O, P, q_0, R) \) where
Figure 3.5: An efficient lookup table for a list of words.
• $Q$ is a finite set of internal states
• $T$ is a finite input alphabet
• $O$ is a finite output alphabet
• $P$ is the transition relation $P \subseteq Q \times (T \times O) \times Q$
• $q_I \in Q$ is the initial state
• $R \subseteq Q$ are the accepting states

A transducer has both an input and an output alphabet. Each transition can read an input symbol and write an output symbol.

### 3.6.3 Exercises

Consider the following FSA.

$Q = \{1, 2, 3, 4\}, T = \{a, b\}, q_I = 1, R = \{3\}$

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,a)</td>
<td>(2)</td>
</tr>
<tr>
<td>(1,b)</td>
<td>(4)</td>
</tr>
<tr>
<td>(2,a)</td>
<td>(2)</td>
</tr>
<tr>
<td>(2,b)</td>
<td>(3)</td>
</tr>
<tr>
<td>(3,a)</td>
<td>(4)</td>
</tr>
<tr>
<td>(3,b)</td>
<td>(3)</td>
</tr>
<tr>
<td>(4,a)</td>
<td>(1)</td>
</tr>
<tr>
<td>(4,b)</td>
<td>(1)</td>
</tr>
</tbody>
</table>

**Exercise 3.6.1** Draw a “circle-and-arrow” diagram for the above FSA.

**Exercise 3.6.2** Name three strings, each longer than 4 symbols, which are accepted by the FSA.

**Exercise 3.6.3** Name three strings, each longer than 4 symbols, which are rejected by the FSA.

**Exercise 3.6.4** Show the step-by-step operation of this machine on the string “a a b a b a b”. Be sure to specify if the string is accepted or rejected.

**Exercise 3.6.5** Construct a finite-state accepctor that accepts the language \{“a b a”\}.

**Exercise 3.6.6** Construct a finite-state acceptor that accepts the language \{“b”\}\{“c”\}*. 

**Exercise 3.6.7** Construct a finite-state acceptor that accepts the language \{“a b a”\} \cup (\{“b”\}\{“c”\})*.

**Exercise 3.6.8** Construct a finite-state acceptor that accepts the language \{e\} \cup (\{“b”\}\{“c”\})*.
3.7 Pushdown Automata

A pushdown automaton (PDA) is an object $M = (Q, T, U, P, q_I, R)$ where

- $Q$ is a finite set of internal states
- $T$ is a finite input alphabet
- $U$ is a finite stack alphabet
- $P$, a finite subset of $(Q \times T^* \times U^*) \times (Q \times U^*)$, is the transition relation
- $q_I \in Q$ is the initial state
- $R \subseteq Q$ are the accepting states

The transition relation is a finite set of instructions. Some examples might be:

- $((q_j, "a", "u"), (q_k, "u"))$: if in state $q_j$, the top stack symbol is $u$, and the next input symbol is $a$, read $a$ and change to state $q_k$.
- $((q_j, \epsilon, \epsilon), (q_k, "v"))$: if in state $q_j$, push $v$ onto the top of the stack, and change to state $q_k$ (no input is read).
- $((q_j, "a", "v u"), (q_k, \epsilon))$: if in state $q_j$, the stack has $u$ on top with $v$ immediately beneath, and the next input is $a$, read $a$, pop $u$ and $v$ off of the stack, and change to state $q_k$.

A pushdown automaton is a FSA with an unbounded stack attached. A stack is a symbol storage structure, and functions just like a stack of books. You can always add a symbol to a stack by pushing the new symbol on the top. The only symbol removal operation is to pop the top symbol off of the stack.

A PDA starts in the initial state, with an empty stack. A string is accepted by a PDA iff it succeeds in reading all of the input string and, after reading all of the input string, it is in an accepting state and the stack is empty. A PDA is said to hang if it reaches a configuration where no further steps can be taken, and there remain either unread input or symbols on the stack. If a PDA hangs, the input string is rejected.

As illustrated in the example transitions, transitions can specify the empty string for the input read, the symbols popped onto the stack, or the symbols pushed onto the stack. In each of these cases the empty string acts as a kind of free match. If the input to be read is the empty string, then the transition can apply regardless of the state of the input. It does not mean that the transition can only apply when no input remains. Similarly, if the string of symbols to be popped from the stack is the empty string, this means that the transition can apply regardless of the current state of the stack; it does not restrict the transition to apply only when the stack is empty.
CHAPTER 3. COMPUTATION

Notation 3.1 When a string of more than one symbol is indicated for a stack operation, a convention is needed as to which end represents the top of the stack. We will represent stacks in text has “rising to the right”, so that the rightmost element is the top of the stack. In keeping with this, a stack operation will be represented with the top symbol to the right, as illustrated in the example transition \((q_j, \text{ “a”}, v u), (q_k, \epsilon)\) above.

Notice that for every FSA, there is an equivalent PDA, one which ignores its stack.

Example 3.21 A PDA accepting the language \(\{ (“a”)^n \cdot (“c”) \cdot (“b”)^n \mid n \geq 0 \}\): \(Q = \{1, 2\}, \mathbb{T} = \{a, b, c\}, U = \{A\}, q_I = 1, \mathbb{R} = \{2\}\), and the transitions are

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>((1, \text{ “a”}, \epsilon))</td>
<td>((1, \text{ “A”}))</td>
<td>read a and push A on the stack</td>
</tr>
<tr>
<td>((1, \text{ “c”}, \epsilon))</td>
<td>((2, \epsilon))</td>
<td>read c and change to state 2</td>
</tr>
<tr>
<td>((2, \text{ “b”}, \text{ “A”}))</td>
<td>((2, \epsilon))</td>
<td>read b and pop A off the stack</td>
</tr>
</tbody>
</table>

Example 3.22 Here is the operation of the above PDA on the string “a a c b b”.

<table>
<thead>
<tr>
<th>State</th>
<th>Unread Input</th>
<th>Stack</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a a c b b</td>
<td>[ ]</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a c b b</td>
<td>[A ]</td>
<td>((1, \text{ “a”}, \epsilon), (1, \text{ “A”}))</td>
</tr>
<tr>
<td>1</td>
<td>c b b</td>
<td>[ A A ]</td>
<td>((1, \text{ “a”}, \epsilon), (1, \text{ “A”}))</td>
</tr>
<tr>
<td>2</td>
<td>b b</td>
<td>[ A A ]</td>
<td>((1, \text{ “c”}, \epsilon), (2, \epsilon))</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>[ A ]</td>
<td>((2, \text{ “b”}, \text{ “A”}), (2, \epsilon))</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>[ ]</td>
<td></td>
</tr>
</tbody>
</table>

3.7.1 Exercises

Here is a PDA accepting the language \(\{ \omega \omega^R \mid \omega \in \{ \text{ “a”, “b”} \}^* \}\).

\(Q = \{1, 2\}, \mathbb{T} = \{a, b\}, U = \{A, B\}, q_I = 1, \mathbb{R} = \{1, 2\}\)

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>((1, \text{ “a”}, \epsilon))</td>
<td>((1, \text{ “A”}))</td>
</tr>
<tr>
<td>((1, \text{ “a”}, \text{ “A”}))</td>
<td>((2, \epsilon))</td>
</tr>
<tr>
<td>((1, \text{ “b”}, \epsilon))</td>
<td>((1, \text{ “B”}))</td>
</tr>
<tr>
<td>((1, \text{ “b”}, \text{ “B”}))</td>
<td>((2, \epsilon))</td>
</tr>
<tr>
<td>((2, \text{ “a”}, \text{ “A”}))</td>
<td>((2, \epsilon))</td>
</tr>
<tr>
<td>((2, \text{ “b”}, \text{ “B”}))</td>
<td>((2, \epsilon))</td>
</tr>
</tbody>
</table>

This PDA exhibits nondeterminism: there are circumstances where the machine has a choice of more than one action it can take. Whether it will successfully accept a string can depend in part on what choice is made at each such circumstance.
3.8. TURING MACHINES

Exercise 3.7.1 Consider the string “a b b a”.

1. Show a step-by-step operation where the machine accepts the string.
2. Show a step-by-step operation where the machine does not accept the string (i.e., it hangs).

Exercise 3.7.2 Show a step-by-step operation of this machine on the string “a b a”.

Exercise 3.7.3 Give a PDA accepting the language \( \{ \omega \cdot \text{“c”} \cdot \omega^R | \omega \in \{ “a”, “b” \}^* \} \).

NOTE: ‘c’ is a terminal symbol in this language. Does your machine exhibit nondeterminism?

Exercise 3.7.4 Using your PDA of 3.7.3, show the step-by-step operation on the string “a a b c b a a”.

Exercise 3.7.5 Using your PDA of 3.7.3, show the step-by-step operation on the string “a c b”.

3.8 Turing Machines

3.8.1 Turing Machine Definitions and Conventions

A Turing machine (TM) is an object \( M = (Q, q_I, h, \mathcal{T}, \mathcal{P}) \) where

- \( Q \) is a finite set of internal states
- \( q_I \in Q \), the initial state
- \( h \) is the halt state (not included in \( Q \)); every Turing machine has the same halt state
- \( \mathcal{T} \) is a finite alphabet, including the blank symbol \( # \) (but not \( L \) or \( R \))
- \( \mathcal{P} \) is a transition function \( (Q \times \mathcal{T}) \rightarrow ((Q \cup \{ h \}) \times (\mathcal{T} \cup \{ L, R \})) \)

A Turing machine replaces the PDA’s stack with a more general storage mechanism, a tape. The tape consists of an unbounded sequence of squares, each of which can contain one symbol. A TM can navigate back and forth along the tape by moving left and right in one-square steps.

A TM can do three kinds of things: it can write a symbol in the current square (overwriting whatever the current symbol is), it can move one square to the left on the tape, or it can move one square to the right. The transitions of a machine are dependent on the current state and the symbol in the tape square being read. Some examples might be:

- \( ((q_i, a), (q_j, u)) \): if in state \( q_i \) and the current tape square contains \( a \), write \( u \) in the current tape square and change to state \( q_j \).
• \(((q_i, a), (q_j, R))\): if in state \(q_i\) and the current tape square contains \(a\), move one square to the right on the tape and change to state \(q_j\).

• \(((q_i, \#), (q_j, L))\): if in state \(q_i\) and the current tape square is blank, move one square to the left and change to state \(q_j\).

• \(((q_i, b), (b, \#))\): if in state \(q_i\) and the current tape square contains \(b\), write a blank in the current tape square and halt.

Once a Turing machine has been started, there are two broad types of outcomes: the machine eventually halts, or it never halts.

Although not required by the definition of Turing machines, we will use some conventions for the encoding of information on tapes to make things simpler. We will always include the symbol ‘[’ in the tape alphabet, to mark the left edge of the tape. When an input string is placed on the tape of an initialized TM, the leftmost tape square will start with the symbol ‘[’ in it, and the read/write head will start out on that square. The intended input string, the input to the computation, will be placed in the squares to the right of that initial square.

**Example 3.23** The following Turing machine computes the function which takes a string \(s \in \{“a”, “b”\}^*\) and replaces all of the \(a\)’s with \(b\)’s (and vice-versa). The machine has three states: \(Q = \{1, 2, 3\}\), and a tape alphabet of \(T = \{a, b, [, \#]\}\). The initial state is \(q_i = 1\), and the machine starts with the tape head pointing to the leftmost square of the input string. A well-formed computation will end with the tape head pointing the leftmost square, which should still contain ‘[’.

The transitions are:

<table>
<thead>
<tr>
<th>(((1,[), (1,R)))</th>
<th>(((3,[), (h,[)))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(((1,a), (2,b)))</td>
<td>(((2,a), (1,R)))</td>
</tr>
<tr>
<td>(((1,b), (2,a)))</td>
<td>(((2,b), (1,R)))</td>
</tr>
</tbody>
</table>
| \(((1,#), (3,L))\) | \(((2,#), (3,L))\) | \n
**Example 3.24** Here is the step-by-step operation of the Turing machine of example 3.23 on the input string “a a b”.

```
3.8. TURING MACHINES

<table>
<thead>
<tr>
<th>State</th>
<th>Tape - Left</th>
<th>Tape - current and right</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/</td>
<td>a a b #</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>/</td>
<td>b a b #</td>
<td>((1,a),(2,b))</td>
</tr>
<tr>
<td>2</td>
<td>/</td>
<td>b a b #</td>
<td>((1,a),(2,b))</td>
</tr>
<tr>
<td>1</td>
<td>/</td>
<td>b b #</td>
<td>((2,b),(1,R))</td>
</tr>
<tr>
<td>2</td>
<td>/</td>
<td>b b #</td>
<td>((1,a),(2,b))</td>
</tr>
<tr>
<td>1</td>
<td>/</td>
<td>b b b #</td>
<td>((2,b),(1,R))</td>
</tr>
<tr>
<td>2</td>
<td>/</td>
<td>b b b #</td>
<td>((1,b),(2,a))</td>
</tr>
<tr>
<td>1</td>
<td>/</td>
<td>b b b #</td>
<td>((1,b),(2,a))</td>
</tr>
<tr>
<td>3</td>
<td>/</td>
<td>b b b #</td>
<td>((2,a),(1,R))</td>
</tr>
<tr>
<td>3</td>
<td>/</td>
<td>b b b #</td>
<td>((1,#),(3,L))</td>
</tr>
<tr>
<td>3</td>
<td>/</td>
<td>b b b #</td>
<td>((3,a),(3,L))</td>
</tr>
<tr>
<td>3</td>
<td>/</td>
<td>b b b #</td>
<td>((3,b),(3,L))</td>
</tr>
<tr>
<td>h</td>
<td>/</td>
<td>b b b #</td>
<td>((3,),(h,))</td>
</tr>
</tbody>
</table>

A Turing machine can be a language acceptor by having it compute a function from $T^*$ to a binary set \{Y,N\}, where the machine accepts a string by halting with Y on the tape, and rejects the string by halting with N on the tape. We will adopt the convention that when a TM is used an acceptor, it should halt with all of the tape blank except for the leftmost square, where the answer symbol, Y or N, has overwritten the initial '['.

Turing machines actually distinguish two classes of languages. A language is Turing decidable if there is a machine that halts for any input, leaving a N or Y indicating if the string is in the language. A language is Turing acceptable if there exists a Turing machine which will halt for any input string in the language, but does not halt when given an input string not in the language. All Turing decidable languages are Turing acceptable, but the reverse is not true.

Whether or not a given Turing machine halts on a given input is a significant issue in the structure of the machine.

3.8.2 Example Machines

Turing machines can accept many languages that PDAs cannot. One example is $\{(\text{"a"})^n \cdot (\text{"b"})^n \cdot (\text{"c"})^n | n > 0\}$.

**Example 3.25** A Turing machine that accepts $\{(\text{"a"})^n \cdot (\text{"b"})^n \cdot (\text{"c"})^n | n > 0\}$. $Q = \{1, \ldots, 8\}$. $q_1 = 1$. $T = \{a, b, c, x, y, z, Y, N, [\], \#\}$. We will think of the actual computation performed by the machine as a function \{a", "b", "c"\} $\times T \rightarrow \{Y, N\}$. The transitions are given in the tables below, with column headings to link sets of transitions to different "phases" of the computation.

The first part of the computation replaces an "a" with an "x" to mark it as checked off. The machine then searches to the right for a "b", and if it finds one replaces it with a "y". It then searches for and replaces a "c" with a "z". After replacing "c" with "z", it returns to the left, looking for the "x" it placed.
The machine then starts scanning to the right, looking for another “a”. If it finds one, it starts another round of replacement, replacing one each of “a”, “b”, “c”.

<table>
<thead>
<tr>
<th>Replace a</th>
<th>Replace b</th>
<th>Replace c</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>((1, [ ),(1,R))</td>
<td>((2, a),(2,R))</td>
<td></td>
<td>((4, a),(4,L))</td>
</tr>
<tr>
<td>((1,b),(6,R))</td>
<td>((2, b),(3,y))</td>
<td>((3,b),(3,R))</td>
<td>((4,b),(4,L))</td>
</tr>
<tr>
<td>((1,c),(6,R))</td>
<td>((2, c),(6,R))</td>
<td>((3,c),(4,z))</td>
<td>((4,c),(4,L))</td>
</tr>
<tr>
<td>((1,#),(7,L))</td>
<td>((2,#),(7,L))</td>
<td>((3,#),(7,L))</td>
<td></td>
</tr>
<tr>
<td>((2,x),(2,R))</td>
<td></td>
<td></td>
<td>((4,x),(1,R))</td>
</tr>
<tr>
<td>((1,y),(5,R))</td>
<td>((2,y),(2,R))</td>
<td>((3,y),(3,R))</td>
<td>((4,y),(4,L))</td>
</tr>
<tr>
<td>((2,z),(6,R))</td>
<td></td>
<td></td>
<td>((4,z),(4,L))</td>
</tr>
</tbody>
</table>

If an “x”, marking a replaced “a”, is encountered instead during the return phase, then the machine goes into a check phase, making sure that there are no remaining unreplaced “b” or “c” symbols. If there are none, then the succeed phase is entered, which erases the tape, puts a “Y” at the beginning, and halts. Otherwise, it enters the fail phase, which erases the tape, puts a “N” at the beginning, and halts.

<table>
<thead>
<tr>
<th>Check</th>
<th>Fail - Rt</th>
<th>Fail - Lf</th>
<th>Succeed</th>
</tr>
</thead>
<tbody>
<tr>
<td>((6,a),(6,R))</td>
<td></td>
<td></td>
<td>((8, [ ),(h,Y))</td>
</tr>
<tr>
<td>((5,b),(6,R))</td>
<td></td>
<td></td>
<td>((7,b),(7,#))</td>
</tr>
<tr>
<td>((5,c),(6,R))</td>
<td></td>
<td></td>
<td>((7,c),(7,#))</td>
</tr>
<tr>
<td>((5,#),(8,L))</td>
<td></td>
<td></td>
<td>((8,#),(8,L))</td>
</tr>
<tr>
<td>((6,x),(6,R))</td>
<td></td>
<td></td>
<td>((8,x),(8,#))</td>
</tr>
<tr>
<td>((5,y),(5,R))</td>
<td></td>
<td></td>
<td>((8,y),(8,#))</td>
</tr>
<tr>
<td>((5,z),(5,R))</td>
<td></td>
<td></td>
<td>((8,z),(8,#))</td>
</tr>
</tbody>
</table>

As the above example machine shows, defining a TM to perform even simple-sounding computations can involve a fair amount of tedious detail. This property extends to the step-by-step operation of the machines as well.

**Example 3.26** The step-by-step operation of the machine for the input “a a b b c c”, broken into several stages.

The start, and the first round of replacement.
3.8. TURING MACHINES

<table>
<thead>
<tr>
<th># - State</th>
<th>Tape - Left</th>
<th>Tape - current and right</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[</td>
<td>a a b b c c #</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>[</td>
<td>a a b b c c #</td>
<td>(1, [ ),(1,R))</td>
</tr>
<tr>
<td>2</td>
<td>[x</td>
<td>a b b c c #</td>
<td>(1, a),(2,x))</td>
</tr>
<tr>
<td>2</td>
<td>[x a</td>
<td>b b c c #</td>
<td>(2, a),(2,R))</td>
</tr>
<tr>
<td>3</td>
<td>[x a y</td>
<td>b c #</td>
<td>(2, b),(3,y))</td>
</tr>
<tr>
<td>3</td>
<td>[x a y b</td>
<td>c c #</td>
<td>(3, b),(3,R))</td>
</tr>
<tr>
<td>4</td>
<td>[x a y b</td>
<td>z c #</td>
<td>(3, c),(4,z))</td>
</tr>
<tr>
<td>4</td>
<td>[x a</td>
<td>y b c #</td>
<td>(4, b),(4,L))</td>
</tr>
<tr>
<td>4</td>
<td>[x</td>
<td>a y b c #</td>
<td>(4, y),(4,L))</td>
</tr>
<tr>
<td>4</td>
<td>[</td>
<td>x a y b c #</td>
<td>(4, a),(4,L))</td>
</tr>
</tbody>
</table>

The second round of replacement.

<table>
<thead>
<tr>
<th># - State</th>
<th>Tape - Left</th>
<th>Tape - current and right</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[x</td>
<td>a y b c c #</td>
<td>(4, x),(1,R))</td>
</tr>
<tr>
<td>2</td>
<td>[x</td>
<td>y b c c #</td>
<td>(4, y),(2,R))</td>
</tr>
<tr>
<td>2</td>
<td>[x x y</td>
<td>b c #</td>
<td>(4, y),(2,R))</td>
</tr>
<tr>
<td>3</td>
<td>[x x y y</td>
<td>z c #</td>
<td>(5, y),(3,R))</td>
</tr>
<tr>
<td>3</td>
<td>[x x y y z</td>
<td>c #</td>
<td>(5, z),(3,R))</td>
</tr>
<tr>
<td>4</td>
<td>[x x y y</td>
<td>z #</td>
<td>(5, z),(4,z))</td>
</tr>
<tr>
<td>4</td>
<td>[x x y y</td>
<td>y y z z #</td>
<td>(5, z),(4,L))</td>
</tr>
<tr>
<td>4</td>
<td>[x</td>
<td>x y y z z #</td>
<td>(5, y),(4,L))</td>
</tr>
</tbody>
</table>

The search for a third “a” turns up a “y” instead, meaning there are no more “a” symbols at the beginning. The machine then checks to make sure there are no remaining unreplaced “b” or “c”.

<table>
<thead>
<tr>
<th># - State</th>
<th>Tape - Left</th>
<th>Tape - current and right</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>[x x y y</td>
<td>y y z z #</td>
<td>(5, y),(5,R))</td>
</tr>
<tr>
<td>5</td>
<td>[x x y y</td>
<td>y z z #</td>
<td>(5, z),(5,R))</td>
</tr>
<tr>
<td>5</td>
<td>[x x y y</td>
<td>z #</td>
<td>(5, z),(5,R))</td>
</tr>
<tr>
<td>5</td>
<td>[x x y y</td>
<td>z z #</td>
<td>(5, z),(5,R))</td>
</tr>
</tbody>
</table>

Finally, the machine erases the tape and halts with a Y.
3.8.3 Universal Turing Machines

A significant fact about Turing machines is that they are finitely describable. A Turing machine can be represented as a string. The string would list the states, the symbol alphabet, the transitions, and so forth. This has a couple of interesting consequences:

- The set of all Turing machines definable with a given alphabet is a subset of the set of all strings on that alphabet.

- It creates the opportunity to feed a string which is the description of a Turing machine as the input to another Turing machine.

It is possible to construct an encoding of Turing machines as strings, along with a special Turing machine, called a universal Turing machine, such that if a string encoding of any Turing machine, along with an input for the encoded machine, are combined to form an input for the universal Turing machine, the universal machine will simulate the execution of the encoded machine on the input.

The possibility of universal Turing machines implies the possibility of building only one Turing machine (a universal Turing machine), but being able to compute the function of any Turing machine by feeding a description of that machine to the (physically built) universal Turing machine. This is, in fact, the origin of the modern programmable computer.

3.8.4 Exercises

**Exercise 3.8.1** Show the step-by-step operation of the Turing machine in example 3.25 on the input string “a b c”.
Exercise 3.8.2 Show the step-by-step operation of the Turing machine in example 3.25 on the input string “a b”.

Exercise 3.8.3 One of the transitions for the machine in example 3.25 is ((1,c),(6,R)). Explain what this transition does, and why the machine has this transition (what does it contribute to the computation of the function?).

3.9 Computability Theory

3.9.1 The Chomsky Hierarchy

There are non-arbitrary relationships between the classes of grammars we have discussed, and the classes of automata we have discussed. The relationships were discovered by Noam Chomsky, during the late 1950s and early 1960s. These relationships form what has come to be known as the Chomsky hierarchy, and are summarized in the table below.

<table>
<thead>
<tr>
<th>Grammars</th>
<th>Machines</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular</td>
<td>Finite State</td>
<td>Regular</td>
</tr>
<tr>
<td>Context-Free</td>
<td>Pushdown</td>
<td>Context-Free</td>
</tr>
<tr>
<td>Context-sensitive</td>
<td>Turing</td>
<td>Turing decidable</td>
</tr>
<tr>
<td>Unrestricted</td>
<td>Turing</td>
<td>Turing acceptable</td>
</tr>
</tbody>
</table>

The languages generated by each class of grammars in the table are precisely the languages accepted by the corresponding class of machines.

In the Chomsky hierarchy, the most powerful and general class of automata is the class of Turing machines, and the most general class of languages is the class of Turing acceptable languages. This naturally raises the question of what the most general class of formal languages is that can be accepted by any machine or procedure. This is not a question for which an answer can be proven, for it begs the question of what can count as a machine or procedure. However, it is widely believed that the class of Turing acceptable languages is the most general class of languages expressible by machine or grammar; it is the most general class of computable languages. This claim is known as Church’s thesis, named after the mathematician Alonzo Church.

Church’s Thesis: the Turing acceptable languages are the most general class of formal languages characterizable by any kind of procedure.

The unrestricted grammars are distinguished from the context-sensitive grammars by the possibility of contraction productions: productions that rewrite a non-terminal as the empty string. The distinction lines up with the distinction between Turing decidable and Turing acceptable in the Chomsky hierarchy. With context-sensitive grammars, the length of the derivational string in a step-by-step derivation never lowers: a non-terminal is always rewritten with at least one symbol, so the length either grows or stays the same. Thus, for any string formed in the process of a derivation, you know that the terminal string ultimately derived will be at least as long as that derivational string. In determining if a given terminal string is generated by a context-sensitive grammar,
one can try all possible derivations up to a certain length, the length of the terminal string. If a particular derivational path gets longer than the terminal string, it can be abandoned, as no further application of productions will result in the given terminal string. This is what makes the languages generated by context-sensitive grammars decidable, and in fact one can prove that they are decidable by showing the existence of Turing machines that, given a terminal string, try all possible derivations for a string that length, and conclude that the string is not in the language if none of them derive the input string. Because there are a finite number of distinct derivations for any given length of derivational string, the possible derivations will be exhausted after a finite number of steps.

Another way to view this approach to evaluating strings is in terms of enumeration. For any grammar, it is possible to enumerate the strings generated by the grammar, in the same sense as is used with countable sets: pair one string with 1, another string with 2, another with 3, and so forth. For context-sensitive grammars, it is possible to enumerate them such that the shortest strings appear the earliest in the enumeration. The Turing machine generates the strings of the language in order by size, and either matches the input string, or quits when it has exhausted all of the strings as short as the input.

The enumeration approach will not have the same effect for grammars with contraction productions. With unrestricted grammars, a derivational string could grow to great length, and then shrink down again, as non-terminals are rewritten as the empty string. The general approach to demonstrating Turing acceptability is also to have the machine enumerate the strings of the language. But the strings are not guaranteed to be enumerated in order by size. Thus, the best the machine can do is keep going through the list of strings. If it finds a string in the language matching the input, then it can conclude that yes, the string is in the language, and halt. If the input string is in the language, that is guaranteed to happen after some finite number of steps. If the input string is not in the language, then the machine just keeps enumerating the strings of the language forever, never halting. It has no way of knowing whether or not the next string it will enumerate matches the input string.

### 3.9.2 Uncomputability

Uncomputable functions exist (in fact, there are a lot of them). This follows directly from the fact that there are countably many possible Turing machines, while there are uncountably many different formal languages on a non-empty finite alphabet.

One example of an uncomputable function is the halting problem. This is the problem of determining whether a given Turing machine will halt on a given input string. We can prove that this problem is not computable, that is, no Turing machine exists which can compute it.

The proof of this result is by contradiction. Suppose there existed a Turing machine H that could determine if other Turing machines would halt. H takes as input a description of a machine M and input I, and determines if M halts on
I. It follows that $H$ can determine if a machine $M$ halts when given a description of itself ($M$) as input.

If we can build $H$, then we can build $H_2$, which takes as input a description of a machine $M$, and determines if $M$ halts when given a description of itself as input. If $M$ halts on a description of itself, then $H_2$ goes into an infinite loop (never halts); if $M$ does not halt on a description of itself, then $H_2$ halts, giving a NO as output.

The problem comes when we give $H_2$ a description of itself as input.

- If $H_2$ determines that $H_2$ halts when given itself as input, then $H_2$ has failed, because by definition it should go into an infinite loop, contradicting the conclusion that it halts when given itself as input.

- If $H_2$ determines that $H_2$ does not halt when given itself as input, then $H_2$ has again failed, because by definition it should halt with output NO, contradicting the conclusion that it does not halt when given itself as input.

Thus, the idea of a Turing machine which can solve the halting problem turns out to be logically contradictory.
Chapter 4

Proof

4.1 Introduction

This chapter presents some basics of first order logic, and of the concept of proving the correctness of conclusions from premises using logic. It then presents a computational technique for automatic theorem-proving called resolution. This technique is embodied in a computer programming language called prolog. The name prolog derives from “programming in logic.” As is explained below, prolog cannot construct proofs for all of general first-order logic, but the subset that prolog is restricted to is still powerful enough to be useful and interesting.

The kinds of proofs constructed by prolog are good for proving that statements hold of individual entities on a case-by-case basis. However, using them to prove a formula correct for an infinite number of entities, for instance proving that a mathematical formula will work for all integers, under this approach would require constructing an infinite number of proofs. This chapter presents a proof technique called mathematical induction, which, given the proper conditions, can in a compact way prove a statement to be simultaneously true of an infinite number of entities.

4.2 Formal Systems of Logic

4.2.1 What is a Logic?

In the most general and intuitive sense, a logic is a way of relating the truth values of different propositions. These relationships allow one to conduct reasoning: determining the truth value of one proposition on the basis of the truth values of other, related, propositions.

Any particular logical system will include one or more rules of inference. A rule of inference can be thought of as a principle relating the truth values of different propositions, allowing one to conclude the truth of a particular proposition, the conclusion, based upon the truth of other propositions, the
**premises.** Logics can differ in what types of things they take as propositions, and what rules of inference are assumed.

Some rules of inference are quite common and familiar:

- *Modus ponens* says that the truth of \( \alpha \) and \( \alpha \Rightarrow \beta \) entail the truth of \( \beta \).

- *Modus tollens* says that the truth of \( \neg \beta \) and \( \alpha \Rightarrow \beta \) entail the truth of \( \neg \alpha \).

Rules of inference are different from logical operators. Logical operators are used to *define* new propositions; they don’t accomplish any reasoning in their own right, just the construction of new propositions from others. Rules of inference are used to determine the truth values of independently defined propositions. The logical operator of implication defines the proposition \( \alpha \Rightarrow \beta \) in terms of the truth values of the propositions \( \alpha \) and \( \beta \). Modus ponens, when adopted in a logical system, does not define the predicate \( \beta \). Modus ponens does allow one to infer the truth of \( \beta \) from the truth of \( \alpha \) and \( \alpha \Rightarrow \beta \). Clearly, the inference rule modus ponens and the logical operator implication are not unrelated; the former is defined in terms of the latter.

Rules of inference relate the syntactic structures of the formulae expressing the propositions. This is the essence of logic: reasoning solely on the basis of the formal structure of formulae.

A formal system of logic consists of three parts:

- A set of propositions.
- A set of axioms, which are propositions presumed to be true.
- A set of rules of inference.

This characterization is actually over-restrictive: by invoking the language of “propositions” and “truth”, it essentially presumes a two-valued logic in which all propositions are true or false, when in fact there are such things as logics with more than two “truth” values. But we will adopt this view for the time being, in deference to the fact that the overwhelming majority of work with logic uses binary, true/false logics.

### 4.2.2 First Order Logic (FOL)

For current purposes, the propositions of a formal logical system can be understood to be a set of strings. An alphabet and a set of well-formedness rules define the strings in the system; these strings are called *well-formed formulae*, or WFFs. The WFFs are the formulae that having meaning within the system; they constitute the propositions.

**Example 4.1** Consider the alphabet \( A = \{ p, q, r, \wedge \} \), along with the rules: (1) \( p, q, \) and \( r \) are formulae; (2) if \( x \) and \( y \) are formulae, then \( x \wedge y \) is a formula. From these, the following are examples of WFFs: \( p, p \wedge p, q \wedge r, r \wedge q, q \wedge r \wedge p \).
4.2. FORMAL SYSTEMS OF LOGIC

The preceding example is a very simple, restricted system. It does precisely determine what the possible WFFs are. It is not, however, first order logic.

In FOL, the formulas for primitive propositions are called atomic formulas, and are formed from terms and predicates.

Terms

1. constants (each constant is a string)
2. variables (each variable is a string)
3. compound function terms \( f(t_1, \ldots, t_m) \), where \( f \) is a function and \( t_i \) are terms (each function \( f \) is a string)

Atomic formulas: \( p(t_1, \ldots, t_k) \)

1. \( p \) is a predicate of arity \( k \) (each predicate is a string).
2. \( t_i \) is a term.

Well-formed formulae are defined recursively, starting with the atomic formulas.

WFFs (below, assume \( \phi \) and are \( \psi \) WFFs)

1. all atomic formulas
2. \( (\phi) \)
3. \( \phi \land \psi \)
4. \( \phi \lor \psi \)
5. \( \neg \phi \)
6. \( \phi \Rightarrow \psi \)
7. \( (\forall x) \phi \)
8. \( (\exists x) \phi \)

The logic is first order because predicates only take terms as arguments, and terms themselves cannot be predicates.

A literal is either an atomic formula \( A \) (a positive literal) or a negated atomic formula \( \neg A \) (a negative literal).

A clause is a disjunction of literals: \( L_1 \lor \neg L_2 \lor \ldots \)
4.2.3 Deduction

*Deduction* is sometimes defined as reasoning from the general to the specific. A perhaps more useful description might be reasoning by applying rules of inference to axioms to conclude the truth of another proposition.

A *theorem* is a proposition that can be deduced from a set of axioms. A *proof* is a particular sequence of valid deductions leading from the axioms to the conclusion of a theorem. Note that there can be many distinct proofs of the same theorem, and different techniques for constructing proofs.

**Example 4.2** Suppose we have the following axioms, where \( b \) is a constant, and \( p \), \( q \), and \( r \) are predicates:

\[
\begin{align*}
  p(b) & \implies q(b) \\
  (q(b) \land r(b)) & \implies s(b) \\
  p(b) & \\
  r(b)
\end{align*}
\]

From those axioms, we would like to prove the following theorem: \( s(b) \), that is, that the predicate \( s \) holds of the constant \( b \).

**Proof.** Observe that, by assumption, \( p(b) \) and \( p(b) \implies q(b) \) are true. By applying modus ponens to these two, we may conclude that \( q(b) \) is true. Because both \( q(b) \) and \( r(b) \) are true (the former we derived, the latter is an axiom), we may conclude that the formula \( (q(b) \land r(b)) \) is true. From \( (q(b) \land r(b)) \) and \( (q(b) \land r(b)) \implies s(b) \), modus ponens permits us to deduce the truth of \( s(b) \), which is our theorem.

The theorem just proven is quite specific and concrete, a predicate holding of a constant. Not every WFF is so straightforward as a candidate for a theorem. For instance, WFFs with unbound variables aren’t really provable propositions. If I assert \( p(x) \), where \( x \) is a variable, just what am I asserting? This is to be distinguished from expressions with bound variables: if I assert \( \forall x (p(x)) \), I’m asserting a specific proposition which must be true or false with respect to some situation of possible values for the variable.

This does not stop us from using WFFs with unbound variables as “theorem-like.” The interpretation we will commonly use in such circumstances is that part of looking for a proof of a WFF with unbound variables is to find values to bind to those variables such that the resulting expression is true. Thus, with \( b \) a constant and \( x \) a variable, I can ask of \( p(b) \) “is it true or not?”, while I can ask of \( p(x) \) “is there a value that can be substituted for \( x \) such that the resulting formula is true?”. In other words, we will interpret the unbound variables as being implicitly existentially quantified: asking about \( p(x) \) will be taken to be asking about the truth of \( \exists x p(x) \). Prolog will use this interpretation when we ask it if it can prove correct a formula with unbound variables in it.

4.2.4 Inconsistency

A set of WFFs is *inconsistent* if it supports the proof of both a formula and that formula’s negation. A set of WFFs is consistent if it is not inconsistent (simply
4.3. RESOLUTION

In propositional logic, a simple example of an inconsistent set of formulas would be \{p, \neg p\}. This is inconsistency in its baldest sense, because one of the formulas is the negation of the other. The set of formulas \{p, \neg p\} entails \( p \) and it entails \( \neg p \), thus it is inconsistent. The formula formed by the conjunction of the two formulas, \((p \land \neg p)\), is itself false, necessarily. Because the conjunction is inconsistent, we know that \((p \land \neg p)\) is false regardless of whether \( p \) itself is true or false.

Inconsistency can provide the basis for another proof technique, proof by contradiction. A proof by contradiction of a theorem is achieved by (temporarily) assuming the negation of the theorem, combining it with axioms, and via deduction concluding a statement known to be false. This last part, the conclusion of something known to be false, is the “contradiction” part. If, by assuming the negation of the theorem, we deduce a contradiction, then the negation of the theorem must be false. It follows that the theorem itself is necessarily true.

The kind of contradiction we will be focusing on is inconsistency: if, by adding the negation of a formula \( f \) to our axioms, we get an inconsistent set of formulas, then the original set of axioms must entail the formula \( f \) itself. The demonstration that \( \neg f \) is inconsistent with the axioms constitutes a proof from the axioms to \( f \).

Recall example 4.2 above. We could construct a proof by contradiction for \( s(b) \).

**Proof.** Suppose, to the contrary, that \( \neg s(b) \) is true. Because both \((q(b) \land r(b)) \Rightarrow s(b) \) and \( \neg s(b) \) are true, it follows (by modus tollens) that \( \neg (q(b) \land r(b)) \) is true. \( \neg (q(b) \land r(b)) \) entails \( \neg q(b) \lor \neg r(b) \). The truth of \( \neg q(b) \lor \neg r(b) \) and \( r(b) \) entails \( \neg q(b) \). The truth of \( \neg q(b) \) and \( p(b) \Rightarrow q(b) \) entails \( \neg p(b) \). BUT, \( p(b) \) is an axiom. This gives us our contradiction. Therefore, the assumption of \( \neg s(b) \) must be false, and the axioms must entail \( s(b) \).

Notice that the proof by contradiction in this case appears a little like the earlier forward proof, but in reverse.

A key to the reasoning here is our knowledge that the set of axioms from example 4.2 is consistent. If we add \( \neg s(b) \) to the axioms, and the resulting set of propositions is inconsistent, then we may conclude that the original set of axioms entails \( s(b) \).

4.3 Resolution

4.3.1 Horn Clauses

General WFFs of first order logic are sufficiently complex to make it difficult to define general proof procedures for them. As a result, prolog uses a strict subset of the formulas of first order logic, called Horn clauses. The primary defining characteristic of a Horn clause is that it has at most one positive literal.

Horn clauses can be any of three types:
Chapter 4. Proof

1. Unit clauses \((L_1)\)
   
   (a) one positive literal, and no negative literals
   
   (b) these can be thought of as facts.

2. Nonunit clauses \((L_1 \lor \neg L_2 \lor \neg L_3 \lor \ldots)\)
   
   (a) one positive literal, and one or more negative literals
   
   (b) these can be thought of as rules.

3. Negative clauses \((\neg L_1 \lor \neg L_2 \lor \ldots)\)
   
   (a) no positive literals, and one or more negative literals
   
   (b) these can be thought of as queries.

The first two types of clauses are collectively referred to as definite Horn clauses, because they have a single positive literal (a single, or definite, conclusion). The neat thing about definite Horn clauses is that a collection of them can never be inconsistent, because it is impossible to directly assert the falsity of a clause using definite Horn clauses.

A set of definite Horn clauses is called a program. We will interpret a program conjunctively: the assertion of a program means the assertion of the simultaneous truth of all clauses in the program. Note that a program is not itself a WFF, it is a collection of WFFs of a particular kind.

The significance of queries (negative clauses) will become apparent below.

4.3.2 Implications as Horn Clauses

Implication is a binary logical operator, denoted as \(p \Rightarrow q\), and sometimes written as “\(p\) entails \(q\)”, or “if \(p\) then \(q\)”. It can be defined as \(q \lor \neg p\). For reasons having to do with prolog notation, it will frequently be convenient to write the arrow notation of the implication pointing from right to left: \(q \Leftarrow p\) means the same thing as \(p \Rightarrow q\).

The definition of implication in terms of a disjunction is quite useful, because it means we can convert basic implications into clauses.

\[
\{s \Leftarrow (p \land q \land r)\} = s \lor \neg (p \land q \land r) = s \lor \neg p \lor \neg q \lor \neg r
\]

Here is a simple illustration:

\[
\{\text{rose} (X) \Leftarrow (\text{red} (X) \land \text{flower} (X))\} = \text{rose} (X) \lor \neg (\text{red} (X) \land \text{flower} (X)) = \text{rose} (X) \lor \neg \text{red} (X) \lor \neg \text{flower} (X)
\]

An implication can be converted directly into a Horn clause, provided:

- the consequent consists of a single positive literal;
4.3. RESOLUTION

- the antecedent consists of a conjunction of positive literals.

In fact, such a structure is exactly the nonunit clause type described above (the rules).

An implication with disjunctions in its antecedent is equivalent to a conjunction of implications. Thus, implications with disjunctive antecedents can be converted to a set of Horn clauses (a program).

\[
\begin{align*}
\{ s \leftarrow (p \lor q \lor r) \} &= (s \leftarrow p) \land (s \leftarrow q) \land (s \leftarrow r) \\
&= (s \lor \neg p) \land (s \lor \neg q) \land (s \lor \neg r)
\end{align*}
\]

4.3.3 Resolution with Horn Clauses

Because we have a direct connection between implications and Horn clauses, we can apply the same kind of deductive reasoning discussed earlier with respect to first order logic. Recall modus tollens, commonly expressed as \((\neg q \land (p \Rightarrow q)) \Rightarrow \neg p\). With Horn clauses, \((p \Rightarrow q)\) is expressed as \((q \lor \neg p)\). Modus tollens can be expressed appropriately for Horn clauses as \((\neg q \land (q \lor \neg p)) \Rightarrow \neg p\). We can prove conclusions from propositions consisting of Horn clauses. Given the truth of the Horn clauses \(\neg q\) and \((q \lor \neg p)\), we can conclude, via this version of modus tollens, the truth of the Horn clause \(\neg p\).

A traditional “forward” proof of a theorem applies inference rules to axioms in order to derive other true propositions, and ultimately derives the theorem. The challenge for automatic theorem proving is how to find the right combination of axioms and inference rules. It often isn’t obvious what a proof should start with. The proof technique of resolution attempts to get around this problem by working backwards: it starts with the statement you want to prove, and tries to construct a proof backwards towards the axioms. In particular, resolution is a form of proof by contradiction. Resolution starts by assuming (to the contrary) that the desired statement is false, and then attempts to derive a contradiction between this assumption and the axioms.

Resolution attempts to derive a contradiction by a series of steps, called resolution steps, each of which is a particular application of the Horn clause version of modus tollens. A query consists of a disjunction of negative literals, such as \(\neg l_1 \lor \neg l_2\). Each definite Horn clause in the database has exactly one positive literal, such as \(l_1 \lor \neg l_3\). A resolution step takes the current query, considers one of the negative literals, and looks for a definite clause in the program that contains the corresponding positive literal. It then resolves the two to form a new query. If we resolve \(\neg l_1 \lor \neg l_2\) and \(l_1 \lor \neg l_3\), modus tollens applies to the \(\neg l_1\) part of the original query, combined with the definite clause \(l_1 \lor \neg l_3\), to conclude \(\neg l_3\). This conclusion takes the place of \(\neg l_1\) in the original query, forming the new query \(\neg l_3 \lor \neg l_2\). Notice that the new query is indeed a query: a disjunction of negative literals. This will always be the result of a resolution step: because each definite clause contains only one positive literal, and it is that term that is “canceled” with the term of the query, what is left to be added to the query will only be any negative literals in the definite clause.
The number of terms in a query can reduce when a query is resolved with a fact. If we resolve the query $\neg l_1 \lor \neg l_2$ with the fact $l_1$, the resulting query, $\neg l_2$, contains one less literal. The most important instance of this is when a query of only one literal is resolved with a fact. If we resolve $\neg l_4$ with $l_4$, no terms remain: the resulting query is empty. This is the indication that a contradiction has been reached. In order for resolution to derive an empty query, it must have just resolved two directly contradictory clauses (in the example, $\neg l_4$ and $l_4$). Because both of the clauses being resolved were assumed to be true, this is the desired contradiction, and one can correctly conclude that the original statement (the negation of the starting query) is provable from the program.

Suppose we have a program of axioms like the following:

$$
\begin{array}{l}
parent(frank) \\
father(frank) \lor \neg parent(frank) \lor \neg male(frank) \\
male(frank)
\end{array}
$$

If we want to prove the statement $father(frank)$ with resolution, we would proceed as follows:

<table>
<thead>
<tr>
<th>Step</th>
<th>Formula</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\neg father(frank)$</td>
<td>Starting query.</td>
</tr>
<tr>
<td>1</td>
<td>$\neg father(frank) \land [father(frank) \lor \neg parent(frank) \lor \neg male(frank)]$</td>
<td>Select a matching positive literal.</td>
</tr>
<tr>
<td>2</td>
<td>$[\neg parent(frank) \lor \neg male(frank)]$</td>
<td>Resolution step.</td>
</tr>
<tr>
<td>3</td>
<td>$[\neg parent(frank) \lor \neg male(frank)] \land parent(frank)$</td>
<td>Select matching literal.</td>
</tr>
<tr>
<td>4</td>
<td>$\neg male(frank)$</td>
<td>Resolution step.</td>
</tr>
<tr>
<td>5</td>
<td>$\neg male(frank) \land male(frank)$</td>
<td>Select matching literal.</td>
</tr>
<tr>
<td>6</td>
<td>empty</td>
<td>Inconsistency!</td>
</tr>
</tbody>
</table>

There is a directedness to the search, which is permitted by the fact that Horn clauses can have at most one positive literal.

Because the database of definite Horn clauses cannot be inconsistent itself, the inconsistency must be the result of the addition of the query (the negation of the statement we wish to prove). Because the negation of the statement leads to a contradiction, the statement itself must follow from the database. The steps that directly lead to the contradiction can then be viewed as constituting a positive proof, by taking them in the reverse order:

<table>
<thead>
<tr>
<th>Step</th>
<th>Formula</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$male(frank)$</td>
<td>3rd axiom</td>
</tr>
<tr>
<td>1</td>
<td>$male(frank) \land parent(frank)$</td>
<td>1st axiom</td>
</tr>
<tr>
<td>2</td>
<td>$[male(frank) \land parent(frank)] \land [\neg parent(frank) \lor \neg male(frank)]$</td>
<td>2nd axiom</td>
</tr>
<tr>
<td>3</td>
<td>$father(frank)$</td>
<td>modus ponens</td>
</tr>
</tbody>
</table>

It should now be clear why negative clauses are called queries: they are only introduced as the negations of statements one wishes to prove. Programs always consist of definite Horn clauses and can never be inconsistent on their own. The possibility of inconsistency only arises when a negative clause, a query, is added. Note that this assumes that the statements we actually want to prove are restricted to conjunctions of positive literals. Thus, in this form of resolution, you quite literally cannot “prove a negative.”
4.4 Prolog Basics

4.4.1 Notation

Constants and predicates in prolog are expressed with lowercase letters and numbers. Terms are expressed with the familiar parenthesis notation. A clause is ended with a period. Facts, which are definite Horn clauses consisting of only a positive literal, are expressed with the single term followed by a period.

Example 4.3 Example facts in prolog.

\[
\text{bark(fido).}
\]
\[
\text{bark(sam).}
\]
\[
\text{meow(missy).}
\]

Rules are expressed by having a special symbol separating the positive literal from the negative ones. The symbol is actually a combination of two keyboard symbols, a colon and a dash: “:-”. The negative literals are separated from each other by commas, and the clause is ended with a period.

Example 4.4 Example rules in prolog.

\[
\begin{array}{ll}
\text{Prolog notation} & \text{Logical interpretation} \\
\text{dog(fido) :- furry(fido), bark(fido).} & \text{dog(fido) \lor \neg furry(fido) \lor \neg bark(fido)} \\
\text{cat(fido) :- furry(fido), meow(fido).} & \text{cat(fido) \lor \neg furry(fido) \lor \neg meow(fido)}
\end{array}
\]

There is another way to think about rules as you are working with them, and that is the implicational form. Consider the first rule in example 4.4, \text{dog(fido) :- furry(fido), bark(fido).}. If we are thinking about resolution, the logical notation \text{dog(fido) \lor \neg furry(fido) \lor \neg bark(fido)} is the easiest to work with, as it emphasizes the positive/negative literal distinction. But if you want to think to yourself what the rule “means”, it can be easier to convert it to implicational logical notation: \text{dog(fido) \leftarrow (furry(fido) \land bark(fido))}. In language, “fido is a dog if fido is furry and fido barks.”

4.4.2 Databases and Queries

Prolog operates by having a reference set of definite Horn clauses, called the database. You (the user) get to control the contents of the database, and changing the contents changes the system’s behavior. Earlier, a set of definite horn clauses was referred to as a program; the database for prolog at a given time is the particular program being used at that time. We will distinguish the two terms so that we can refer to arbitrary collections of definite Horn clauses as programs; it may be convenient at times to think of prolog’s database as consisting of a combination of several programs. Prolog’s reason for existence is to determine if a given statement can be proven, using the contents of the database. Note that technically prolog cannot conclude that a statement is false; it can only conclude that the statement cannot be proven true with its database.

Prolog jumps into action whenever it is given a query. The query is the negation of the statement whose provability we want to determine (recall the
discussion of Horn clauses above, where a query was characterized as a negative clause). Actually, when we interact with prolog, we only need to give the statement itself that we want to prove; prolog will automatically form the “proper” query by taking the negation. It takes the negation so that it can use resolution to determine if the original statement is provable.

Consider the following database:

\[
\begin{align*}
\text{bark}(\text{fido}). \\
\text{bark}(\text{sam}). \\
\text{meow}(\text{missy}).
\end{align*}
\]

If we pose the following queries to prolog using that database, the indicated responses will be given.

\[
\begin{align*}
?\text{- bark}(\text{fido}). & \quad \text{yes} \\
?\text{- bark}(\text{missy}). & \quad \text{no}
\end{align*}
\]

### 4.4.3 Search in Prolog

The description of resolution given in section 4.3 goes a long way towards indicating how a theorem-proving algorithm would work. But it does not explain everything. In particular, it does not explain:

1. the order in which the terms of a query are considered for resolution.
2. the order in which a database is searched when looking for clauses to resolve with a query.

When presented with a query containing multiple terms, prolog processes them from left to right. This is an arbitrary convention. Thus, if you pose a query with multiple terms, you can control the order in which the terms are resolved, by choosing how to order the terms in your query. The more interesting question is how prolog handles the construction of a new query after a resolution step. Prolog answers this by using what is known as depth-first search. If resolution with a clause from a database adds any new terms to the query, those terms are added on the left, so that they are evaluated first, before the other terms of the prior query. Thus if the query \(~\text{cat}(\text{missy}) \lor \neg \text{home}(\text{missy})\) is resolved with the rule \(\text{cat}(\text{missy}) \lor \neg \text{furry}(\text{missy}) \lor \neg \text{meow}(\text{missy})\), the resulting query will be \(\neg \text{furry}(\text{missy}) \lor \neg \text{meow}(\text{missy}) \lor \neg \text{home}(\text{missy})\), with the new terms from the rule added to the front of the query. This way, the new things just introduced will be further resolved first (if possible).

The order in which the database is searched depends heavily on the order inherent in the database. Some of the conditions that can affect the order of clauses in prolog’s database are implementation-dependent. However, if you type some clauses into a file, and then load that file into prolog using the \texttt{consult()} predicate, the clauses will have the same order in the database that they do in your file. Prolog searches the database from top to bottom, looking for a clause with a positive literal that matches (is the negation of) the first term of the current query.
4.4. PROLOG BASICS

The more complex part of the search involves what happens when more than one clause matches the query term. Prolog picks the first one in the database and resolves, but what if that ultimately fails, whereas the later clause would lead to a successful proof? To be sure, prolog needs to ultimately attempt each of them. This is accomplished with a technique known as backtracking. When prolog selects a clause from the database for resolution, it makes a note of which clause it used. If that attempt fails, prolog then resumes the search from that point in the database, looking for another clause to resolve with. If it reaches the end of the database, it must give up.

The place where prolog makes a note of which clause it tried is called the backtrack stack. An entry on the stack, which we will call a backtrack state, includes a copy of the prior query, and the number of the clause it was resolved with. That way, if one resolution of the first query term leads to failure, prolog can backtrack to the query at that point, and look for a different clause to resolve with the query term.

Example 4.5 The pet database (no variables version).

<table>
<thead>
<tr>
<th>#</th>
<th>Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bark(fido).</td>
</tr>
<tr>
<td>2</td>
<td>meow(missy).</td>
</tr>
<tr>
<td>3</td>
<td>furry(fido).</td>
</tr>
<tr>
<td>4</td>
<td>furry(missy).</td>
</tr>
<tr>
<td>5</td>
<td>dog(fido) :- furry(fido), bark(fido).</td>
</tr>
<tr>
<td>6</td>
<td>dog(missy) :- furry(missy), bark(missy).</td>
</tr>
<tr>
<td>7</td>
<td>cat(fido) :- furry(fido), meow(fido).</td>
</tr>
<tr>
<td>8</td>
<td>cat(missy) :- furry(missy), meow(missy).</td>
</tr>
<tr>
<td>9</td>
<td>pet(fido) :- dog(fido), home(fido).</td>
</tr>
<tr>
<td>10</td>
<td>pet(missy) :- dog(missy), home(missy).</td>
</tr>
<tr>
<td>11</td>
<td>pet(fido) :- cat(fido), home(fido).</td>
</tr>
<tr>
<td>12</td>
<td>pet(missy) :- cat(missy), home(missy).</td>
</tr>
<tr>
<td>13</td>
<td>home(missy).</td>
</tr>
</tbody>
</table>

Example 4.5 shows the pet database, with 13 clauses. Suppose we provide prolog with this database. We want to consider how prolog responds to some queries.

Example 4.6 ?- meow(missy).

Prolog searches the database for a matching clause. Clause 2 matches, so prolog puts 2 on the backtrack stack, and resolves clause 2 with the query. The resulting query is empty. This indicates that a contradiction has been found, and prolog concludes that the meow(missy) can be proven. Prolog returns Yes.

Example 4.7 ?- meow(fido).

Prolog searches the database for a matching clause. None can be found in the database, so no contradiction can be derived. Prolog returns No.
Example 4.8 \(-\) \(\text{dog(fido)}\).

The first matching clause is \#5. Prolog puts 5 on the backtrack stack, and resolves. The resulting query is \(\text{furry(fido), bark(fido)}\). We can represent this point in the computation with

<table>
<thead>
<tr>
<th>Query</th>
<th>0 furry(fido), bark(fido)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>5 dog(fido)</td>
</tr>
</tbody>
</table>

Prolog then looks for a match for the first term of the query, starting from the top of the database. The \#0 at the beginning of the query indicates that the search for a match for the first term will start after clause \#0, that is, the search will start at the beginning of the database. The first match it finds is \#3, so it puts 3 on the backtrack stack, and resolves.

<table>
<thead>
<tr>
<th>Query</th>
<th>0 bark(fido)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>5 furry(fido), bark(fido)</td>
</tr>
</tbody>
</table>

Prolog then looks for a match for \(\text{bark(fido)}\). The first match is \#1, so it puts 1 on the backtrack stack, and resolves. This results in an empty query:

<table>
<thead>
<tr>
<th>Query</th>
<th>1 bark(fido), furry(fido), bark(fido)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>5 furry(fido), bark(fido)</td>
</tr>
</tbody>
</table>

Thus, prolog returns \textbf{Yes}.

As that example shows, the backtrack stack can accumulate several backtrack states. Thus far, the stack may not seem to have had much purpose. But look at the next example.

Example 4.9 \(-\) \(\text{pet(missy)}\).

The first matching clause is \#10. Prolog marks the backtrack stack, and resolves.

<table>
<thead>
<tr>
<th>Query</th>
<th>0 dog(missy), home(missy)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>10 pet(missy)</td>
</tr>
</tbody>
</table>

Prolog now looks for a match for \(\text{dog(missy)}\). The first match is \#6:

<table>
<thead>
<tr>
<th>Query</th>
<th>0 furry(missy), bark(missy), home(missy)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>6 dog(missy), home(missy)</td>
</tr>
</tbody>
</table>

The first match for the next term is \#4:

<table>
<thead>
<tr>
<th>Query</th>
<th>0 bark(missy), home(missy)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>4 furry(missy), bark(missy), home(missy)</td>
</tr>
</tbody>
</table>

Prolog now looks for a match for \(\text{bark(missy)}\). No matches exist in the database, so this attempt fails. Prolog responds by backtracking: it pulls off the top entry on the backtrack stack, restores the query from the entry, and resumes the search for a match for the first term, starting from just past the position in the entry from the stack. Thus, the restored query is \(\text{furry(missy), bark(missy), home(missy)}\), and prolog resumes the search for a clause matching \(\text{furry(missy)}\) just after clause \#4 in the database.
4.4. PROLOG BASICS

Query 4 furry(missy), bark(missy), home(missy)
Stack 10 pet(missy)

However, there is not another match for furry(missy) in the database, so that term fails also. Prolog now backtracks again, pulling the next entry off of the stack, restoring that query, and resuming the search for the first term, dog(missy), after clause 6.

Query 6 dog(missy), home(missy)
Stack 10 pet(missy)

That search also fails, so prolog backtracks again, pulling the next (and only remaining) entry off of the stack, restoring the goal, and resuming the search for a match for pet(missy) after clause 10.

Query 10 pet(missy)
Stack

This time, there is another match for the goal: clause 12. So, prolog puts #12 and the query on the stack, and resolves with clause 12:

Query 0 cat(missy), home(missy)
Stack 12 pet(missy)

The first match for the first term is #8:

Query 0 furry(missy), meow(missy), home(missy)
Stack 12 cat(missy)

Each of the three terms in this query match facts in the database:

Query 0 meow(missy), home(missy)
Stack 12 furry(missy)

Query 0 home(missy)
Stack 12 meow(missy)

Query 13 home(missy)
Stack 12 pet(missy)

The desired contradiction has been reached, and prolog answers Yes.

4.4.4 Exercises

Exercise 4.4.1 Enter the Pet database into prolog, and try some queries. Do you get the answers that you expect?
Exercise 4.4.2 In prolog, after loading the Pet database, enter the query `trace`. Then enter the query `pet(missy)`. Repeatedly press return, and observe the output, comparing it to the discussion of example 4.9.

4.5 Database Prolog

4.5.1 Variables

In addition to constants like `fido` and `missy`, prolog permits variables. Prolog distinguishes variables by having them begin with an uppercase letter (all constants begin with lowercase letters). Prolog uses a subset of FOL, so variables can be substituted for arguments of predicates, but they cannot be substituted for predicates by themselves: `bark(X)` is a valid term, but `X(fido)` is not.

Giving prolog a query containing variables makes a more complex demand: find, if possible, assignments of values to the variables that result in a provable statement.

```
?- meow(Beast).
Beast = missy
yes
```

When prolog finds variable bindings, you can tell prolog you are satisfied by pressing Enter, and it will return yes (as immediately above). Alternatively, you can press `;` (semicolon), which tells prolog to search for other variable bindings resulting in provable statements.

```
?- furry(Beast).
Beast = fido;
Beast = missy;
no
```

Prolog is able to continue the search for other variable values because it has not yet deleted its backtrack stack. When you press `;`, prolog continues searching by taking the top backtrack state from the stack, and continuing from there, just as it would have if it had encountered a failed goal.

4.5.2 Search with Variables

Queries with variables

When you give prolog a query with a variable, it must search for a match in the database. But, with a variable, the notion of a match is more complex: for each clause in the database, prolog must ask, “is there a value I can bind to the variable in order to make the two terms literally match?”

Consider the query `furry(X)`, used with the Pet database. To find a match, it must find a positive literal in the database with the same predicate. The first one in the Pet database is clause #3, `furry(fido)`. If the value `fido` is substituted for the variable `X`, then the two terms match. In this case, that
exhausts the query, so prolog returns a Yes, but with an indication of the variable binding used to achieve the proof: \( X = \textit{fido} \).

**Rules with variables**

Variables may also be used in the clauses appearing in the database.

**Example 4.10** \( \text{dog}(X) \leftarrow \text{furry}(X), \text{bark}(X) \).

The rule in example 4.10 has three occurrences of the variable \( X \). As you would expect, prolog will require that all three instances of the variable remain identical; if a value is substituted for one instance of the variable, the same value must be substituted for other instances within the same clause. The same does not apply across clauses in a database: there is no significance to the use of the same variable name in different clauses. The reason is that each definite Horn clause should be understood to implicitly include a universal quantifier for each variable. Thus, the logical form for the rule in example 4.10 is \( \forall X \left[ \text{dog}(X) \lor \neg \text{furry}(X) \lor \neg \text{bark}(X) \right] \). The quantifier asserts that the rule can apply to any possible value, and restricts the scope of that occurrence of the variable to within the clause. If a definite Horn clause contains more than one distinct variable (variables with different names), then there will be a separate universal quantifier for each.

When resolution matches a query term with a variable-containing rule via substitution of a value for the variable, that value is substituted for the other occurrences of the variable prior to construction of the new query. If we have a query of \( \text{dog} (\text{fido}) \) and we resolve it with the rule in example 4.10, the resulting new query is \( \text{furry}(\text{fido}), \text{bark}(\text{fido}) \).

Rules with variables in prolog gives us the power that FOL gets from variable-containing expressions with universal quantification: a relationship between predicates can be stated once, and it applies to all possible argument values. We can now construct a new version of the Pet database, using rules with variables.

**Example 4.11** *The Pet database (with variables).*

<table>
<thead>
<tr>
<th>#</th>
<th>Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \text{bark}(\text{fido}). )</td>
</tr>
<tr>
<td>2</td>
<td>( \text{meow}(\text{missy}). )</td>
</tr>
<tr>
<td>3</td>
<td>( \text{furry}(\text{fido}). )</td>
</tr>
<tr>
<td>4</td>
<td>( \text{furry}(\text{missy}). )</td>
</tr>
<tr>
<td>5</td>
<td>( \text{dog}(X) \leftarrow \text{furry}(X), \text{bark}(X). )</td>
</tr>
<tr>
<td>6</td>
<td>( \text{cat}(X) \leftarrow \text{furry}(X), \text{meow}(X). )</td>
</tr>
<tr>
<td>7</td>
<td>( \text{pet}(X) \leftarrow \text{dog}(X), \text{home}(X). )</td>
</tr>
<tr>
<td>8</td>
<td>( \text{pet}(X) \leftarrow \text{cat}(X), \text{home}(X). )</td>
</tr>
<tr>
<td>9</td>
<td>( \text{home}(\text{missy}). )</td>
</tr>
</tbody>
</table>

Notice that this version of the Pet database is shorter; the variables allow a single statement of a relationship between predicates that earlier was separately stated for each potential argument.
4.5.3 Recursive Rules

The following is a prolog program for summing the first N numbers (1...n).

**Example 4.12** Program for the sum of the first N integers.

\[
\text{sum}(1,1).
\]

\[
\text{sum}(N,\text{Sum}) :\neg N>1, \text{M is } N-1, \text{sum}(M,\text{Msum}), \text{Sum is } N+\text{Msum}.
\]

The second clause is an example of a recursive rule: the rule for `sum()` is defined in terms of `sum()`. Some special operator notation is used here, following prolog practice: in N>1, the predicate is “>”, with N in the position of the first argument and 1 in the position of the second argument. This expression of “>” would be equivalent to a predicate >\((X,Y)\) in the more traditional notation, returning true if X and Y are bound to numeric values and the value of X is greater than the value of Y. The function “is” is more exceptional in its behavior (see exercise 4.5.7): it evaluates the expression after the “is” to a single number, and then binds the variable preceding the “is” to that number, failing if the variable is already bound to some other value.

4.5.4 Exercises

**Exercise 4.5.1** Using the new version of the Pet database (the one with variables), work through the step-by-step behavior of prolog for the query `cat(Beast)`.

**Exercise 4.5.2** Using the new version of the Pet database (the one with variables), work through the step-by-step behavior of prolog for the query `dog(missy)`.

**Exercise 4.5.3** Using the new version of the Pet database (the one with variables), work through the step-by-step behavior of prolog for the query `pet(missy)`.

**Exercise 4.5.4** Load the following program into prolog. The predicate `advisor(faculty, student)` means that `faculty` was the advisor of `student`.

```
% The advisor database.
advisor(halle, prince).
advisor(halle, mccarthy).
advisor(halle, hayes).
advisor(prince, ito).
advisor(mccarthy, alderete).
advisor(smolensky, tesar).
advisor(hayes, zuraw).

eastcoast(halle).
eastcoast(prince).
eastcoast(mccarthy).
eastcoast(alderete).
eastcoast(smolensky).
```
4.6. PURE PROLOG

\begin{verbatim}
eastcoast(tesar).
eastcoast(zuraw).
westcoast(hayes).
westcoast(ito).
\end{verbatim}

1. Let \texttt{ecadvisor} be true of an individual who is both an advisor and resides on the east coast. Create a prolog predicate expressing this in terms of the predicates \texttt{advisor} and \texttt{eastcoast}.

2. Add your definition to the prolog file, load it into prolog, and test it out. Does it give the proper answers?

\textbf{Exercise 4.5.5} Consider the following definition for a predicate \texttt{acadancestor}(X,Y), meaning that \(X\) is the academic ancestor of \(Y\). Type it into a file and consult it along with the Advisor DB file. Based upon your understanding of the contents of the database, construct and test two queries, one that should succeed and one that should fail. Are the answers correct? Why?

\begin{verbatim}
acadancestor(Old,Young) : - advisor(Old,Young).
acadancestor(Old,Young) : - advisor(Old,Mid), acadancestor(Mid,Young).
\end{verbatim}

\textbf{Exercise 4.5.6} Write a prolog program to compute the factorial of a number. The call should look like \texttt{factorial(5,Ans)} where the response will have variable \texttt{Ans} bound to the value of 5!.

\textbf{Exercise 4.5.7} What is the difference between \texttt{X is 5-2} and \texttt{X = 5-2} in prolog (try it out and see, then explain)?

4.6 Pure Prolog

4.6.1 Unification

Unification is the process of instantiating the variables of two expressions with values such that the expressions become the same. Unification is a nontrivial topic because of the possibility of compound terms, created by the permission of function symbols in the construction of terms. A substitution is an assignment of values to variables. A substitution is applied to an expression by replacing each occurrence of a substituted variable with its value.

\textbf{Example 4.13} Substituting \(a\) for \(X\): \(p(X)\{X/a\} = p(a)\)

\textbf{Example 4.14} Substituting \(f(Y)\) for \(X\): \(p(X,Y)\{X/f(Y)\} = p(f(Y),Y)\)

\textbf{Example 4.15} Substituting \(a\) for \(X\), and \(g(a,b)\) for \(Z\), leaving \(Y\) unsubstituted: \(p(X,Y,f(X,Z))\{X/a,Z/g(a,b)\} = p(a,Y,f(a,g(a,b)))\)
A unifier is a substitution for variables of two expressions that unifies them.

**Example 4.16** \( p(a, X) \) and \( p(Y, b) \) are unified by \( \{X/b, Y/a\} \).

**Example 4.17** \( p(a) \) and \( q(X) \) are not unifiable, because the top-level predicates \( p \) and \( q \) do not match.

Sometimes there may be more than one unifier for a unifiable pair of expressions. One substitution \( S_1 \) is more general than another \( S_2 \) for an expression \( E \) if \( E \{S_2\} \) can be arrived at by applying some further substitution to \( E \{S_1\} \). The most general unifier (mgu) of a pair of expressions is that substitution which is a unifier, and is more general than any other unifier.

**Example 4.18** \( p(X, b, Z) \) and \( p(a, Y, Z) \) are unified by both \( \{X/a, Y/b\} \) and \( \{X/a, Y/b, Z/c\} \), but \( \{X/a, Y/b\} \) is more general.

### The “Occurs” Check

Consider attempting to unify the pair of expressions \( p(X) \) and \( p(q(X)) \). A simple approach would note the matching predicates, and simply unify the \( X \) of the first expression with whatever the argument is of the second expression, that is, unify with \( \{X/q(X)\} \). Try making this substitution back into the two expressions; are they now the same?

\[
\begin{align*}
p(X) & \rightarrow p(q(X)) \\
p(q(X)) & \rightarrow p(q(q(X)))
\end{align*}
\]

The problem can be traced directly to the fact that we are trying to substitute, for the variable \( X \), an expression which itself contains \( X \). Such a move will never lead to success. Thus, when attempting a unification, one must check proposed substitutions to make sure that the variable does not occur in the expression being substituted for it. This is known as **the occurs check**.

### 4.6.2 Lists

A list is a finite sequence of terms. In prolog, a list is surrounded by square brackets, with the elements separated by commas. The first element in a list is called the **head**, and the list of all elements except the head is the **tail**.

**Example 4.19** \([a, b, c, d]\) is a list with four elements. The head of the list is \( a \). The tail of the list is the list \([b, c, d]\).

**Example 4.20** \([\ ]\) is the empty list; it is the sole list containing no elements.

The decomposition into head and tail can be more directly represented, and it is often convenient to do so. The separation between the head and the tail is denoted with a vertical bar.

**Example 4.21** \( [H|T] \) unifies with \( [a, b, c, d] \) via \( \{H/a, T/[b, c, d]\} \).
4.6.3 Soundness and Completeness of Inference

An inference procedure is sound if and only if any sentence derivable by the inference procedure from the database is in fact logically implied by the database. An inference procedure is complete if and only if any sentence logically implied by the database is in fact derivable by the inference procedure from the database.

While resolution proper is sound, the form realized in most implementations of prolog is not, because the occurs check is often left out of the unification procedure for computational efficiency reasons. Resolution itself is not complete; there are provable statements that resolution will not derive.

4.6.4 Exercises

For the next five exercises, determine if the pair of expressions can be unified. If so, find the most general unifier for the pair. If not, explain.

Exercise 4.6.1 \( p(X) \quad p(a) \)
Exercise 4.6.2 \( p(X) \quad q(a) \)
Exercise 4.6.3 \( p(X, Y) \quad p(f(Z), a) \)
Exercise 4.6.4 \( p(X, Y) \quad p(a) \)
Exercise 4.6.5 \( p(X, Y) \quad p(f(Y), X) \)

Exercise 4.6.6 Consider the following prolog rule: \( \text{containsbruce([bruce|T])} \). What does it mean? What sorts of arguments will be true of the predicate \( \text{containsbruce} \), according to this clause? You may want to enter it into prolog, and test it out.

Exercise 4.6.7 Consider the following prolog rule: 
\( \text{containsbruce([H|T]):-containsbruce(T)} \). What does it mean? What sorts of arguments will be true of the predicate \( \text{containsbruce} \), according to this clause? You may want to enter it into prolog, and test it out.

Exercise 4.6.8 Write a prolog program defining a one-argument predicate, \( \text{containsbruce()} \), that is true precisely when its argument is a list, and the constant \text{bruce} appears at least once in that list.

Exercise 4.6.9 Give your program from the previous exercise the query \( \text{containsbruce(bruce)} \). What answer does it give? Why?
CHAPTER 4. PROOF

4.7 Parsing with Prolog

4.7.1 Representing String Positions with Difference Lists

It would be convenient to be able to represent an input sentence as part of the query, rather than having to add facts to the database to represent the sentence (and retract any facts representing a previous input sentence). Lists allow us to represent a sequence. If we capitalize on lists and unification, we can represent the structure of the input logically. The standard way to do this is with difference lists. A pair of difference lists identifies a substring as that portion which is in the first of the lists, but not the second (it is the difference between the two).

Consider the example sentence “the dog barks”. With difference lists, we would represent the first word with the pair ([the,dog,barks],[dog,barks]). By convention, we will always have the two lists differ only in what is on the left-hand side of the first list.

Here is how the position of each of the three words in our example sentence are represented:

<table>
<thead>
<tr>
<th>Word</th>
<th>First List</th>
<th>Second List</th>
</tr>
</thead>
<tbody>
<tr>
<td>the</td>
<td>[the,dog,barks]</td>
<td>[dog,barks]</td>
</tr>
<tr>
<td>dog</td>
<td>[dog,barks]</td>
<td>[barks]</td>
</tr>
<tr>
<td>barks</td>
<td>[barks]</td>
<td>[ ]</td>
</tr>
</tbody>
</table>

The second list for each word consists of the rest of the sentence after that word. Each position is, in a sense, represented by its complete right context. Doing it in this way allows us to take advantage of the head/tail structure of lists.

The difference list approach easily generalizes to phrases:

<table>
<thead>
<tr>
<th>Phrase</th>
<th>First List</th>
<th>Second List</th>
</tr>
</thead>
<tbody>
<tr>
<td>the dog</td>
<td>[the,dog,barks]</td>
<td>[barks]</td>
</tr>
<tr>
<td>the dog barks</td>
<td>[the,dog,barks]</td>
<td>[ ]</td>
</tr>
</tbody>
</table>

4.7.2 Grammar Rules as Prolog Rules

Grammar rewrite rules can be represented as logical statements about a sentence.

\[ S \rightarrow NP \ VP \]

can be represented in the database with the following rule:

\[ s(P0,P) :- np(P0, P1), \ vp(P1, P). \]

In the above rule, P0, P1, and P refer to difference lists. So the rule states: “the string that is the difference between string P0 and string P is a sentence if the substring that is the difference between P0 and P1 is an NP, and the substring that is the difference between P1 and P is a VP.” We can instantiate the variables to represent the fact that “the dog barks” is a sentence as follows:
Individual words and their types can be represented in the database as follows:

\[
det([\text{the}|\text{Rest}], \text{Rest}).
\]
\[
noun([\text{dog}|\text{Rest}], \text{Rest}).
\]

The rule needed to describe an NP as a determined followed by a noun would be:

\[
np(P_0, P) :- det(P_0, P_1), noun(P_1, P).
\]

A query to this database of the form \(np([\text{the}, \text{dog}, \text{barks}], [\text{barks}])\) will be provable now.

We show where a phrase is in the sentence by giving everything that follows it. This makes it easy to parse left-to-right, because the representation of each phrase contains the rest of the sentence following the phrase.

### 4.7.3 Constructing Tree Representations

When given a valid sentence, we would like our parser to do more than just reply “Yes.” We want it to give us the syntactic analysis that is the basis for the judgment. We can take advantage of the recursive way in which prolog operates, along with the nature of unification, and get prolog to construct the parse tree as a side effect of proving the sentence grammatical.

Consider the following prolog clauses.

\[
np(np(DET, NOUN), P_0, P) :- det(DET, P_0, P_1), noun(NOUN, P_1, P).
\]
\[
noun(noun(dog), [dog|Rest], Rest).
\]

Compared to the previous section, the predicates have an additional argument added (the first argument). The first argument is intended to unify with a tree representation of the constituent. Our trees will be recursively nested predicate structures. The tree for the sentence “The dog barks.” will be something like

\[
s(np(det(the), noun(dog)), vp(verb(barks))).
\]

These structures can be built compositionally. The base cases are the individual words. Note that the fact for the noun “dog” returns, as its tree structure, \(\text{noun(dog)}\). The tree structure for the NP rule indicated composes the tree representation out of the representations for the satisfying determiner and noun, \(np(DET, NOUN)\).

Now, if we want to parse the sentence “the dog barks”, we give the query
s(Tree, [the,dog,barks],[]).

If the program is able to parse the sentence, that means it is able to find a value to bind to the variable Tree to create a provable statement. That value will be the parse tree. Thus, instead of just answering yes, Prolog gives the value for Tree that makes it provable. For us, this is equivalent to having Prolog construct the parse tree. We rig the statements so that Prolog, as a kind of side effect, happens to construct and return a parse tree while proving that the string is a sentence.

4.8 Mathematical Induction

4.8.1 Summing Numbers (Again)

Prove: $1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$.

The prolog summing procedure could be used to construct a proof for any particular value of $n$. But how can you prove the formula correct for all counting numbers, without constructing an infinite number of proofs?

The Principle of Mathematical Induction

Let $S$ be a subset of $\mathbb{N}$ with the following properties:

(a) $1 \in S$.

(b) For all $n \in \mathbb{N}$, $(n - 1) \in S$ entails $n \in S$.

Then $S = \mathbb{N}$.

Theorem 4.1 $1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$

Proof. Base Case: $n = 1$: $1 = \frac{2}{2} = \frac{1(1+1)}{2}$. Correct.

Induction Hypothesis: $1 + 2 + \ldots + (n - 1) = \frac{(n-1)(n-1+1)}{2} = \frac{(n-1)n}{2}$.

$1 + 2 + 3 + \ldots + n = (1 + 2 + \ldots + n - 1) + n$

$= \frac{(n-1)n}{2} + n$

$= \frac{n^2 - n}{2} + \frac{2n}{2}$

$= \frac{n^2 + n}{2}$

$= \frac{n(n+1)}{2}$

By mathematical induction, $1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$ for all $n \in \mathbb{N}$. $\blacksquare$
4.8.2 Induction over Grammar Derivations

Mathematical induction is not limited to arithmetic formulas. Often, it can be applied to recursively defined spaces of structures.

**Theorem 4.2** Every string generated by the following grammar is of even length.

\[
\begin{align*}
\Sigma & \rightarrow X \\
X & \rightarrow a\ Y \\
Y & \rightarrow a\ X \mid a
\end{align*}
\]

**Proof.** Observe first that the only way to rewrite the start symbol is as X, so the derivation of every string has at least one occurrence of nonterminal X. We will base our induction upon the number of occurrences of X in the derivations of various strings.

Base Case: all strings having only one X in their derivation. There is only one such string, “a a”, with derivation \( \Sigma \Rightarrow X \Rightarrow a\ Y \Rightarrow a\ a \). The only step having a choice is the rewriting of Y, and the only alternative introduces a second X into the derivation. The string “a a” has even length.

Induction Hypothesis: All strings having \((n - 1)\) X’s in their derivation have even length.

Consider a string with \(n\) X’s in its derivation. The final X in the derivation must have resulted from the application of rule Y→a X. The alternative rule Y→a could have been chosen, giving a derivation with only \((n - 1)\) X’s. By the induction hypothesis, that resulting string would have been of even length. Therefore, at the introduction of the final X into the current derivation, there must be a string of an even number of occurrences of ‘a’, call that number L, followed by X.

\[a \ldots a\ X\]

The next two steps of the derivation are forced:

\[(a \ldots a)\ X \Rightarrow (a \ldots a)\ a\ Y \Rightarrow (a \ldots a)\ a\ a\]

The choice of the final rule is forced to avoid introducing yet another X into the derivation. A total of 2 additional symbols have been added to the string of length L, and since L is even, \((L+2)\) is even. Therefore, any string with \(n\) X’s in its derivation is of even length.

By mathematical induction, all strings with one or more X’s in their derivation have even length. As all strings generated by the grammar must have at least one X in their derivation, it follows that all have even length. ■

4.8.3 Exercises

**Exercise 4.8.1** Prove, using mathematical induction, that \[\frac{1}{1\cdot2} + \frac{1}{2\cdot3} + \ldots + \frac{1}{n(n+1)} = \frac{n}{n+1}.\]
Exercise 4.8.2 Prove, using mathematical induction, that \(1^2 + 2^2 + 3^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}\).

Exercise 4.8.3 Prove, using mathematical induction, that the total number of distinct subsets of a set with \(n\) elements is \(2^n\).
Chapter 5

Mathematical Analysis

5.1 Sequences and Series

5.1.1 Infinite Sequences

An infinite sequence is an ordered collection of numbers. More rigorously, it is a function whose domain is the positive integers. The positive integers, taken in increasing order, themselves form a sequence: the function is \( f(n) = n, \{1, 2, 3, \ldots\} \). The value \( f(n) \) is often called the \( n \)th term of the sequence. It should be emphasized that the formal definition of function as a mapping is what is intended here; it is not the case that an arbitrary sequence will always be representable with some simple, closed-form mathematical formula.

Example 5.1 The sequence \( f(n) = \frac{1}{n} \), also representable as \( \{1, \frac{1}{2}, \frac{1}{3}, \ldots\} \).

The numbers of this sequence are all positive, but the terms get successively smaller. This means that the terms are also getting successively closer to 0. The property of interest to us is that for any positive number \( \varepsilon \), no matter how small, there is a point in the sequence beyond which all of the terms are closer to zero than \( \varepsilon \). This means that the sequence has limit 0. It can also be said to converge on 0.

The distance between two real numbers is the absolute value of their difference: the distance between \( x \) and \( y \) is \( |x - y| \).

A sequence \( f(n) \) has the limit \( L \), denoted \( \lim_{n \to \infty} f(n) \), if, for every real number \( \varepsilon > 0 \), there exists an integer \( N \) such that whenever \( n > N \), \( |f(n) - L| < \varepsilon \).

Many sequences do not have a limit. They are sometimes called divergent sequences. A sequence which does have a limit is a convergent sequence.

Theorem 5.1 Let \( f(n) = 5 + \frac{1}{n^2}, \{6, 5\frac{1}{4}, 5\frac{1}{7}, \ldots\} \). \( \lim_{n \to \infty} f(n) = 5 \).

Proof. Let \( \varepsilon > 0 \) be an arbitrarily small real number. We want to find a \( N \) so that, for all members of the sequence after \( f(N) \), the sequence members are
closer to 5 than \( \epsilon \); in other words, \( n > N \) will guarantee that \(|f(n) - 5| < \epsilon\). All sequence members are greater than 5, and the sequence is strictly decreasing, so if we can find one member of the sequence less than \( \epsilon \) away from 5, we are guaranteed that all of the following members of the sequence are, also.

We can prove that this is always the case by giving a formula for \( N \) in terms of \( \epsilon \). For this sequence, let \( N = \lceil \sqrt{\frac{1}{\epsilon}} \rceil \) ([x] denotes the ceiling of x, meaning the smallest integer value that is larger than or equal to x). Then if \( n > N \),

\[
|f(n) - 5| = f(n) - 5 = 5 + \frac{1}{n^2} - 5 = \frac{1}{n^2} < \frac{1}{N^2} < \left( \sqrt{\frac{1}{\epsilon}} \right)^2 < \epsilon
\]

Thus, for any \( \epsilon > 0 \) there exists an \( N \) such that for all \( n > N \), \(|f(n) - 5| < \epsilon\).

How did we actually arrive at this useful formula \( N > \sqrt{\frac{1}{\epsilon}} \) that made the proof work out? By working the logic backwards (you actually do this part before constructing the proof). Suppose \(|f(N) - 5| < \epsilon\). We know that all numbers in the sequence are greater than 5, so it must be that \( 0 < f(N) - 5 < \epsilon \). Then,

\[
\left(5 + \frac{1}{N^2}\right) - 5 < \epsilon
\]

\[
\frac{1}{N^2} < \epsilon
\]

\[
1 < (\epsilon) N^2
\]

\[
\frac{1}{\epsilon} < N^2
\]

\[
\sqrt{\frac{1}{\epsilon}} < N
\]

Therefore, if \( n > \sqrt{\frac{1}{\epsilon}} \), \( f(n) - 5 < \epsilon \).

To see a particular case, suppose we choose \( \epsilon = 0.002 \). Then \( \sqrt{\frac{1}{\epsilon}} = \sqrt{\frac{1}{0.002}} = 22.361 \). So, we can choose \( N = 23 \): \( f(23) = 5 + \frac{1}{(23)^2} = 5.0019 \), so \(|f(23) - 5| = 5.0019 - 5 = 0.0019 < 0.002\).
5.2. INFINITE SERIES

Example 5.2 The sequence \( f(n) = (-1)^n, \{ -1, 1, -1, 1, ... \} \), is a divergent sequence.

The statement \( \lim_{n \to \infty} f(n) = \infty \) means that for every real number \( P > 0 \), there exists an integer \( N \) such that whenever \( n > N \), \( f(n) > P \).

Example 5.3 The sequence \( f(n) = 2^n, \{ 2, 4, 6, ... \} \) has limit \( \infty \).

Proof. Let \( P \) be a real number such that \( P > 0 \). Let \( N = [P] \). Whenever \( n > N \),

\[
\begin{align*}
  f(n) &= 2n \\
  &> 2N \\
  &> 2 \times [P] \\
  &> P
\end{align*}
\]

5.1.2 Exercises

Tell whether each of the following sequences has a limit, and if so, what the limit is.

Exercise 5.1.1 \( f(n) = \frac{n-1}{n}; \{ 0, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, ... \} \)

Exercise 5.1.2 \( f(n) = (-1)^n; \{ -1, \frac{1}{2}, -\frac{1}{3}, \frac{1}{4}, ... \} \)

Exercise 5.1.3 \( f(n) = n \cdot f(n-1), \) with \( f(1) = 1 \): \( \{ 1, 2, 6, 24, 120, ... \} \)

Exercise 5.1.4 \( f(n) = (-1)^n n, \) with \( f(1) = -1 \): \( \{ -1, 2, -3, 4, -5, ... \} \)

5.2 Infinite Series

5.2.1 Sequences of Partial Sums

Let \( f(i) \) be some infinite sequence. The sum

\[
\sum_{i=1}^{\infty} f(i) = f(1) + f(2) + f(3) + ... 
\]

is called a series. A series has an associated sequence of partial sums

\[
S_n = f(1) + f(2) + ... + f(n) = \sum_{i=1}^{n} f(i) 
\]
Example 5.4 The series $\sum_{i=1}^{\infty} \frac{1}{i^2}$ has the sequence of partial sums $\left\{ \frac{1}{2}, \frac{1}{2} + \frac{1}{4}, \frac{1}{2} + \frac{1}{4} + \frac{1}{8}, \ldots \right\}$, also representable as $\left\{ \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \ldots \right\}$.

An infinite series is said to converge if its sequence of partial sums converges. The limit of the partial sums for a convergent series is the sum of that series. The series is literally equal in value to the limit of its partial sums.

Example 5.5 The series $\sum_{i=1}^{\infty} \frac{1}{i^2} = 1$, because

$$\lim_{n \to \infty} S_n = \lim_{n \to \infty} \frac{2^n - 1}{2^n} = 1$$

The fact that $\sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{2^n - 1}{2^n}$ can be proven using mathematical induction.

Theorem 5.2 If an infinite series $\sum_{i=1}^{\infty} f(i)$ is convergent, then $\lim_{n \to \infty} f(n) = 0$.

The fact that a sequence of terms has limit 0 does not guarantee that the series of those terms is convergent.

Theorem 5.3 The harmonic series, $\sum_{i=1}^{\infty} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots$, is divergent.

Proof. Look at the following selected partial sums:

- $S_1 = 1$
- $S_2 = 1 + \frac{1}{2}$
- $S_4 = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4}$
- $\quad > 1 + \frac{1}{2} + \left( \frac{1}{4} + \frac{1}{4} \right) = 1 + \frac{1}{2} + \frac{1}{2}$
- $S_8 = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8}$
- $\quad > 1 + \frac{1}{2} + \left( \frac{1}{4} + \frac{1}{4} \right) + \left( \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} \right) = 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{2}$

In general, $S_{2^n} \geq 1 + n\left( \frac{1}{2} \right)$. So, the series eventually will overtake any positive number, once it adds up $\frac{1}{2}$ enough times. ■

5.2.2 Series with Positive and Negative Terms

A series converges absolutely if the series of the absolute values of the terms converges. If a series converges, but not absolutely, it converges conditionally.
Example 5.6 The series \( \sum_{i=1}^{\infty} \frac{(-1)^{i-1}}{i} \) converges conditionally.

The series does not converge absolutely, because the series of the absolute values is the harmonic series.

\[
\sum_{i=1}^{\infty} \frac{(-1)^{i-1}}{i} = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots
\]

\[
= \left[ 1 - \frac{1}{2} \right] + \left[ \frac{1}{3} - \frac{1}{4} \right] + \ldots
\]

\[
= \sum_{n=1}^{\infty} \left[ \frac{1}{2n-1} - \frac{1}{2n} \right]
\]

\[
= \sum_{n=1}^{\infty} \left[ \frac{1}{(2n-1)2n} \right]
\]

\[
= \sum_{n=1}^{\infty} \left[ \frac{1}{n^2 + (3n^2 - 2n)} \right]
\]

\[
< \sum_{n=1}^{\infty} \left[ \frac{1}{n^2} \right]
\]

All of the terms of the modified series (indexed with \( n \)) are greater than zero and less than \( \frac{1}{n^2} \). We assert without proof here that \( \sum_{n=1}^{\infty} \left[ \frac{1}{n^2} \right] \approx 1.64 \).

If a series converges conditionally, then the positive and negative “pieces” diverge: the series formed by adding all of the positive terms of the original series diverges, as does the series formed by adding all of the negative terms of the original series. The convergence of the series is the result of the positive and negative terms canceling each other out sufficiently, so that the differences sum to a finite value.

5.3 Derivatives

5.3.1 Rate of Change

Consider a car moving in a straight line. Suppose that in 2 minutes it has traveled 20 feet. Then the rate of change of the position of the car is 20 feet in 2 minutes, or 10 feet per minute.

The concept of rate of change is often used when describing lines plotted on Cartesian coordinates. Consider the line defined by the equation \( y = 2x + 1 \):
CHAPTER 5. MATHEMATICAL ANALYSIS

When \( x = 0 \), \( y = 1 \). When \( x \) moves from 0 to 1, \( y \) grows from 1 to 3. The slope of a line is the rate of change of the line. The slope of a line can be calculated from any two points on the line: if \((x_1, y_1)\) and \((x_2, y_2)\) are points on a line, the slope of the line is \(\frac{y_2 - y_1}{x_2 - x_1}\). For the line \(y = 2x + 1\), the slope is \(\frac{3 - 1}{1 - 0} = 2\).

Actually, the slope of the line can be read directly from the equation, if the equation is in the proper form. A linear equation is in slope-intercept form if it is written \(y = mx + b\), where \(m\) and \(b\) are real numbers. For an equation in that form, \(m\) is the slope of the line, and \(b\) is the \(y\)-intercept, the value of \(y\) when \(x\) is 0.

A line has a single slope; the rate of change is the same everywhere on the line. But, other functions do not have a constant rate of change. Consider a car sitting at a stoplight. It isn’t moving, so its position isn’t changing. While sitting still, the rate of change is 0. But, when the light turns green, the car starts to move. It is moving, so its rate of change is greater than zero. In fact, the rate of change increases. Eventually, the rate of change starts to decrease, until it reaches 0, at which point the car is again standing still. A graph could be drawn of the position of the car with respect to time, with position along the \(y\) axis, and time along the \(x\) axis. It starts at position 0, starts moving at time 1, and finally comes to a stop at time 3, after traveling 1 unit.
5.3. DERIVATIVES

Plot of the path of the car, with time along the x-axis and position along the y-axis.

Before time 1, and after time 3, the slope of this graph is 0. In between 1 and 3, the curve is non-linear. In fact, the curve has a variety of rates of change at various points in the interval (1,3), just has the car travels at a variety of speeds at various times between time 1 and time 3. We will define the rate of change of a curve at a point as the slope of the line tangent to the curve at that point.

A line tangent to the curve at x = 1.5.

The question is, how do you find out what the tangent line is at an arbitrary point on the curve?

5.3.2 Limits of Functions

One possible line of attack is to estimate the slope of the tangent line by finding the slope of a similar line. A secant line is a line which intersects a curve in two
places. If we look at a secant line which intersects at a point just before and just after the point of interest, we get a line which is likely to be similar to the tangent line. We can easily compute the slope of the secant line, because two points fully determine a line.

As an illustration, we will use the function $f(x) = -x^3 + 3x + 2$.

Suppose we want to find the slope of the tangent line at $x = 1.25$. We could make an estimate by looking at the secant line intersecting the curve at $x = 1.0$ and $x = 1.25$. The slope of that line is $\frac{f(1.25) - f(1.0)}{1.25 - 1.0} = -0.8125$.

We can improve the estimate by moving the defined left point of the secant line closer to the tangent point, 1.25. The next plot shows three secant lines (all anchored at 1.25 on the right side): the first at $x = 1$, the second at $x = 1.1$, and the third at $x = 1.2$. We can define these intervals with respect to the distance between the secant point and the tangent point. Call that distance $h$. 
5.3. DERIVATIVES

Then the first secant has $h = 0.25$, the second secant line has $h = .15$, and the third has $h = .05$.

![Graph showing three secant lines approaching x=1.25](image)

The third line is so tight that it is hard to see at the level of magnification of the plot. Its slope is, however, still an approximation.

The key to finding the slope of the tangent line is to “sneak up on it” by selecting a sequence of $h$ values that approaches zero. As $h$ gets closer to 0, $f(x - h)$ gets closer to $f(x)$. A sequence of values of $h$ implicitly defines a sequence of values $f(x - h)$. It also defines a sequence of numbers that are the slopes of the secant lines associated with the $h$ values: $\frac{f(x) - f(x - h)}{x - (x - h)} = \frac{f(x) - f(x - h)}{h}$. If the sequence of secant line slopes has a limit, then that limit will be the slope of the tangent line itself. If we can find that limit, we can dispense with the modifier “approximation”; we will have found the slope of the tangent line exactly.

The expression “a sequence of $h$ values that approaches zero” may seem somewhat vague. Which sequence should be used? The answer is that it doesn’t really matter, so long as the sequence approaches zero. The sequence produced by letting $h = \frac{1}{n}$ with $n \to \infty$ would do just fine: $\lim_{n \to \infty} \frac{1}{n} = 0$. For the sake of simplicity and generality, it is customary to write $\lim_{h \to 0}$ to indicate the limit of a sequence defined by substituting values of $h$ into an expression, where the values of $h$ themselves converge to zero.

First, look at the secant line slopes for some decreasing values of $h$. The formula for the slope is $\frac{f(1.25) - f(1.25 - h)}{h}$. As the value of $h$ gets smaller and smaller, approaching 0, the values of the secant line slopes appear to be approaching some value also.
Next, look carefully at the limit we want to evaluate: \( \lim_{h \to 0} \frac{f(1.25) - f(1.25 - h)}{h} \).

Intuitively, as \( h \) approaches 0, the numerator is approaching 0, because the two terms being subtracted are getting closer together. At the same time, the denominator is approaching 0, because it is just \( h \). So, the value of the limit will depend, in a sense, on how fast each of the numerator and the denominator approach 0.

Now, we will try the algebraic approach.

\[
\begin{align*}
f(x) &= -x^3 + 3x + 2 \\
f(1.25) &= 3.79687 \\
f(1.25 - h) &= 3.79687 + 1.6875h - 3.75h^2 + h^3
\end{align*}
\]

Plugging these values into the formula for the slope, we get

\[
\frac{3.79687 - (3.79687 + 1.6875h - 3.75h^2 + h^3)}{h} = \frac{-1.6875 + 3.75h^2 - h^3}{h} = -1.6875 + 3.75h - h^2
\]

Therefore, \( \lim_{h \to 0} \frac{f(1.25) - f(1.25 - h)}{h} = \lim_{h \to 0} [-1.6875 + 3.75h - h^2] = -1.6875 \).

The last step is justified because \( h \) is going to zero, so a constant number multiplied by \( h \) will approach 0 as \( h \) approaches 0. If you look back at the table of slopes for the secant lines, the value -1.6875 hopefully appears as a plausible limit to that sequence of numbers. Thus, we can calculate the slope of the line that is precisely tangent to the curve at \( x = 1.25 \) by "sneaking up on it" with a converging sequence, and calculating the limit of that sequence.

### 5.3.3 Rates of Change Provide Useful Information

In the example above, the rate of change was a change with respect to time; specifically, the change in physical position of a car with respect to time. However, a rate of change can more generally be the rate at which a function \( f(x) \) is changing with respect to the value of a variable, \( x \), that it is dependent upon. For instance, if a language learner’s performance \( p(x) \) is dependent upon some variable \( x \), then the rate of change of \( f \) with respect to \( x \) indicates whether the learner’s performance will increase (get better) or decrease (get worse) as the value of \( x \) is increased.

Consider the function in the following plot again:
We can predict the sign (positive, negative) of the rate of change of this function by looking at the plot. In the interval $(-1, 1)$, the value of $f(x)$ is increasing as $x$ increases, so the rate of change is positive. In the intervals $(-2, -1)$ and $(1, 2)$, the value of $f(x)$ is decreasing as $x$ increases, so the rate of change is negative.

The points $x = -1$ and $x = 1$ are special for this function. Look at $f(1)$. It is the very highest point in the area around 1. The tangent line at $x = 1$ is perfectly horizontal, so the slope of the tangent line is 0. The point $x = 1$ is a local maximum for the function $f(x)$. That is because $f(1)$ is the largest value of $f(x)$ for all the points nearby, or local, to 1.

Now look at $f(-1)$. It is the very lowest value of $f(x)$ for all points near -1. The point $x = -1$ is a local minimum for $f(x)$. The tangent line at $x = -1$ is also horizontal, and so its slope is also 0. In general, local maxima and local minima are referred to as local optima. A local maximum which is also at least as high as the value of $f$ for any other value of $x$ is called a global maximum. The mirror image of that definition is a global minimum.

Notice that the rate of change is 0 for both local maxima and local minima. Those are two of the four possible cases where the rate of change can be zero at a point. The third possibility is that the function is just a horizontal line around the point, so that all values of $f(x)$ in the locale are equal. The fourth possibility is called a saddle point, illustrated in the plot below at $x = 0$. 

\[ f(x) = -x^3 + 3x + 2 \]
A function with a Saddle Point at $x = 0$

Because local optima are interesting, the fact that the rate of change is 0 at them is useful.

### 5.3.4 Derivatives Define Functions

The derivative of a function $f(x)$ generally is another function $f'(x)$, where the value of $f'$ at any particular point $x$ is the rate of change of $f$ at $x$. This can be accomplished algebraically by simply not substituting a specific value for the $x$ when computing the limit.

Recall the function $f(x) = -x^3 + 3x + 2$. To take the derivative of this point at point $x$, we apply the definition:

$$
\lim_{h \to 0} \frac{f(x) - f(x - h)}{h} = \lim_{h \to 0} \frac{[-x^3 + 3x + 2] - [- (x - h)^3 + 3(x - h) + 2]}{h} \\
= \lim_{h \to 0} \frac{[-x^3 + 3x + 2] - [- (x^3 - 3hx^2 + 3h^2x - h^3) + (3x - 3h) + 2]}{h} \\
= \lim_{h \to 0} \frac{[-x^3 + 3x + 2] - [-x^3 + 3hx^2 - 3h^2x + 3x + h^3 - 3h + 2]}{h} \\
= \lim_{h \to 0} \frac{0 - [3hx^2 - 3h^2x + h^3 - 3h]}{h} \\
= \lim_{h \to 0} \frac{-3hx^2 + 3h^2x - h^3 + 3h}{h} \\
= \lim_{h \to 0} \frac{-3hx^2 + 3hx - h^2 + 3}{h} \\
= -3x^2 + 3
$$

Again, all terms with an $h$ in them have limit zero as $h \to 0$, so they effectively drop out. The result is the derivative function $f'(x) = -3x^2 + 3$. For all values of $x$, the derivative of $f(x)$ is the value of $f'(x)$.

Recall from the previous section that local optima occur at points where the derivative is zero. If we can express the derivative as a function in its own right,
then we can find local optima by finding the points at which the function is zero. For the above example, the derivative will be zero for values of $x$ satisfying the equation $-3x^2 + 3 = 0$. Solving for $x$ gives

$$-3x^2 + 3 = 0$$
$$x^2 = 1$$
$$x = \{1, -1\}$$

If we can express the derivative as an algebraic expression, then we can determine the zero points of the derivative algebraically.

5.4 Integration

5.4.1 Going the other way: Adding up rates of change

If you know how fast a car has been traveling for some interval of time, you should be able to figure out how far it has traveled. If, in addition, you know where it was at the beginning of the time interval, then you know exactly where it is, by adding the amount it has traveled to its starting position.

If a car was traveling 20 miles per hour for 2 hours (in a single direction), you can easily compute how far it has traveled:

$$20 \text{ mph} \times 2 \text{ hr.} = 40 \text{ mi.}$$

Multiplying the time traveled by the rate is the logical step. We could represent the distance traveled with a rectangle: one side of the rectangle is the rate of change, and the other side is time traveled at that rate. The area of the rectangle is the product of the two sides, i.e., the product of the rate and the time.

If the car travels at different rates at different times, we can easily accommodate by having a different rectangle for each period when the car was traveling at a constant rate: compute the area of each rectangle, and then add together the areas of the rectangles.

Thus, if we are given a simple graph indicating the rate of the car at different times, we should be able to compute how far the car has traveled between any two times, by adding up the areas of the regions under the plot between those two times. An example is shown below.
A car travels at rate 1 from time $t=1$ to $t=3$, and rate 3 from $t=3$ to $t=4$.
Total distance is $1 \times 2 + 3 \times 1 = 5$.

Thus, adding up the consequences of rates of change can be viewed as the area under a function: it is an “area” because it is the product of the two dimensions of the plot of the function: the independent variable (in the car example, time), and the value of the function (in the car example, rate). This “adding up” process is known as integration.

Notice that this computation is something like the reverse of a derivative. A derivative takes a position function and finds another function giving the rate of change. Applying integration to a function over an interval, known as taking the integral of that function, and finds the net change in the value of that function over the interval. Note that the “net change” of the function is equivalent to the difference between the values of the function at the beginning and end of the interval. The integral of a function $f(x)$ over the interval $[a, b]$ will always be equal to $f(b) - f(a)$. If we add up all the changes in a car’s position over an interval of time, the result will be the same as subtracting the starting position of the car from the finishing position of the car.

### 5.4.2 Constantly Changing Rates

Recall the car graph from section 5.3.1.
When the rates of change are always locally constant (unchanging for some period of time), the simple multiply approach will give you an exact answer. But cars don’t instantaneously change from standing still to traveling at high speed. They gradually accelerate from a rate of zero to larger rates. The graph below shows the actual rate function for the car with the position function depicted above.

How are we to compute the distance traveled? We can characterize it geometrically: we want to know the area under the curve. But how do we find that? We’d like to somehow add up the area “under each point”, but that isn’t really well defined; a point doesn’t have a width.

In trying to find a tangent line, we started with a line that was just an approximation, but was easy to find. For the area, we could imagine a similar thing: find some approximating region with an area that is easy to calculate. A natural move would be to put some rectangle under the curve.
Of course, a single rectangle is going to be a poor estimate of the area. Several narrower rectangles will probably give a better approximation. In fact, the more rectangles we use, the better the approximation should be. And that provides our opportunity to use the same trick we used for derivatives: take a sequence of approximations, and see if it converges to a limit.

### 5.4.3 Definite Integrals

The integral of \( f(x) \) over the interval \([a, b]\) can be written as

\[
\int_{a}^{b} f(x) \, dx = \lim_{n \to \infty} \left[ \sum_{x=1}^{n} \left( \frac{b-a}{n} \right) f \left( a + \frac{x(b-a)}{n} \right) \right]
\]

It should be pointed out that this is not the only way to formulate an integral; this is a way that uses right rectangles (each rectangle has as its height the value of the function at the right edge of its interval). There are other ways of formulating integrals that get you the same answer in the end. In this formula, the \( n \) in the summation is the number of rectangles used in the approximation. The width of each rectangle is the width of the entire interval, \((b - a)\), divided by the number of rectangles, \( n \). Thus, \( \left( \frac{b-a}{n} \right) \) is the width of each rectangle. The height of each rectangle is determined by the value of the function \( f(x) \) at the right edge of each rectangle. The value the function is evaluated at, \( a + \frac{x(b-a)}{n} \), is the left edge of the interval, \( a \), plus a multiple of the width of the rectangles. The index \( x \) in the summation inside the limit counts over the rectangles. The height of the third rectangle is the value of the function at the point \( a + 3 \times \left( \frac{b-a}{n} \right) \), or three rectangle widths over from the beginning of the integral.

The limit increases the number of rectangles towards infinity. Notice that as the number of rectangles gets larger, the width of each gets smaller:

\[
\lim_{n \to \infty} \left( \frac{b-a}{n} \right) = 0
\]

Having the width of the rectangles go to zero is analogous to letting \( h \) go to zero for derivatives. In both cases, you cannot evaluate what is happening at a single point directly, so you evaluate smaller and smaller intervals, that shrink so as to approach a single point in size.

For a visual illustration, consider the following rate of change function.
A rate of change function.

To approximate the area under this curve, we could construct a few rectangles, and add together the areas of the rectangles. For example, here we subdivide the interval of interest, $[1,3]$, into five subintervals, and project a rectangle over each subinterval, with the height of each rectangle determined by the value of the rate function at the right edge of the subinterval.

The area approximated by five rectangles (the last rectangle has zero height, because the function has value zero at the right edge of the interval).
Clearly, five rectangles only provide an approximation. The rectangles are higher than the function at some points, and lower at others. But now look at an approximation with 20 rectangles.

While not perfect, 20 rectangles are clearly a closer approximation; the undershoots and overshoots are much smaller than with only 5 rectangles. Now look at the approximation with 200 rectangles.

This is how integration works. As the width of the rectangles approaches zero, the error in the approximation approaches zero.

### 5.4.4 Improper Integrals

The integral of \( f(x) \) over the interval \([a, \infty)\) is an example of an improper integral, because it is not integrating over a finite interval. At first glance, it may not be apparent how to apply the definition of the definite integral to this case; simply substituting \( \infty \) for \( b \) won’t work, since it yields undefined expressions like \((\infty - a)\). The key is to apply, yet again, our approach to dealing with infinities in uncountable sets: sneak up on it with a limit.

\[
\int_a^\infty f(x) \, dx = \lim_{b \to \infty} \int_a^b f(x) \, dx
\]

The result is “a limit of a sequence of limits”. But that is perfectly fine, so long as the sequence of limits converges. Effectively, this approach computes the area under the function for increasingly large intervals, and finds the limit of that sequence of areas as the right edge of the interval approaches positive
5.4. INTEGRATION

Figure 5.1: The area approximated by 200 rectangles.

infinity. If the sequence converges, then you can compute the exact integral of
the function over an infinite interval.

Improper integrals can go to positive infinity, negative infinity, or both.

\[
\int_{-\infty}^{b} f(x) \, dx = \lim_{a \to -\infty} \int_{a}^{b} f(x) \, dx
\]

Taking an integral to infinity in both directions at once might sound extremely
tricky at first, but like many things, it isn’t too hard once you know the right
way to look at it.

\[
\int_{-\infty}^{\infty} f(x) \, dx = \lim_{a \to -\infty} \int_{-a}^{a} f(x) \, dx
\]
Chapter 6

Probability and Statistics

6.1 The Nature of the Beast(s)

The world is an uncertain place. There are many (all?) situations in which we are not certain about what is going to happen next or why. However, that does not imply that we know nothing. In many situations, we know that some events are not possible while others are, and we have much stronger expectations of some events than others. Probability and statistics are both concerned with quantifying that uncertain knowledge, and reasoning with it.

Probability theory is a branch of mathematics, focusing on nonnegative functions that sum to 1. Probability provides a mathematical expression of uncertainty; the probability of an event is a quantification of our expectation about the occurrence of that event. Probabilistic models are used to model systems about whose behavior we are uncertain. If a model contains several elements that exhibit uncertain, or random, behavior, probability theory provides the mathematical framework for characterizing the behavior of those elements, and combining those characterizations into a model of the overall system. Probability theory is about starting with descriptions of systems, and reasoning toward predictions about the behavior of the system. To put it another way, probability theory is about predicting what kinds of data will occur in various situations.

Statistical theory is the study of making decisions based upon data. As such, statistical concepts are central to all of the sciences. To put it another way, statistics is about starting with data, and reasoning toward conclusions about the underlying system that produced the data. Normally, a statistical analysis assumes that the underlying system generating the data is a probabilistic model of some sort, and it uses the data to try to infer properties of that underlying system.

The uncertainty being quantified by probabilities is normally a property of our knowledge of a system, as opposed to a property of the system itself. For example, we may believe that a patient has a low probability of having a disease. Then we administer a medical test, get a positive reading, and decide that the
patient has a much higher probability of having the disease. The medical test
did not cause anything to change inside the patient (they either have the disease
or they don’t, pure and simple). What changed is our knowledge concerning
the patient.

6.2 Basic Probability

6.2.1 Outcomes, Sample Spaces, and Events

A random experiment is the observation of an occurrence. The term “random”
does not necessarily mean unpredictable or uncontrolled; it is intended to allow
for the possibility that there is uncertainty about the occurrence in advance of its
occurring. An elementary outcome is a possible result of a random experiment.
Uncertainty about the outcome of a random experiment exists when there is
more than one elementary outcome for the experiment. The sample space for
a random experiment is the set of all possible elementary outcomes for that
experiment.

Example 6.1 A coin flip is a random experiment. The sample space for a coin
flip contains two elementary outcomes: heads and tails. The sample space can
be represented \{H, T\}.

Example 6.2 Another random experiment is a coin flip and a roll of a single
six-sided die. The sample space has twelve elementary outcomes:
\{(H, 1), (H, 2), (H, 3), (H, 4), (H, 5), (H, 6), (T, 1), (T, 2), (T, 3), (T, 4), (T, 5), (T, 6)\}.

An event is a subset of a sample space.

Example 6.3 Recall the random experiment of example 6.2. The event that
the die comes up 3 is the set \{(H, 3), (T, 3)\}. The event that the die comes up
as an even number is a set with six elementary outcomes,
\{(H, 2), (T, 2), (H, 4), (T, 4), (H, 6), (T, 6)\}.

Events are “where the action is” in probability theory. Events are typically
characterized by properties of outcomes we care about.

6.2.2 Probability Distributions

A probability distribution for a sample space assigns probabilities to events,
subject to several requirements (let \(S\) be the sample space):

- The probability of any event is a nonnegative real number.
- \(P(S) = 1\) (the probability assigned to the entire sample space is always
  1).
- \(P(\{\}) = 0\) (the probability assigned to the empty set is always 0).
• \( P(A \cup B) = P(A) + P(B) - P(A \cap B) \)

If follows from the requirements that no event may have a probability greater than 1.

For simple probability distributions, it is common to definitionally assign probabilities to events containing only one elementary outcome, and obtain the probabilities of larger events by summing the probabilities of the contained single-outcome events. This is not mandatory however. In fact, when we look at continuous probability distributions later in this chapter, non-zero probabilities will only be assigned to events containing many outcomes.

Example 6.4 The probability distribution for a fair coin flip assigns \( P(\{H\}) \) = \( P(\{T\}) \) = 0.5. It follows from the definition of probability distribution that \( P(\{H,T\}) = 1 \) and \( P(\{\}) = 0 \).

For simple distribution definitions like this, it is almost unavoidable to talk of probability distributions assigning probabilities “to outcomes”. This is perfectly natural, and normally does not cause confusion, just remain aware that technically probability distributions assign probabilities to events. When someone (like your instructor, for example) says “the probability of the outcome tails”, you should take that to mean “the probability of the event containing the outcome tails”.

We can construct compound events out of simple ones. We can engage in “logical” reasoning about compound events by translating standard logical operators into their set-theoretic counterparts. Here are some basic rules of event combination (let \( S \) be a sample space, and events \( A, B \subseteq S \)).

\[
\begin{align*}
P(A \text{ and } B) &= P(A \cap B) \\
P(A \text{ or } B) &= P(A \cup B) = P(A) + P(B) - P(A \cap B) \\
P(\text{not } A) &= 1 - P(A)
\end{align*}
\]

Two events are mutually exclusive if they have no sample points in common, that is, \( A \cap B = \emptyset \). In such a special circumstance, \( P(A \text{ and } B) = P(A \cap B) = 0 \), and \( P(A \text{ or } B) = P(A) + P(B) \).

### 6.2.3 Bayes Theorem

Conditional Probability

One way of interpreting probability is in reference to inherent randomness in the world. Another way is in reference to the amount of information we have about a situation. In general, the more information we have about a situation, the more precise and confident we can be in our expectations about the outcome. We can express the relationship between information and probability in terms of conditional probability.

The probability of \( A \) given \( C \) is denoted \( P(A|C) \) and is defined as

\[
P(A|C) = \frac{P(A \cap C)}{P(C)}
\]
A useful form is obtained by simple algebraic rearrangement:

\[ P(A \cap C) = P(A|C) P(C) \]

The commutative symmetry of intersection means that we can get the same result by conditioning on \( A \) instead of \( C \). This can be shown by swapping \( A \) and \( C \):

\[ P(C \cap A) = P(C|A) P(A) = P(A|C) P(C) \]

Two events are independent if the occurrence of one has no effect on the probability of the other. If \( A \) and \( B \) are independent, then \( P(A|B) = P(A) \). When events are independent, we get a special rule for the probability that both events occur:

\[ A, B \text{ independent: } P(A \text{ and } B) = P(A|B) P(B) = P(A) P(B) \]

**Bayes Theorem**

Bayes Theorem is a very powerful and important result that allows us to relate conditional probabilities in both directions: it allows us to relate \( P(A|B) \) to \( P(B|A) \). NOTE: in this context, \( \overline{A} \) is interpreted “not \( A \”; it has nothing to do with mean (average).

\[ P(A|B) = \frac{P(A) P(B|A)}{P(A) P(B|A) + P(\overline{A}) P(B|\overline{A})} \]

You can arrive at this by thinking of it in the following way:

\[ P(A|B) = \frac{P(A \text{ and } B)}{P(B)} = \frac{P(A \text{ and } B)}{P(A \text{ and } B) + P(\text{NOT } A \text{ and } B)} \]

**The Paradox of False Positives**

Suppose there is a rare disease which infects one out of every 1000 people in a population. We have a test for this disease, to tell if someone has it in the early stages. The test is not perfect, but it is good:

- If a person has the disease, the test comes back positive (indicating that they have the disease) 99% of the time.
- If a person does not have the disease, the test comes back negative (indicating that they do not have the disease) 98% of the time.

In other words, if a patient does not have the disease, there is only a 2% chance of a false positive test result.

The test is described above in terms of the probability of the test result given the disease state of the patient. But in using the test we would like to
6.2. BASIC PROBABILITY

switch those around: we want to know the probability of the disease state of the patient given the test result. In other words, if you test positive, what is the probability that you have the disease?

We can express some of the basic facts just given in terms of probabilities. A random experiment in this case consists of a single patient. We can express what we know as follows:

- \( P(\text{dis}) = 0.001 \) (the prob. that a patient has the disease)
- \( P(\text{pos} \mid \text{dis}) = 0.99 \) (the prob. that a person tests positive if they have the disease)
- \( P(\text{neg} \mid \text{nodis}) = 0.98 \) (the prob. that a person tests negative if they do not have the disease)

Relationships among the various probabilities of interest are expressed in the following table:

<table>
<thead>
<tr>
<th></th>
<th>has disease</th>
<th>no disease</th>
<th>SUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>test positive</td>
<td>( P(\text{dis and pos}) )</td>
<td>( P(\text{nodis and pos}) )</td>
<td>( P(\text{pos}) )</td>
</tr>
<tr>
<td>test negative</td>
<td>( P(\text{dis and neg}) )</td>
<td>( P(\text{nodis and neg}) )</td>
<td>( P(\text{neg}) )</td>
</tr>
<tr>
<td>SUM</td>
<td>( P(\text{dis}) )</td>
<td>( P(\text{nodis}) )</td>
<td>1</td>
</tr>
</tbody>
</table>

We can use the descriptions of disease and test above to fill in the table:

- \( P(\text{dis and pos}) = P(\text{pos} \mid \text{dis})P(\text{dis}) = (0.99)(0.001) = 0.00099 \)
- \( P(\text{nodis and pos}) = P(\text{pos} \mid \text{nodis})P(\text{nodis}) = (0.02)(0.999) = 0.01998 \)

Plugging in, we get

<table>
<thead>
<tr>
<th></th>
<th>has disease</th>
<th>no disease</th>
<th>SUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>test positive</td>
<td>0.00099</td>
<td>0.01998</td>
<td>0.02097</td>
</tr>
<tr>
<td>test negative</td>
<td>( P(\text{dis and neg}) )</td>
<td>( P(\text{nodis and neg}) )</td>
<td>( P(\text{neg}) )</td>
</tr>
<tr>
<td>SUM</td>
<td>0.001</td>
<td>0.999</td>
<td>1</td>
</tr>
</tbody>
</table>

We can fill in the rest of the table given the knowledge that the columns and rows must sum up:

<table>
<thead>
<tr>
<th></th>
<th>has disease</th>
<th>no disease</th>
<th>SUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>test positive</td>
<td>0.00099</td>
<td>0.01998</td>
<td>0.02097</td>
</tr>
<tr>
<td>test negative</td>
<td>0.00001</td>
<td>0.97902</td>
<td>0.97903</td>
</tr>
<tr>
<td>SUM</td>
<td>0.001</td>
<td>0.999</td>
<td>1</td>
</tr>
</tbody>
</table>

Now we have an answer to our question: if you test positive, the probability that you have the disease is

\[
P(\text{dis} \mid \text{pos}) = \frac{P(\text{dis and pos})}{P(\text{pos})} = \frac{0.00099}{0.02097} = 0.0472
\]
In other words, less than 5%!

How could this be, if the test is so accurate? The answer lies in the base rate of occurrence of the disease: it is very rare. Note that the test is informative: if you test positive, the probability of your having the disease has increased from 1 in 1000 to 1 in 21. This change in probability doesn’t reflect any change in you (you either have the disease or you don’t); it reflects a change in our knowledge, a change in our uncertainty about the situation.

This may seem strikingly counterintuitive, especially when it is presented in terms of probabilities (as was done above). It can sometimes help to think of it in terms of frequency of occurrence. If we select 1000 people at random, we expect one of them to have the disease, and 999 of them to not have the disease. If we administer the test to all of the people selected, we expect to be administering the test to 999 people without the disease. Because the test has a 2% false positive rate, we expect \((0.02) (999) = 19.98\) false positives, call it 20 false positives in our group of 1000. On the other hand, our group only has one person with the disease; if we simply grant that the test will correctly come back positive for the person with the test, then we expect one true positive in our group of 1000. Thus, we expect a total of 21 people to test positive, and we expect one of those to actually have the disease.

In a sense, the low base rate of occurrence (only 1 in 1000 people has the disease) “stacks the deck” against the test. In a group of 1000 people, the test gets 999 chances to render a false positive, and only one chance to render a true positive.

### 6.2.4 Exercises

**Exercise 6.2.1** Suppose two balanced (heads and tails have equal probability) coins are tossed and the upper faces observed.

- List the sample points for the experiment.
- Let \(A\) denote the event that exactly one head is observed and \(B\) the event that at least one head is observed. List the sample points in \(A\) and \(B\).
- Find \(P(A), P(B), P(A \cap B), P(A \cup B), \) and \(P(\overline{A} \cup B)\).

**Exercise 6.2.2** There are ten applicants to a Linguistics department with three positions to fill. How many different ways can the department fill the three positions? If each applicant has an equal probability of being selected to fill one of the positions, what is the probability that the three positions will be filled with the three applicants who appear at the top of a list ordered alphabetically by the applicants’ names?

**Exercise 6.2.3** Let \(A\) and \(B\) be events such that \(P(A) = .5, P(B) = .3,\) and \(P(A \text{ and } B) = .1\). Find the following:

- \(P(A \mid B)\)
6.3. RANDOM VARIABLES

- \( P(B|A) \)
- \( P(A \cup B) \)
- \( P(A \cap B) \)
- \( P(A|A \cup B) \)
- \( P(A|A \cap B) \)
- \( P(A \cap B|A \cup B) \)

**Exercise 6.2.4** A new phonetic correlate for stress is proposed. It renders an independent judgement for each syllable on the presence/absence of stress. If a syllable is in fact not stressed, there is a .05 probability that the test will conclude (incorrectly) that the syllable is stressed.

- What is the probability that the test, when given three unstressed syllables, will conclude that all three are stressed?
- What is the probability that the test, when given three unstressed syllables, will conclude that at least one of the three is stressed?

**Exercise 6.2.5** If \( A \) and \( B \) are independent events, show that \( A \) and \( \overline{B} \) are also independent. Are \( \overline{A} \) and \( \overline{B} \) independent?

**Exercise 6.2.6** Of the linguistic utterances recorded from informants I and II, 40% come from I and 60% come from II. Informant I has an error rate of 8% (8% of their utterances are in fact ungrammatical), and informant II has an error rate of 10%. If an utterance is selected at random from the recordings, what is the probability that the utterance is grammatical?

### 6.3 Random Variables

A probability distribution can be viewed as something like a generative grammar: it spits out (generates) members of its sample space. However, a distribution makes specific commitments to the probability of generating a particular outcome on any given experiment.

A random variable is the outcome (usually assumed to be numerical) of a random experiment. We can talk about the probability of different outcomes for an experiment in terms of the probability of the associated random variable taking different values. Those probabilities are defined by whatever probability distribution characterizes the experiment: the random variable is a way of referring to the value spit out by the distribution in some arbitrary instance. Thus, a random variable and its associated probability distribution are really two sides of the same coin.
6.3.1 Properties of Numeric Random Variables

The Virtues of Numeric Variables

Much work in probability and statistics assumes that the possible outcomes are numeric values. A number of useful properties, including expected value and variance, can be defined and analyzed for distributions with numeric values. These properties can sometimes be carried over to distributions on sample spaces that are not inherently numeric. For instance, with coin-flipping, the sample space is \{H, T\}. But, one can map this sample space to one of numeric values, for instance mapping \(H\) to 1, and \(T\) to 0, yielding a derived sample space of \{0, 1\}.

When no apparent or meaningful mapping of the sample space exists, then these property definitions don’t apply, but then the intuitions behind them usually don’t make sense either. Consider a sample space of strings, and a probability distribution over them. Compare, using your intuitions, the seemingly incoherent notion of “the average string” with the comparably quite straightforward notion of “the average length of a string”.

Expected Value

The expected value of a random variable, also known as the mean of the random variable, is the sum of the possible values, each weighted by their probability. The two terms carry two notations: \(E[X]\) for “the expected value of random variable \(X\)”, and \(\mu\) for “the mean of the random variable.”

\[
\mu = E[X] = \sum_{x} xp(x)
\]

The expected value of a random variable is sometimes thought of as the “center” of the distribution. It is the point around which the possible outcomes are centered, weighted by their probability.

Variance

A random variable also has a variance, which is denoted \(\sigma^2\) and is defined as

\[
\sigma^2 = \sum_{x} (x - \mu)^2 p(x)
\]

If you compare the formula for the variance of a random variable to the formula for the expected value, you will observe that \(\sigma^2 = E[(X - \mu)^2]\). It is the expected value of the square of the distance between the random variable and its mean.

The variance of a random variable can be thought of as a measure of the “spread” of the distribution, measuring how “far” from the center (the mean) points typically are. The standard deviation of a random variable, \(\sigma\), is the square root of its variance.

\[
\sigma = \sqrt{\sigma^2}
\]
It is really a different form for the same value. Each form has its advantages. The advantages of the variance form are theoretical, as will be shown in the section on combining random variables. The standard deviation has advantages when thinking more descriptively about a distribution. The unit of the standard deviation is the same as that of the original random variable, rather than the square of the unit. If a random variable takes values of quantities of meters, then the standard deviation will be some number of meters, while the variance will be expressed in meters squared. Roughly speaking, one expects the value of a random variable to be less than one standard deviation away from the mean about 67% of the time. The value of the random variable will be within two standard deviations of the mean about 95% of the time.

### 6.3.2 Combining Random Variables

One can construct new random variables by taking arithmetic combinations of other random variables. Fortunately, one can often compute the mean and variance of the new random variables as functions of the means and variances of the component random variables.

For linear functions of a single random variable, the following relations hold:

\[
E[aX + b] = aE[X] + b
\]

\[
\sigma^2[aX + b] = a^2\sigma^2[X]
\]

One can also combine different random variables. For expected values, the expected value of the sum is simply the sum of the expected values:

\[
E[X + Y] = E[X] + E[Y]
\]

In the general case, the variance of the sum of two random variables can be complex. But, if the random variables are independent, a really cool property holds: the variances sum also.

\[
\sigma^2[X + Y] = \sigma^2[X] + \sigma^2[Y]
\]

This provides another motivation for defining variance in the standard way.

### 6.3.3 Binomial Random Variables

It is common to analyze situations into a series of elementary experiments, each of which has two possible outcomes: success and failure. If each experiment is independent of the others (success in one experiment does not influence the probability of success in others), and the probability of success is the same for all such experiments, then the experiments may be referred to as Bernoulli trials.

The outcome of a set of Bernoulli trials is commonly expressed with a binomial random variable. Given a probability \( p \) of success on any particular trial, and a sequence of \( n \) trials, the binomial random variable \( X \) is the number of
CHAPTER 6. PROBABILITY AND STATISTICS

trials that have success as their outcome. Thus, a particular binomial random variable has two parameters: probability of success $p$ and number of trials $n$. A complete outcome for a series of Bernoulli trials is a length $n$ sequence of successes and failures. The sample space for a series of Bernoulli trials is the set of all possible such sequences. The random variable $X$ taking a particular value corresponds to an event: the set of all sequences with that particular number of successes.

**Example 6.5** Let binomial random variable $X$ be the number of heads that occur with two flips of a fair coin (we are arbitrarily defining heads as success, and tails as failure). $p = 0.5$ (my definition of a fair coin) and $n = 2$. The probability distribution for $X$ is $P(X = 0) = 0.25$, $P(X = 1) = 0.5$, $P(X = 2) = 0.25$.

There is a general formula, given $n$ and $p$, for computing the probability that a binomial random variable takes a particular value (note that the probability that $X$ has a value below 0 or above $n$ is automatically 0):

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

This formula should be familiar from the Binomial Theorem. $\binom{n}{k}$ is the number of different ways of getting $k$ successes in $n$ trials. $p^k (1-p)^{n-k}$ is the probability of occurrence of any particular sequence with a total of $k$ successes (and, by implication, $n - k$ failures). It is obtained by multiplying together the probability of each elementary outcome (remember, by definition the trials are independent). We get the total probability of the event $X = k$ by adding up the probabilities of all the possible outcomes in the event: since they all have the same probability, $p^k (1-p)^{n-k}$, and there are $\binom{n}{k}$, we can compute the sum by multiplying.

Notice that a binomial random variable is a useful device for turning lots of different kinds of events into numbers: success and failure, which can be anything from coin flips to testing positive for a disease, are converted to a numeric outcome (number of successes).

**Mean and Variance**

The mean of a binomial random variable is

$$\mu = np$$

The variance of a binomial random variable is

$$\sigma^2 = np(1-p)$$

Note that these formulas make sense, given what we know about combining random variables. Think of the random variable for $n = k$ as the sum of $k$
random variables each with \( n = 1 \) (i.e., a separate random variable for each trial). Each such individual trial has an expected value of

\[
E[X_i] = 0(1-p) + 1(p) = p
\]

The variance for such an individual trial is

\[
\sigma^2[X_i] = (0-p)^2(1-p) + (1-p)^2(p) = p^2(1-p) + p(1-p)^2 = p(1-p)(p+(1-p)) = p(1-p)
\]

The expected value of the sum of \( n \) such random variables should be the sum of the expected values of those random variables, each of which is \( p \).

\[
E[X_1 + X_2 + \ldots + X_n] = E[X_1] + E[X_2] + \ldots + E[X_n] = p + p + \ldots + p = np
\]

Similarly, the variance of the sum of \( n \) such random variables will be \( n \) times the variance of each individual variable, or \( np(1-p) \).

### 6.3.4 Exercises

**Exercise 6.3.1** Let \( X \) denote a binomial random variable with \( n = 10 \) trials and probability of success \( p = .6 \).

- Find \( P(X = 5) \).
- Find \( P(8 < X) \).
- Find \( P(1 \leq X \leq 9) \).

**Exercise 6.3.2** A language generation device generates both active and passive voice sentences. The probability of generating an active voice sentence is \( .75 \), and each sentence is generated independently. Suppose we sample five sentences. We are only concerned with the voice (active or passive) of each sentence; an example sample would be AAPAP (active, active, passive, active, passive).

- How many different possible samples are there?
- How many different samples are possible that have exactly four active sentences?
- How many different samples are possible that have at least three passive sentences?
- What is the probability that a sample contains exactly one passive sentence?
• What is the probability that a sample contains at most two active sentences?

Exercise 6.3.3 A multiple-choice exam has 15 questions. Each question has 5 possible answers, only one of which is correct. Suppose that a student taking the exam answers by guessing randomly on each question.

• What is the probability that the student answers exactly 10 questions correctly?

• What is the probability that the student answers at least 10 questions correctly?

• A student’s score is determined by dividing that student’s number of correct answers by the total number of questions. What is the expected value of score of the student randomly guessing?

6.4 Continuous Random Variables

Suppose we have a sample space with an uncountably infinite number of values, and every outcome is equally likely. For instance, the random variable $X$ might take as possible values any real number in the interval $[0,1]$. What is the probability that $X$ is any particular value; what is $P(X = .025)$? For a finite sample space with equally likely outcomes, we just divided one by the number of possible outcomes (the probability of rolling a particular number on a six-sided die is $\frac{1}{6}$). But with an infinite number of possible outcomes, that approach gives us $\frac{1}{\infty}$. What do we do with that?

The correct answer is that $P(X = .025) = 0$. This assertion creates another mystery: if the total probability of all outcomes has to add up to 1, how can we add up the probabilities of the outcomes, each of which is 0, and get all those zeroes to add up to 1?

Now consider a different question about the same random variable: what is the probability that the value is between 0 and .5, $P(0 \leq X \leq .5)$? Intuitively, since it is one-half the total sample space, and all outcomes are equally likely, that should be a probability of 0.5, and in fact that is the correct answer. We need a notion of probability for continuous variables that will capture this.

The solution is to assign probabilities not to individual points, but to continuous ranges of points. To do this, we will need to call upon our mathematics for uncountable sets, i.e., calculus.

6.4.1 Probability Density Functions

Consider the following function $u(x)$, which expresses something like what we want to say.

$$u(x) = \begin{cases} 
0 & if \quad x < 0 \\
1 & if \quad 0 \leq x \leq 1 \\
0 & if \quad 1 < x 
\end{cases}$$
This function has a constant positive value, 1, between the x-values of 0 and 1, and is zero elsewhere. It doesn’t give us probabilities, but it indicates where we want the possible events to be.

Suppose we wanted to “add up” the portion of this function between 0 and 0.5. Our knowledge of calculus tells us that we need to integrate \( u(x) \) between 0 and 0.5. Because it is a constant function, that will just be the area of the rectangle with height 1 and width \((0.5 - 0) = 0.5\). The area is thus \(0.5 \times 1 = 0.5\). This gives us exactly the result we expected intuitively. It should be further comfort that, under this approach, \( P(0 \leq X \leq 1) = 1 \times 1 = 1\).

The function \( u(x) \) described above is an example of a probability density function, or pdf. It is not itself a probability function: \( u(x) \neq P(x) \). But, it provides the basis for a probability function, one which assigns probabilities to intervals. In general, if \( f(x) \) is a pdf for a random variable \( X \), then

\[
P(a \leq X \leq b) = \int_{a}^{b} f(x) \, dx
\]

Relating this to calculus, we cannot describe the probability of an individual value for \( X \) for the same reason that we can’t describe the distance that a car travels in a single instant of time. For a car to travel a distance, we must be talking about a “measurable” interval of time. For an event defined on a continuous space, the event must include a “measurable” interval of outcomes.

The traditional defining properties of probability distributions are preserved by the following mandatory restrictions on pdfs:

- \( \forall x, 0 \leq f(x) \)
- \( \int_{-\infty}^{\infty} f(x) \, dx = 1 \)

The probability of an event consisting of two disjoint intervals is the sum of the probabilities of the events consisting of each individual interval. If \((a, b)\) and \((c, d)\) are disjoint, then \( P(a \leq X \leq b \text{ OR } c \leq X \leq d) = P(a \leq X \leq b) + P(c \leq X \leq d). \)
6.4.2 Mean and Variance

The Mean

The mean of a continuous random variable with pdf \( f(x) \) is

\[
\mu = \int_{-\infty}^{\infty} x f(x) \, dx
\]

This may make a little more sense if you observe the symmetry with the mean for a discrete random variable:

\[
\mu = \sum_{\text{all } x} xp(x)
\]

The Variance

The variance of a continuous random variable with pdf \( f(x) \) is

\[
\sigma^2 = \int_{-\infty}^{\infty} (x - \mu)^2 f(x) \, dx
\]

Again, comparison with the discrete case is instructive:

\[
\sigma^2 = \sum_{\text{all } x} (x - \mu)^2 p(x)
\]

6.4.3 Interpreting Probabilities

With finite sample spaces, if the probability of an event is zero, it means that the event never happens; it is impossible. With continuous sample spaces, that intuitive property doesn’t apply. \( P(A) = 0 \) no longer automatically means that \( A \) is impossible! Individual values are possible, but any one of them by themselves is not enough to have positive probability: only continuous intervals of values can have positive probability.

6.4.4 Normal Random Variables

The Normal pdf

The normal distribution is the famous “bell curve” that you are always hearing about. It is actually a family of distributions with two parameters: mean \( \mu \) and standard deviation \( \sigma \). The terms “mean” and “standard deviation” mean the same thing as usual here, it is just that instead of independently, completely specifying a distribution and then computing what its mean and standard deviation are, we instead specify a shape (normal), mean and standard deviation, and determine from those the distribution.

The pdf for a normal random variable with mean \( \mu \) and standard deviation \( \sigma \) is

\[
f(x|\mu, \sigma) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{1}{2} \left( \frac{x-\mu}{\sigma} \right)^2}
\]
A celebrated special member of this family is the \textit{standard normal distribution}. This is a normal distribution with $\mu = 0$ and $\sigma = 1$. Here is a plot of the pdf of a standard normal random variable. The pdf itself is $f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$.

The probability density function for a standard normal random variable.

The normal distribution is celebrated because of the general observation that data which are influenced by many small and independent random effects ("noise") are approximately normally distributed. This observation is referred to by some as the "Fuzzy Central Limit Theorem", even though it is not really a theorem at all, but an observation/generalization.

\textbf{Cumulative Probability Functions (CPFs)}

A \textit{cumulative probability function} (cpf) for a random variable $X$ takes a value $a$ and maps it to the probability that the rv $X$ is less than or equal to $a$. The cpf of a continuous random variable is closely related to the pdf, and in fact can be defined in terms of it:

$$P(X \leq a) = F(a) = \int_{-\infty}^{a} f(x) \, dx$$

We will commonly denote the pdf with lowercase $f(x)$ and the cpf with uppercase $F(a)$.

For a standard normal random variable, the cpf is $F(a) = \int_{-\infty}^{a} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} \, dx$, plotted below, along with the corresponding pdf.
CHAPTER 6. PROBABILITY AND STATISTICS

The cumulative probability function $F(a)$ for a standard normal random variable.

The probability density function $f(x)$ for a standard normal random variable.

Notice that the cpf is strictly nondecreasing. This follows from the definition: as you move right along the horizontal axis, more probability (0 or greater) is being added, so the function, can only go up.

Comparing the pdf and cpf, you can now more easily appreciate the pdf as a rate function, in line with our intuitions about derivatives in calculus. If the cpf is the integral of the pdf, then the pdf must be the derivative of the cpf. In fact, the pdf defines a distribution by telling you how fast probability is accumulating for a given value; it tells you how fast the cpf is changing at that value.

Cumulative probability functions are really handy, because they make it easy to compute the probabilities for various intervals.

- $P(a \leq 2) = P(-\infty \leq a \leq 2) = F(2)$
- $P(-3 \leq a \leq 7) = P(a \leq 7) - P(a \leq -3) = F(7) - F(-3)$
- $P(4 \leq a) = 1 - P(a \leq 4) = 1 - F(4)$
The Z Transformation

It will prove extremely convenient to be able to convert any normal random variable to a standard normal rv. We can do this via the \( Z \) transformation.

\[
Z = \frac{X - \mu}{\sigma}
\]

In the formula above, \( Z \) is a random variable, defined relative to a normal random variable \( X \) with mean \( \mu \) and standard deviation \( \sigma \). By taking the value of \( X \), subtracting its mean and dividing by its standard deviation, we normalize it. The result, \( Z \), is a standard normal random variable.

**Example 6.6** Suppose \( X \) is a normal random variable with \( \mu = 5 \) and \( \sigma = 1.3 \). We can construct a corresponding standard normal rv \( Z = \frac{X - 5}{1.3} \). Thus, \( P(4 \leq X \leq 6) = P\left(\frac{4-5}{1.3} \leq Z \leq \frac{6-5}{1.3}\right) = P(-.769 \leq Z \leq .769) \).

If we have some efficient mechanism for computing the probabilities of intervals for standard normal random variables, then we can compute probabilities of intervals for all normal random variables, by using the Z transformation to convert them to standard normals.

### 6.4.5 Exercises

**Exercise 6.4.1** Let \( Z \) denote a standard normal random variable (i.e., with mean \( \mu = 0 \) and standard deviation \( \sigma = 1 \)).

- Find \( P(2 < Z) \).
- Find \( P(-2 < Z < 2) \).
- Find \( P(0 < Z < 2) \).
- Find \( P(0 < Z < 1.73) \).

**Exercise 6.4.2** Let \( X \) denote a normal random variable with mean \( \mu = 2 \) and standard deviation \( \sigma = 0.1 \).

- Find \( P(2.2 < X) \).
- Find \( P(1.8 < X < 2.2) \).
- Find \( P(2 < X < 2.2) \).
- Find \( P(2 < X < 2.173) \).

**Exercise 6.4.3** Let \( L \) denote the duration, in seconds, of a long vowel uttered by a particular speaker. \( L \) has a normal distribution, with mean \( \mu = 0.20 \) seconds, and standard deviation \( \sigma = 0.01 \) seconds.

- Find \( P(L < 0.19) \), the probability that a particular occurrence of a long vowel has duration less than 0.19 seconds.
- Find \( P(0.18 < L < 0.22) \), the probability that a particular occurrence of a long vowel has duration between 0.18 and 0.22 seconds.
6.5 Sample Distributions

6.5.1 Random Samples

All of statistics is based on the (somewhat elusive) notion of random sample. In probability theory, random samples are easy: just define the appropriate random experiments. If you want a random sample of size $n$ from some distribution, just have the distribution generate an outcome $n$ independent times. The sample is random by definition. It is also a purely mathematical abstraction.

In statistics, a sample is a collection of data taken from the world in some way. A sample can be said to be truly random if the procedure that was used to collect it was equally likely to select any of the logically possible samples. Any factor that makes some of the logically possible samples more likely to be selected by the procedure than others is a source of bias. The principles of statistics work by assuming that all possible samples are equally likely, and using combinatorial properties of the sample space to make connections between the actually collected sample and the population from which the sample was collected.

In real life, completely random samples are essentially impossible to come by. But that shouldn’t discourage us from using statistical principles. If the biases that exist in data collection are independent (probabilistically speaking) of the questions we are interested in answering, they will not affect the conclusions we draw based on the assumption that a sample is random. If we are aware of biases in the data collection process, and can identify key properties of those biases, we can compensate for them. Furthermore, minimizing bias and proceeding on the assumption that a sample is random is often the best we can do, if we are so bold as to draw any conclusions at all based on data.

6.5.2 Binomial Sample Distribution

Suppose we have a machine that repeatedly and independently performs a process that results in either success or failure (like flipping a coin, or making a product without defects, or telling a joke). Suppose further that the probability of success on each instance of the process is the same: $p$. If we ran the process $n$ times, and collected the outcomes we would have a random sample of size $n$.

We could characterize a particular random sample in terms of the proportion of successes: the number of successes (which we will label $x$) divided by the total number of trials ($n$). This proportion is a lot like the probability of success $p$: we would expect the proportion of successes to be somewhere close to $p$ most of the time. Because this is a property of the sample, we will denote it as such with the symbol $\hat{p}$, pronounced “p-hat”.

$$\hat{p} = \frac{x}{n}$$

We gain insight into what different random samples might be like when we realize that the number of successes in the sample can be described as a binomial
random variable $X$ with probability of success $p$ and number of trials $n$. We can define another random variable as a function of $X$, $\hat{P} = \frac{X}{n}$ (we might call this “big p-hat”). The random variable $X$ gives us a distribution for the number of successes in different random samples of size $n$, and the random variable $\hat{P}$ gives us a distribution for the proportion of success in different random samples.

Now, we can ask (and answer) all of the “standard” random variable questions:

- $E[\hat{P}] = \frac{E[X]}{n} = \frac{np}{n} = p$
- $\sigma^2[\hat{P}] = \frac{\sigma^2[X]}{n^2} = \frac{np(1-p)}{n^2} = \frac{p(1-p)}{n}$
- $\sigma[\hat{P}] = \sqrt{\sigma^2[\hat{P}]} = \frac{\sqrt{p(1-p)}}{\sqrt{n}}$

Two really cool things become apparent here. First, the distribution of $\hat{P}$ is centered on $p$, just as we would expect. Second, the standard deviation has a denominator that is a function of $n$ ($\sqrt{n}$ to be precise). This means that as the sample size grows, the standard deviation decreases. These two facts are essential to all of modern statistics. In particular, the standard deviation of $\hat{P}$ is dependent on sample size, but not on population size.

We know one other thing, courtesy of the large-sample behavior of binomial random variables: as $n$ gets very large, the distribution of $\hat{P}$ is approximately normal. It can’t be exactly normal, because a normal distribution extends all the way from $-\infty$ to $\infty$, whereas $\hat{P}$ can only range from 0 to 1. But within that range, its distribution can be closely estimated by a normal distribution.

### 6.5.3 The Central Limit Theorem

Consider now an arbitrary probability distribution generating numeric outcomes. We will refer to the distribution of any one outcome with the random variable $X_i$, and let $\mu$ and $\sigma$ refer to the mean and standard deviation of $X_i$. If we consider generating $n$ outcomes (a random sample), and take the mean of that sample, that mean is also a random variable:

$$\bar{X} = \frac{X_1 + X_2 + ... + X_n}{n}$$

Given our knowledge for combining random variables, we can determine the following:

- $E[\bar{X}] = \mu$
- $\sigma[\bar{X}] = \frac{\sigma}{\sqrt{n}}$

But how is $\bar{X}$ distributed? It turns out that we can answer this question regardless of the distribution of $X$. $\bar{X}$ is (approximately) normally distributed!
This result (which is not a fuzzy observation at all, but a provable mathematical theorem), is known as the Central Limit Theorem: as \( n \) gets large, \( X \) approaches a normal distribution with mean \( \mu \) and standard deviation \( \frac{\sigma}{\sqrt{n}} \).

\[
P(a \leq X \leq b) = P\left( \frac{a - \mu}{\sigma/\sqrt{n}} \leq Z \leq \frac{b - \mu}{\sigma/\sqrt{n}} \right)
\]

The above formula has the \( z \) transformation built into it, and expresses the probability in terms of an interval for a standard normal random variable \( Z \).

### 6.6 Point Estimators and Confidence Intervals

Now, we want to reverse directions, just as we did with Bayes Theorem and conditional probabilities. We start with a sample, and then make an estimate of the underlying system.

#### 6.6.1 Descriptive Statistics

A data set consists of data points, sometimes called observations. If a data set contains \( n \) data points, we will commonly denote them as \( x_1, x_2, \ldots, x_n \). The discussion of datasets below assumes that all datapoints are numeric.

The sample mean of a dataset (commonly known as the average) is

\[
\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i
\]

The median of a dataset is determined by sorting the data by size. If there are an odd number of data points (\( n \) is odd), the median is the middle point: \( x_j \), where \( j = \left\lfloor \frac{n}{2} \right\rfloor \). If \( n \) is even, then the median is the average of the two data points closest to the middle: \( \frac{1}{2} (x_j + x_{j+1}) \), where \( j = \frac{n}{2} \).

The sample variance of a dataset is

\[
s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2
\]

The standard deviation is the square root of the sample variance: \( s = \sqrt{s^2} \).

The sample mean, median, and sample variance are all descriptive statistics. They are all intended to summarize some property of a dataset. The sample mean tells you, in a sense, where the “center” of the sample is; what value the data points are clustered around. The sample variance tells you the “spread” of the dataset around its mean, the “average” distance of a point (really, the square of the distance) away from the mean. You may wonder, then, why the definition of variance has \( (n - 1) \) in the denominator, instead of \( n \). The reasons are rather technical, and we won’t elaborate on them here; suffice it to say for now that using \( (n - 1) \) “works better”, both in theory and in practice.
Descriptive statistics in and of themselves tell you something about datasets. But often, we don’t really care that much about a dataset for its own sake. What we care about is the underlying system that gave rise to the dataset. We want to use properties of the dataset to draw inferences about properties of the underlying system. That is the subject of statistical theory, and the most useful statistics are the ones that provide the most useful information about the underlying system. For the descriptive statistics defined above, the names point the way. The sample mean is the statistical counterpart of the expected value, or mean, of a random variable in probability theory. Likewise, the sample variance is the statistical counterpart to the variance of a random variable in probability theory. Statistical theory tells us, for example, how good a measure of the mean of the underlying system we get from the mean of a particular sample.

6.6.2 Estimating $p$

Suppose we have a binomial system that generates observations. We know (or assume) that the system is properly a binomial system, but we don’t know what the value of $p$ is for the system. We can try to estimate the value of $p$ based upon a random sample from the system.

So, we collect a sample of size $n$, and compute the sample success proportion, $\hat{p}$. The value $\hat{p}$ is our estimate of $p$ based upon the sample. In general, different samples will often give us different estimates: in the last section, we described the distribution of the different estimates with the random variable $\hat{P}$, which we will call an estimator. In particular, it is often called a point estimator, because it is estimating a particular numeric value.

For a given estimate, we would like to know: how good an estimate is it? For reasonable sizes of $n$, we know that $\hat{P}$ is approximately normally distributed. This tells us what a particular $\hat{p}$ is likely to be, relative to the true $p$. For instance, for a normal random variable, the value will be within 1 standard deviation of the mean (in distance, less than $\sigma$ away from $\mu$) about 68% of the time (if you integrate the pdf for a normal distribution from $\mu - \sigma$ to $\mu + \sigma$, the result will be about 0.68). If we expand our interval from $+/-$ one standard deviation to $+/-$ 1.96 standard deviations, we get a range within which $\hat{P}$ will be 95% of the time.

Now, we take advantage of a specific fact about distances: they are symmetric. If, 95% of the time, $\hat{P}$ is less than 1.96$\sigma$ away from $p$, then it is also the case that, 95% of the time, $p$ is less than 1.96$\sigma$ away from $\hat{P}$. So, given a particular estimate, we can conclude the following about the actual value of $p$ of the system:

$$P (\hat{p} - 1.96\sigma \leq p \leq \hat{p} + 1.96\sigma) = 0.95$$

Said in English, we are 95% confident that the true $p$ is within 1.96$\sigma$ of $\hat{p}$. The interval $(\hat{p} - 1.96\sigma, \hat{p} + 1.96\sigma)$ is known as a confidence interval.

If we have our sample, and we have computed $\hat{p}$, all we need to compute our confidence interval is the value of $\sigma$, the true underlying standard deviation.
Unfortunately, if we don’t know the underlying \( p \), we probably don’t know \( \sigma \) either. So we need to do a little further estimation, and compute the standard error of the sample, using \( \hat{p} \):

\[
SE(\hat{p}) = \frac{\sqrt{\hat{p}(1-\hat{p})}}{\sqrt{n}}
\]

\( SE(\hat{p}) \) is only an estimate of \( \sigma \), but it is the best we can do working from a sample, and it is at least theoretically well-behaved. Substituting \( SE(\hat{p}) \) for \( \sigma \), we get our de facto confidence interval formula:

\[
P(\hat{p} - 1.96 \times SE(\hat{p}) \leq p \leq \hat{p} + 1.96 \times SE(\hat{p})) = 0.95
\]

### 6.6.3 Estimating Means

The above analysis rested on the observation that the distribution of \( \hat{P} \) is approximately normal. Recall that the Central Limit Theorem guaranteed this same property for the sample means \( \bar{X} \) of any distribution. We could apply the same procedure as done for estimating \( p \) above, provided we have a way of estimating the standard deviation. This is done with the general notion of *sample standard error*, based upon the sample standard deviation \( s \):

\[
SE(\bar{X}) = \frac{s}{\sqrt{n}}
\]

Recall from the first section that \( s = \sqrt{s^2} \), where \( s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \).

Thus, we can estimate the true mean \( \mu \) of any distribution by taking a random sample, computing the sample mean \( \bar{x} \) and the sample standard error, and using our confidence interval formula:

\[
P(\bar{X} - 1.96 \times SE(\bar{X}) \leq \mu \leq \bar{X} + 1.96 \times SE(\bar{X})) = 0.95
\]

For our specific sample mean \( \bar{x} \), we have 95% confidence that the true mean \( \mu \) is \( \bar{x} \pm 1.96 \times SE(\bar{X}) \).

### 6.6.4 Adjusting Confidence Levels

The above formulas use a *confidence level* of 0.95, or 95%; that is where the 1.96 factor comes from. We can easily use other confidence levels if we wish. However, there is a natural inverse relationship between the confidence level and the width of the confidence interval. For a fixed sample, the higher the degree of confidence, the wider the corresponding confidence interval.

The confidence level is often stated in an alternative way, in terms of a significance level. The *significance level* \( \alpha \) is simply the difference between the confidence level and 1. The value of one determines the value of the other. If the confidence level is 95%, then the significance level is \( \alpha = (1 - 0.95) = 0.05 \).
6.7. HYPOTHESIS TESTING

6.6.5 Exercises

Exercise 6.6.1 You have a random sample of numeric data. You want to use this data to estimate the mean of the distribution that generated the data. The sample size is $n = 15$, sample mean is $\bar{x} = 5.2$, and the sample standard deviation is $s = 1.3$.

1. What is the standard error? Give the formula, and then use a calculator to compute it.

2. Compute the confidence interval for the mean $\mu$ of the distribution using a confidence level of 0.95.

Exercise 6.6.2 You have a random sample of outcomes from a series of Bernoulli trials. You want to use this data to estimate the probability of success of the distribution that generated the data. The sample size is $n = 50$, and the observed number of successes is $x = 30$.

1. What is the sample proportion of success $\hat{p}$?

2. What is the standard error of the sample?

3. Compute the confidence interval for the probability of success $p$ of the underlying distribution, using a confidence level of 0.95.

Exercise 6.6.3 You have a random sample of numeric data generated by an unknown distribution. You want to use this data to estimate the mean of the generating distribution. The sample size is $n = 49$, sample mean is $\bar{x} = 22$, and the sample standard deviation is $s = 4.3$.

1. What is the standard error?

2. Compute the confidence interval for the mean $\mu$ of the distribution using a confidence level of 0.99.

6.7 Hypothesis Testing

6.7.1 Null and Alternative Hypotheses

A hypothesis is a claim about the underlying system, often about something like the mean of a distribution. We can test hypotheses by collecting a sample of data, defining an event based upon that data, and then determining how probable that event is given each of the hypotheses.

The **null hypothesis** is usually the “by chance” interpretation of circumstances, or the “what everyone would have expected anyway” interpretation. The **alternative hypothesis** is the alternative that you are willing to consider. Precisely how you formulate the null and alternative hypotheses depends a lot on what you are trying to accomplish, and what you already know about the situation you are studying.
The style of argumentation used in statistical hypothesis testing directly reflects that of empirical science generally. You cannot use a data sample to “prove” a specific hypothesis correct; there will always be a multitude of hypotheses consistent with any data sample. What you can do with a data sample is falsify a hypothesis, or in this case, at least argue that a data sample renders a particular hypothesis extremely implausible. Hypothesis testing works by calculating the probability of some appropriate event (based upon the sample actually taken), assuming the null hypothesis. The appropriate event is usually constructed with reference to an alternative hypothesis; in a sense one is testing the null hypothesis in opposition to the alternative hypothesis. If the probability of this event is low enough (below some pre-decided level of significance), then the null hypothesis can be rejected in favor of the alternative hypothesis. Data which lead to rejection of the null hypothesis implicitly provide support for the alternative hypothesis.

6.7.2 Large Sample Test for the Population Mean

Suppose you have an unending supply of spoken sentences from the language Raset. You have been told that the average length (measured in seconds) of a sentence in Raset is 16 seconds. However, you are mildly suspicious of the source; suspecting Raset speakers to be on the long-winded side, you wonder if the average might actually be more than 16 seconds. You would like to test this statistically.

**Choose the hypotheses**

- Null Hypothesis $H_0$: $\mu = 16$ seconds
- Alternative Hypothesis $H_a$: $16 < \mu$ seconds

We sometimes denote the mean value assumed in the null hypothesis as $\mu_0$. In this example, $\mu_0 = 16$.

**Choose a test statistic**

By computing the standard error, and applying the $Z$-transformation, we get a random variable that is approximately normally distributed. Our test statistic will be called $z_{obs}$, and is defined in terms of the sample mean $\bar{x}$, the sample standard deviation $s$, the sample size $n$, and the mean predicted by our null hypothesis, labeled $\mu_0$.

$$z_{obs} = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}$$

In the case of our Raset example, $\mu_0 = 16$.

**Choose a significance level**

$\alpha = 0.05$
Select a random sample and compute the p-value

Suppose we select a random sample of 49 sentences from our Raset source. We measure the length of each, and get a sample mean of $\bar{x} = 16.1$ and a sample standard deviation of $s = 0.35$. Using this information, we compute the test statistic.

$$z_{obs} = \frac{16.1 - 16}{0.35/\sqrt{49}} = 2$$

Now we want to compute a p-value based upon the test statistic. Because we are suspicious that the sample mean is higher than the null hypothesis mean, we ask the question: if the null hypothesis is true, what is the probability of seeing a sample mean as high or higher than ours? This translates into the question of the probability of a normal random variable taking a value larger than our test statistic. That is our p-value.

$$P(2 < z) = 0.0228$$

There is only a 2.3% probability of seeing a sample mean as high as ours, if the null hypothesis were true. This p-value is less than our significance level ($p$-value < $\alpha$), so we reject the null hypothesis, and conclude that the average length of a Raset sentence is greater than 16 seconds.

6.7.3 Alternative Hypotheses

What you adopt as your alternative hypothesis depends upon what you are trying to test, as well as what you already know about the situation.

Consider tests for the mean of a population. The standard null hypothesis will be some specific value for the population mean, $\mu = \mu_0$. There are three standard alternative hypotheses to choose from:

- $\mu_0 < \mu$
- $\mu < \mu_0$
- $\mu \neq \mu_0$

Which $H_a$ you use depends upon what you already know (or believe) about the situation, and affects the calculation and evaluation of the p-value.

$H_a : \mu_0 < \mu$ uses the p-value $P(z_{obs} < z)$. We are measuring the area under the standard normal pdf in the interval $(z_{obs}, \infty)$.

$H_a : \mu < \mu_0$ uses the p-value $P(z < z_{obs})$. We are measuring the area under the standard normal pdf in the interval $(-\infty, z_{obs})$.

$H_a : \mu \neq \mu_0$ uses the p-value $P(|z_{obs}| < |z|)$. We are measuring the combined area under the standard normal pdf in two intervals: $(-\infty, -|z_{obs}|) + (|z_{obs}|, \infty)$.

Example 6.7 A series of $n = 10$ Bernoulli trials are run. You want to evaluate the claim that $p = 0.3$ for these trials. You suspect that $p$ is actually lower, so
you adopt an alternative hypothesis of $p < 0.3$. You decide to use a significance level of $\alpha = 0.05$. The sample of 10 trials is \{F,F,F,S,F,F,F,F,F\}.

First, you compute the sample proportion: $\hat{p} = \frac{1}{10} = 0.1$. Because this is a series of Bernoulli trials, we know the standard deviation of $\hat{P}$ under the null hypothesis as a function of $p_0$ (i.e., we don’t need to estimate it with the sample standard error, like we do when we are testing a general population mean):

$$\sqrt{\frac{p_0(1-p_0)}{n}}.$$  

Our test statistic is then $z_{\text{obs}} = \frac{\hat{p} - p_0}{\sqrt{\frac{p_0(1-p_0)}{n}}} = \frac{0.1-0.3}{\sqrt{0.3(0.7)/10}} \approx -0.2/0.11 = -1.43$.

Now, you compute the relevant p-value: $P(z < z_{\text{obs}}) = P(z < -1.43) = .0764$. Our p-value is not less than $\alpha$, so we cannot reject the null hypothesis.

In the example above, the null hypothesis could not be rejected at the given confidence level. This does not mean that the null hypothesis has been shown to be correct, or anything like that. It just means that you have failed to conclusively show it to be incorrect; the standard phrase is “failure to find an effect.” This could be for any number of reasons: the null hypothesis really is true, your sample size was too small to provide conclusive evidence, etc.

### 6.7.4 Exercises

**Exercise 6.7.1** You want to test a hypothesis about the probability of success of a process with a binomial distribution. The hypothesis you want to test is that $p = 0.4$. You strongly suspect that if $p$ is not equal to 0.4, then it must instead be greater than 0.4. You collect a random sample of outcomes, with a sample size of $n = 70$ and you observe $x = 35$ successes. You decide to use a significance level of $\alpha = 0.05$.

1. What is your null hypothesis?
2. What is your alternative hypothesis?
3. What is the test statistic $z_{\text{obs}}$?
4. What is the p-value?
5. Do you reject the null hypothesis?

**Exercise 6.7.2** You want to test a hypothesis about the mean of a distribution based upon a sample. The hypothesis is that the true mean is $\mu = 511$. Use, as your alternative hypothesis, that the mean is less than 511. The sample size is $n = 20$, the sample mean is $\bar{x} = 493$, and the sample standard deviation is $s = 55$. Use a significance level of $\alpha = 0.05$.

1. What is the test statistic $z_{\text{obs}}$?
2. What is the p-value?
3. Do you reject the null hypothesis?
Exercise 6.7.3 You suspect that a coin is weighted so that one side is more probable than the other as the result of a flip. However, you do not have a good guess as to which side might be favored. You select, as the null hypothesis, that $p = 0.5$, and your alternative hypothesis is $p \neq 0.5$. You try several flips with the coin, and you get 12 heads and 8 tails.

1. What is the sample proportion $\hat{p}$? (assume that success is defined as heads)
2. What is the test statistic $z_{obs}$? Give the formula, and then use a calculator to compute it.
3. What is the $p$-value?
4. Do you think that the coin is weighted? How confident are you of your answer?

Exercise 6.7.4 A conjecture is made that the mean length of the utterances made by a young child is 3 words. To test this, you randomly select a set of utterances made by the child. The sample size (number of utterances) is $n = 50$, the sample mean is $\overline{x} = 3.3$, and the sample standard deviation is $s = 1.1$. We want to have as the null hypothesis that the mean is 3.0, and the alternative hypothesis that the mean is larger than 3.0. We will assume that the sample mean is normally distributed (because the sample size is sufficiently large).

1. What is the standard error? Give the formula, and then use a calculator to compute it.
2. What is the test statistic $z_{obs}$? Give the formula, and then use a calculator to compute it.
3. What is the $p$-value?
4. Suppose we choose our decision significance level to be $\alpha = .05$. Can we reject the null hypothesis?
5. Suppose we choose our decision significance level to be $\alpha = .01$. Can we reject the null hypothesis?