Metamathematics

Jacob Denson

February 22, 2016

Table Of Contents

Part I

Mathematical Logic

Chapter 1

What's Logic All About?

Mathematics is the use of mathematical proof to determine truth from observation. Metamathematics is the use of mathematical proof to determine the truths of mathematics; mathematical procedures are scrutinized to determine their correctness and limitations. Though studied since the time of Aristotle, 'hard' metamathematics arose in response to paradoxes in early 20th century mathematics, which counteract fundamental notions of definition and inference. Each individual step of the paradoxes is intuitive, yet the conclusion is ludicrous.

Example (Cantor)**.** *For any set X, Cantor proved that there is no injective function from $\mathcal{P}(X)$ to X. When we take X to be the set of all things, then $\mathcal{P}(X) \subset X$, so injecting $\mathcal{P}(X)$ into X is simple.*

Example (Burali-Forti)**.** *Consider the set Ω of all ordinals. Then Ω is a well-ordered set, and therefore order isomorphic to some element of itself, and thus to a proper segment of itself. But no well-ordered set is order isomorphic to a proper subset of itself.*

Example (Russell)**.** *Consider the following set.*

$$X = \{x : x \notin x\}$$

Russell's paradox rests on an innocent question: Is X in itself? If X was in X, then by definition, we would find that X cannot be contained in itself. So we are lead to believe that $X \notin X$. But then, by construction of X, we conclude X is in X after all! More colloquially, consider a town with a single barber, shaving everyone who does not shave themselves. Does the barber shave himself?

Set theory was the first major source of paradoxes, since it was the first mathematical language powerful enough to encapsulate subjects in a single mathematical object. Some considered these arguments the doom of set theory, a "plague on mathematics" to be cleansed from the field. But logical paradoxes were not limited to set theory. In their work on the Continuum hypothesis, Julius König and Jules Richard found paradoxes attacking the notion of mathematical definability, the ability to specify some mathematical object by english phrases.

Example (Richard)**.** *There are countably many english expressions, yet uncountably many real numbers. We conclude that some real numbers cannot be described in english. Consider a particular enumeration containing all describable real numbers. From the enumeration, construct the expression*

"The number constructed from Cantor's diagonal argument on the enumeration considered."

This number is described in english, though not considered in the enumeration.

Example (König)**.** *There are some real numbers which cannot be describable in english. If the real numbers could be well-ordered, we could consider the expression,*

"The least number not definable in english."

And this number is definable, a contradiction.

Analysis of definability was key to the discovery of Gödel's incompleteness theorem, and Tarski's undefinability theorem. Related to definability are self-referential statements.

Example (Löb)**.** *Consider the Preposition B, defined to be true when* $B \implies A$ *is true. If B is true, then* $B \implies A$ *is true, so A is true. But then* $B \implies A$ *is true by construction, so B is true, and we conclude A is true. Since A was arbitrary, so we conclude that every logical statement is true!*

Example (Curry)**.** *Consider*

$$C = \{x : (x \in x) \implies P(x)\}$$

Then $(C \in C)$ *holds if and only if* $(C \in C) \implies P(C)$ *holds. We must have* $C \in C$*, for if* $C \notin C$*, then* $(C \in C) \implies P(C)$ *holds vacantly. But this implies* $(C \in C) \implies P(C)$*, so* $P(C)$ *is true. We conclude* $P(C)$ *is true, irrespective of the content of the statement* $P(C)$*.*

The classical diagnosis of the paradoxes is that knowledge of traditional logic must be sharpened; metamathematics provides the tools for such an endeavour. Which logical procedures summarize the construction of paradoxes? What standards do we use to define a mathematical statement? Our only hope is to apply the precise weapon of mathematical rigor. In the face of adversity, we do what mathematicians do best – define and conquer.

1.1 Formal Systems

A real life situation can never be modelled completely accurately. A physicists's models are ideals, carved from reality in all senses but experimental parameters. No model describes a systems evolution exactly, and it is myopic to suggest a model's perfection. In spite of this, physics still does a bloody good job! In metamathematics, we attempt to form a mathematical model of mathematical principles. Some principles are pinned down for examination, others lost. We hope this model has enough vitality to provide key insights into real-life mathematics. Whether the method is successful it up to interpretation.

A source of confusion in physics is the stylistic treatment of assumptions as absolute facts. A physicist describes "a planet moving according to the equation $\ddot{x} = -m/x^2$" even if he is actually talking about the dynamical system whose evolution is described by the differential equation $\ddot{x} = -m/x^2$, which *models* the motion of a planet. Such expressions are unavoidable, since they are much more viceral, the formal equivalent dry to the bone. Keep this principle in mind as we begin to build models of logic.

The method of the mathematician proceeds as follows. First, she accepts some fundamental statements found by observation, known as axioms. Using accepted logical derivations, additional statements are derived. We shall construct formal models guided by this methodology. We shall focus first on modelling mathematical sentences, and then on modelling how other sentences are derived from each setence. By studying sets of these statements, we attempt to further our knowledge of the mathematical method. These models are known as formal systems.

The basic object of study in metamathematics is deceptively simple. We take a set Λ, known as an alphabet, and consider **strings** over that

alphabet, finite (possibly empty) sequences with elements in Λ. The set of all such strings over Λ will be denoted Λ^*. The empty string is denoted ε. A string $(v_1,\dots,v_n)$ is often denoted $v_1v_2\dots v_n$; often the choice of alphabet will lead to no confusion. We shall identify an element in Λ with the corresponding one letter string in Λ^*. The concatenation of two strings $s = s_1\dots s_n$ and $w = w_1\dots w_m$, denoted sw, is the string $s_1\dots s_nw_1\dots w_m$. A **substring** of a string s is a string u such that $s = wuv$, for some strings w and v. Substrings are consequtive subsequences of characters in s. If we view concatenation as an associative algebraic operation, then Λ^* is the smallest *monoid* containing Λ, relative to the operation of concatenation. A **language** is a subset of strings over an alphabet. We shall also want to perform **substitution**. If $s \in \Lambda^*$, $x = (x_1,\dots,x_n)$ are distinct letters of Λ, and $w = (w_1,\dots,w_n) \in \Lambda^*$, then we shall let $s[w_1/x_1,\dots,w_n/x_n] = s[w/x]$ denote the string in Λ^* obtained from swapping the x_i with w_i.

In mathematical logic, we model certain mathematical statements as a language over an alphabet, and attempt to formalize the mathematical method via operations on these strings. This notion is encapsulated in what is called a **formal system**. The most general definition consists of a language L, a set of axioms, which form a subset of L, and a set of inference rules, pairs (Γ, s) where $\Gamma \subset L$ is the premises of the inference, and s is the conclusion. The **theorems** of a formal system are members of the smallest set such that

1. every axiom is a theorem
2. if (Γ, s) is an inference rule, and all elements of Γ are theorems, then s is a theorem.

If F is a formal system, we shall let $\vdash_F s$ state that s is a theorem of F.

The whole idea of metamathematics may seem circular. This is the reason for formality. By simulating mathematical understanding in a mathematical model, formal mathematics becomes a second order object. Proofs about this system rely on intuition, whereas proofs in the system have static, precise formulations. In the conversion mathematics becomes stale, abstract, and obtuse. Such is the price to pay for rigorous understanding.

Chapter 2

Prepositional Logic

We shall begin with propositional logic, the simplest formal system to analyze truth. To understand propositional logic, we construct a mathematical model, known as a formal language, which represents the language in which mathematics is performed. The formal language is then analyzed by common mathematical notion. The standard formal language for logic is an analysis of strings, sequences of abstract symbols from a given alphabet. Strings represent mathematical statements; manipulating these strings models how a mathematician infers some mathematical statement from another. It is best to see the tool in action to understand its utility, so we proceed swiftly.

2.1 Syntax

Each abstract symbol in an alphabet represents a precise form in colloquial speech. We begin with prepositional logic, which analyzes basic notions of truth and falsity. Some statements are **atomic**, because they cannot be divided into more base statements. "Socrates is a man" is an atomic statement, as is "every woman is human". "Socrates is a man and every woman is a human" is not atomic, for the statement consists of two separate statements, composed by the connective "and". In English, "every woman is a human" can be broken into statements such as "Julie is a human" and "Laura is a human", yet propositional logic still considers this statement as atomic; the model does not have the capability to precisely model understanding of these complex statements, which are the realm of

predicate logic. We shall discuss predicate logic in the next chapter.

Definition. Let Λ be a set disjoint from $\{(,), \wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\}$. The **propositional language with atoms in** Λ, denoted $\mathrm{SL}(\Lambda)$, is the smallest subset of

$$(\Lambda \cup \{(,), \wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\})^*$$

such that

1. $\Lambda \subset \mathrm{SL}(\Lambda)$.
2. If $s, w \in \mathrm{SL}(\Lambda)$, then $(\neg s), (s \wedge w), (s \vee w), (s \Rightarrow w), (s \Leftrightarrow w) \in \mathrm{SL}(\Lambda)$.

An element of $\mathrm{SL}(\Lambda)$ is called a **formula** or **statement**.

Each **connective** represents a certain linguistical form. Later on, the connection of symbols to meaning will become clear. For now, they are abstract symbols without intrinsic meaning.

Connective	Name of Connective	Meaning of statement
$\neg s$	Negation	"s is *not* true"
$s \wedge w$	Conjuction	"s *and* w is true"
$s \vee w$	Disjunction	"s *or* w is true"
$s \Rightarrow w$	Implication	"*If* s is true, *then* w is true"
$s \Leftrightarrow w$	Bicondition	"s is true, *if, and only if,* w is true"

Take care to notice that $\mathrm{SL}(\Lambda)$ is the *smallest* set constructed, in the same way that most 'smallest objects' exist in mathematics, because the intersection of sets satisfying the set of statements defining the set also satisfy the statements. This property leads to the most useful proof method in logic.

Theorem 2.1 (Structural Induction)**.** *Consider a proposition that can be applied to* $SL(\Lambda)$*. Suppose the proposition is true of all elements of* Λ*, and that if the proposition is true of* s *and* w*, then the proposition is also true of* $(\neg s), (s \wedge w), (s \vee w), (s \Rightarrow w)$*, and* $(s \Leftrightarrow w)$*. Then the proposition is true for all of* $SL(\Lambda)$*.*

Proof. Let P be some property to consider. Note the set

$$K = \{s \in (\Lambda \cup \{(,), \wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\})^* : P(s) \text{ is true}\}$$

satisfies axioms (1) and (2) which define $\mathrm{SL}(\Lambda)$, so $\mathrm{SL}(\Lambda) \subset K$. □

Consider the following application.

Theorem 2.2. *Any sentence in $SL(\Lambda)$ contains as many left as right brackets.*

Proof. Any atom in Λ contains no left brackets, and no right brackets, and thus the same number of each. Let s have n pairs of brackets, and let w have m. Then $(\neg s)$ contains $n+1$ pairs of brackets, and $(s \circ w)$, where $\circ \in \{\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\}$, contains $n+m+1$ brackets. By structural induction, we have proved our claim. □

We need a more in depth theorem to correctly parse statements. If statements can be parsed in two different ways, they become ambiguous. For instance, what is the value of $2+7-5-4$? Is it, by collecting factors in pairs, $9-1=8$, or $9-9=0$. This is the reason for parenthesis. One splits mathematical logic into syntax, studying strings, and semantics, interpreting strings. We are studying syntax to begin with, so that we may begin semantics. Equations must be understood before we calculate with them.

Theorem 2.3. *If $wu = s \in SL(\Lambda)$, where $w \neq \varepsilon$, then w has at least as many left brackets as right brackets, and $w = s$ if and only if w has the same number of left and right brackets.*

Proof. If $s \in \Lambda$, then s is only one letter long, so $s = w$, and has no brackets. Now let s and s' satisfy the theorem. We split our proof into two cases.

1. $wu = (\neg s)$: March through all cases. Suppose w has the same number of left brackets than right. w cannot equal (or $(\neg$, nor $(\neg v$, where v is a prefix of s; by induction, v has at least as many left brackets as right brackets, and then w has more left brackets than right brackets. Thus w must equal $(\neg s)$.

2. $wu = (s \circ s')$: Continue the string march. w cannot equal $(,$ nor $(v$ by induction. Similarily, w cannot equal $(s\circ$, nor $(s \circ v$, where v is a substring of s', so w must equal $(s \circ s')$.

Careful analysis of each case also shows that w must have at least as many left brackets as right brackets. □

Corollary 2.4. *Every string in $SL(\Lambda)$ can be written uniquely as an atom Λ, or $(\neg s)$ and $(s \circ w)$, where s and w are elements of $SL(\Lambda)$. The unique connective in the representative is known as the* **principal connective** *of the statement.*

Proof. Such representations trivially exist. Suppose we have two representations. If one of the representations is an element of Λ, the other representation must have a string of length one, and is therefore equal. If we have two representations $(\neg s) = (\neg w)$, then by chopping off symbols, $s = w$. It is impossible to have two distinct representations $(s \circ w) = (\neg u)$, for no element of $\mathrm{SL}(\Lambda)$ begins with $\neg$. Finally, suppose we have two representations $(s \circ w) = (u \circ v)$. Then either u is a substring of s, or s is a substring of u, and one is the prefix of the other. But both have balanced brackets, which implies $s = u$, and by chopping letters away, $w = v$. Thus representations are unique. □

The corollary above allows us to construct recursive definitions. Let $\Lambda = \{A, B, C\}$, and $\Gamma = \{X, Y, Z\}$. We would like to consider $\mathrm{SL}(\Lambda)$ and $\mathrm{SL}(\Gamma)$ the same, by considering a natural bijection. We extend the map

$$X \mapsto A \quad Y \mapsto B \quad Z \mapsto C$$

by the definition

$$f((s \circ w)) = (f(s) \circ f(w)) \quad f((\neg s)) = (\neg f(s))$$

Such a map is well defined and injective, by the corollary.

Before we finish with syntax, it is interesting to discuss a less natural, but syntactically simpler method of forming sentences, called **polish notation**, after its inventor Jan Łukasiewicz. Rather than writing connectives in **infix notation**, like $(u \wedge v)$ and $(u \Rightarrow v)$, we use **prefix notation**, like $\wedge uv$ and $\Rightarrow uv$. We do not need brackets to parse statements anymore, but it is much more easy to read the statement "$((a \Rightarrow (b \vee c) \wedge d)$", than "$\wedge \Rightarrow a \vee bcd$".

2.2 Semantics

We can understand the discussion in the last section without any understanding of what symbols mean. Now we want to interpret the symbols, giving the symbols meaning. A basic semantic method is to define

whether a statement is 'true'. A **truth assignment** on a set Λ is a map $f : \Lambda \to \{\top, \bot\}$.

Example. *An n-ary* **boolean function** *is a truth assignment, where* $\Lambda = \{\top, \bot\}^n$*. It is common to define such functions by truth tables. For n-ary boolean functions, we form a table with* $n+1$ *columns, and* 2^n *rows. In each row, we fill in a particular truth assignment in* $\{\top, \bot\}^n$*, and in the last column, the value of image of the truth assignment under* f*. One may combine multiple n-ary truth functions into the same table for brevity. There are* 2^{2^n} *n-ary truth functions.*

x	y	$H_\wedge(x,y)$	$H_\vee(x,y)$	$H_\Rightarrow(x,y)$	$H_\Leftrightarrow(x,y)$	$H_\neg(x)$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$	$\top$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$

This table defines the boolean functions $H_\circ$*.*

Homomorphisms between two rings R and S, extend to homomorphisms between the polynomial rings $R[X]$ and $S[X]$. Similarily, we may extend truth assignments f on Λ to assignments f_* on $\mathrm{SL}(\Lambda)$. This is done recursively. Let f be an arbitrary truth assignment. Define

$$f_*(\neg s) = H_\neg(f_*(s)) \qquad f_*(s \circ w) = H_\circ(f_*(s), f_*(w))$$

then f_* is defined on all of $\mathrm{SL}(\Lambda)$.

Let $s \in \mathrm{SL}(\Lambda)$ be a statement. s is a **tautology**, if, for any truth assignment f on Λ, $f_*(s) = \top$, a **contadiction** if $f_*(s) = \bot$, a **cotingent statement** if s is neither a contradiction nor a tautology, and **satisfiable** if it is not a contradiction. We summarize the statement "s is a tautology" by $\models s$. We say a statement s **semantically implies** w, written $s \models w$, if $\models s \Rightarrow w$, or correspondingly, if $f_*(w) = \top$ whenever $f_*(s) = \top$.

Suppose that we wish to verify whether $s \in \mathrm{SL}(\Lambda)$ is a tautology. We say a variable $x \in \Lambda$ **occurs** in s if it is an element of the string. There can only be a finite number of variables $x_1, \ldots, x_n$ which occur in s. Define a boolean function $g : \{0,1\}^n \to \{0,1\}$,

$$g(y_1 \ldots, y_n) = f_*^{(y_1,\ldots,y_n)}(s)$$

where $f^{(y_1,\dots,y_n)}$ is the truth assignment formed by mapping x_i to y_i. This is well defined, because if h and k are two truth assignments which agree on the x_i, then they agree at s. Then one verifies that s is a tautology if and only if $g(y_1,\dots,y_n) = \top$ for all choices of the y_i. Thus one can determine whether a statement is a tautology by constructing a truth table. To prevent errors, it is best to construct a truth table for the subformulas in the principal connective of a formula before constructing the truth table of the formula. This may be done side by side, in the same table.

Example. *For instance, for any variable $A \in \Lambda$, $A \vee \neg A$ is a tautology, $A \wedge \neg A$ is a contradiction, and $\neg A$ is contingent, which is verified by the truth table*

A	$A \vee \neg A$	$A \wedge \neg A$	$\neg A$
$\top$	$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$	$\top$

The first formula is the **law of excluded middle**.

Example. *Let $A \vDash B$ be a tautology, and suppose that A and B have no variables in common. Then A is a contradiction, or B is a tautology, for if there is a truth assignment on the variables of A which make the statement true, and a truth assignment on B which cause the statement to be false, then we may combine the truth assignments to create a truth assignment in which $A \Rightarrow B$ is false.*

Theorem 2.5 (Semantic Modus Ponens)**.** *If $\vDash s$ and $s \vDash w$, then $\vDash w$.*

Proof. Let f be a truth assignment. Then $f_*(s) = \top$ and

$$f_*(s \Rightarrow w) = H_{\Rightarrow}(f_*(s), f_*(w)) = H_{\Rightarrow}(\top, f_*(w)) = \top$$

This holds only when $f_*(w) = \top$. □

Theorem 2.6. *If $\vDash w$, then $\vDash w[s/x]$*

Proof. Consider a truth assignment f. We shall define another truth assignment $\tilde{f}$ such that $f(v[s/x]) = \tilde{f}(v)$ for all v. Define $\tilde{f}(x_i) = f(s_i)$, and if $y \notin x$, define $\tilde{f}(y) = f(y)$. Our base case, where v is a variable, satisfies the claim by construction. Then, by induction, if $v = (u \circ w)$, then

$$\tilde{f}_*(v) = H_\circ(\tilde{f}_*(u), \tilde{f}_*(w)) = H_\circ(f_*(u[s/x]), f_*(w[s/x])) = f_*(v[s/x])$$

A similar proof answers the case where $u = (\neg v)$. Since w is a tautology,

$$f_*(w[s/x]) = \tilde{f}_*(w) = \top$$

So $w[s/x]$ is a tautology. □

Corollary 2.7. *If $v \vDash w$, then $v[s/x] \vDash w[s/x]$.*

Theorem 2.8 (Craig Interpolation)**.** *Let $A \vDash B$, and let the set of variables shared by A and B be $x_1, \ldots, x_n$. Then there is a statement C, known as the* **interpolant***, containing only the variables x_i, such that $A \vDash C$ and $C \vDash B$.*

Proof. We proceed by induction on the number of variables in A which do not occur in B. If every variable in A occurs in B, let $C = A$. In the general case, fix some variable x in A but not in B, take y in both A and B, and define

$$D = A[(y \wedge \neg y)/x] \vee A[(y \vee \neg y)/x]$$

If $f_*(A) = \top$, and $f_*(x) = \bot$, then $f_*(A[(y \wedge \neg y)/x]) = \top$. If $f_*(x) = \top$, then $f_*(A[(y \vee \neg y)/x]) = \top$, so $A \vDash D$. In addition, $D \vDash B$. Let $f_*(D) = \top$. Then $f_*(A[(y \wedge \neg y)/x]) = \top$, or $f_*(A[(y \vee \neg y)/x]) = \top$. In the first case, we modify the truth assignment f so that $f_*(x) = \bot$ (without changing the values of B or D). Then $f_*(A) = \top$, so $f_*(B) = \top$. The other case follows by letting $f_*(x) = \top$. By induction, there is an iterpolant C for which $D \vDash C$ and $C \vDash B$, and then $A \vDash C$, since $(x \Rightarrow y) \wedge (y \Rightarrow z) \vDash x \Rightarrow z$. □

Example. *Consider $A = (y_1 \Rightarrow x) \wedge (x \Rightarrow y_2)$ and $B = (y_1 \wedge z) \Rightarrow (y_2 \wedge z)$. Then $A \vDash B$. We shall find an interpolant.*

$$\begin{aligned} C = &[(y_1 \Rightarrow (y_1 \vee \neg y_1)) \wedge ((y_1 \vee \neg y_1) \Rightarrow y_2)] \\ &\vee [(y_1 \Rightarrow (y_1 \wedge \neg y_1)) \wedge ((y_1 \wedge \neg y_1) \Rightarrow y_2)] \end{aligned}$$

This equation can be simplified to $\neg y_1 \vee y_2$.

2.3 Truth Functional Completeness

We hope that propositional logic can model all notions of truth, such that all truth functions can be formed from our original set. Here we argue why our logic can model all such notions. Let Λ be a set of boolean functions.

The **clone** of Λ is the smallest set containing Λ and all projections $\pi_k^n : \{0,1\}^n \to \{0,1\}$, defined

$$\pi_k^n(x_1,\ldots,x_n) = x_k$$

and in addition, if $g : \{0,1\}^n \to \{0,1\}$ and $f_1,\ldots,f_n : \{0,1\}^m \to \{0,1\}$ are in the clone, then so is $g(f_1,\ldots,f_n) : \{0,1\}^m \to \{0,1\}$. Λ is **truth functionally complete** if its clone is the set of all boolean functions.

Example. *$\{H_\neg, H_\wedge, H_\vee\}$ is a truth functionally complete, since every formula can be put in* **conjunctive normal form**, *for we may write the 'true and false' functions*

$$\top(x) = \top = H_\vee(H_\neg(x), x) \qquad \bot(x) = \bot = H_\wedge(H_\neg(x), x)$$

and then we write

$$f(x_1,\ldots,x_n) = \bigvee_{\substack{(y_1,\ldots,y_n)\in\{0,1\}^n \\ f(y_1,\ldots,y_n)=\top}} \bigwedge_{i=1}^{n} H_\Leftrightarrow(x_i, y_i)$$

where we identify an element of $\{\top,\bot\}$ with the constant boolean function, define $\bigcirc_{i=1}^n f_i = H_\circ(f_n, \bigcirc_{i=1}^{n-1})$, and take

$$H_\Leftrightarrow(x,y) = H_\wedge(H_\Rightarrow(x,y), H_\Rightarrow(y,x)) = H_\wedge(H_\vee(H_\neg(x),y), H_\vee(H_\neg(y),x))$$

This can be further reduced, by noticing that

$$H_\wedge(x,y) = H_\neg(H_\vee(H_\neg(x), H_\neg(y)))$$

which is a truth functional form of Boole's inequality, implying $\{\neg,\vee\}$ is truth functionally complete. We can also consider **disjunctive normal form**, *left to the reader to define.*

Example. *The mathematician Henry M. Sheffer found a single truth function which is truth functionally complete. Consider the* **Sheffer stroke** *$x|y$, also known as* **NAND**, *defined by the truth table*

x	y	$H_\mid(x,y)$
$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\top$
$\top$	$\bot$	$\top$
$\top$	$\top$	$\bot$

Then $H_\neg(x) = H_|(x,x)$, *and* $H_\vee(x,y) = H_|(H_\neg(x), H_\neg(y))$, *which implies, since this set is truth functionally complete, that the sheffer stroke is truth functionally complete.*

The previous example is incredibly important to circuit design. Logical statements can be represented by boolean functions. Since all truth functions can be built from the sheffer stroke, we need only make an atomic circuit for the sheffer stroke, and then all other circuits are constructed by combining sheffer strokes. This is why computers consist of millions of NAND gates.

2.4 Deduction

When mathematicians want to derive whether a statement is true, they do not construct truth functions. Instead, they argue *why* the statement is true. Here we shall provide the mechanics for modelling a mathematical argument. We will show that truth tables and arguments are equivalent – a statement is a tautology if and only if it can be proved.

The set of **theorems** of $\mathrm{SL}(\Lambda)$ are elements of the smallest set such that for any $s, w, u \in \mathrm{SL}(\Lambda)$,

1. Any axiom is a theorem, where the axioms are any statement of the form

$$(A1) \qquad s \Rightarrow (w \Rightarrow s)$$
$$(A2) \qquad (s \Rightarrow (w \Rightarrow u)) \Rightarrow ((s \Rightarrow w) \Rightarrow (s \Rightarrow u))$$
$$(A3) \qquad (\neg s \Rightarrow \neg w) \Rightarrow ((\neg s \Rightarrow w) \Rightarrow s)$$

   where s, w are arbitrary statements.

2. Modus Ponens holds in our system. If $s \Rightarrow w$ and s are theorems, then w is a theorem. When we apply modus ponens, we may write that the theorem was obtained by (MP).

We shall write $\vdash s$ to state that s is a theorem.

For statements, the 'smallness' characterization gives us structural induction. For theorems, smallness gives us the notion of a 'proof'. A statement s is a theorem of $\mathrm{SL}(\Lambda)$ if and only if there is a sequence of formulae $(s_1, \ldots, s_n)$ such that $s_n = s$, and each s_i is either an axiom, or is obtained

from some s_j and s_k by modus ponens, where $j,k < i$. This sequence is known as a proof. Often, we list a proof from top to bottom, where we reference how we obtained each element of the sequence alongside the proof.

Example. *Let us construct a proof of* $\vdash s \Rightarrow s$*, for any* $s \in SL(\Lambda)$.

$\mathsf{s} \Rightarrow \mathsf{s}$	
1. $(\mathsf{s} \Rightarrow ((\mathsf{s} \Rightarrow \mathsf{s}) \Rightarrow \mathsf{s}))$	*(A1)*
2. $(\mathsf{s} \Rightarrow ((\mathsf{s} \Rightarrow \mathsf{s}) \Rightarrow \mathsf{s})) \Rightarrow ((\mathsf{s} \Rightarrow (\mathsf{s} \Rightarrow \mathsf{s})) \Rightarrow (\mathsf{s} \Rightarrow \mathsf{s}))$	*(A2)*
3. $((\mathsf{s} \Rightarrow (\mathsf{s} \Rightarrow \mathsf{s})) \Rightarrow (\mathsf{s} \Rightarrow \mathsf{s}))$	*(1),(2),(MP)*
4. $(\mathsf{s} \Rightarrow (\mathsf{s} \Rightarrow \mathsf{s}))$	*(A1)*
5. $\mathsf{s} \Rightarrow \mathsf{s}$	*(3),(4),(MP)*

In future proofs, we shall be able to use $\vdash s \Rightarrow s$ *implicitly, since we now know the statement can be proved in any of its forms. We will denote its application by* (I).

In mathematics, we work in systems where implicit assumptions are made. In group theory, we assume that operations are associative. In geometry, we assume there is a line between any two points. To perform mathematics, we add additional axioms to logic, and prove results from these axioms. If Γ is a subset of $\mathrm{SL}(\Lambda)$, then we may consider each member of Γ to be an axiom. We write $\Gamma \vdash s$ if one can prove s assuming all elements of Γ are axioms. That is, we may write a sequence $(s_1,\ldots,s_n)$, where $s_n = s$, and each s_i is either an axiom, an element of Γ, or is obtained by modus ponens from previous elements of the sequence.

Theorem 2.9 (Deduction Theorem)**.** *If* $\Gamma \cup \{s\} \vdash w$*, then* $\Gamma \vdash s \Rightarrow w$.

Proof. We prove the theorem by induction of the size of the proof of w. Consider a particular proof $(s_1,\ldots,s_n)$ of w from $\Gamma \cup \{s\}$. Suppose that $n = 1$. Then $s_1 = w$, and w must either be an axiom, an element of Γ, or equal to s. In the first and second case, the proof is equally valid in Γ, and so $\Gamma \vdash s \Rightarrow w$ follows from the axiom $(w \Rightarrow (s \Rightarrow w))$. If $s = w$, Then we have shown that $\Rightarrow w \Rightarrow w$, so obviously $\Gamma \vdash s \Rightarrow w$.

Now we consider the problem proved for $m < n$. $s_n = w$ is either an axiom, an element of Γ, equal to s, or proved by modus ponens from $s_i =$

$(u \Rightarrow w)$ and $s_j = u$. We have already shown all but the last case. By induction, $\Gamma \vdash s \Rightarrow (u \Rightarrow w)$ and $\Gamma \vdash s \Rightarrow u$. But

$$(s \Rightarrow (u \Rightarrow w)) \Rightarrow ((s \Rightarrow u) \Rightarrow (s \Rightarrow w))$$

is an axiom, so $\Gamma \vdash s \Rightarrow w$. □

Example. *For any statements s and w, $\{s \Rightarrow w, w \Rightarrow u\} \vdash s \Rightarrow u$. This follows from a basic application of (A2). But this implies the two cut rules, that*

$$\vdash (s \Rightarrow w) \Rightarrow ((w \Rightarrow u) \Rightarrow (s \Rightarrow u))$$
$$\vdash (w \Rightarrow u) \Rightarrow ((s \Rightarrow w) \Rightarrow (s \Rightarrow u))$$

which is much more tricky to prove.

Example. *Let us prove the double negation elimination axiom, $\vdash \neg\neg s \Rightarrow s$ by the deduction theorem.*

$\neg\neg s \Rightarrow s$	
1. $\neg\neg s$	
2. $((\neg s \Rightarrow \neg\neg s) \Rightarrow ((\neg s \Rightarrow \neg s) \Rightarrow s)$	*(A3)*
3. $(\neg\neg s) \Rightarrow (\neg s \Rightarrow \neg\neg s)$	*(A1)*
4. $(\neg s \Rightarrow \neg\neg s)$	*(1),(3),(MP)*
5. $(\neg s \Rightarrow \neg s) \Rightarrow s$	*(2),(4),(MP)*
6. $\neg s \Rightarrow \neg s$	*(I)*
7. s	*(5),(6),(MP)*
8. $\neg\neg s \Rightarrow s$	*(1-7),(DT)*

In future proofs, application of the statement will be denoted $(\neg\neg E)$. Now lets prove negation introduction, $\vdash s \Rightarrow \neg\neg s$.

$s \Rightarrow \neg\neg s$	
1. $\neg\neg\neg s \Rightarrow \neg s$	$(\neg\neg E)$
2. s	
3. $(\neg\neg\neg s \Rightarrow \neg s) \Rightarrow ((\neg\neg\neg s \Rightarrow s) \Rightarrow \neg\neg s)$	*(A3)*
4. $(\neg\neg\neg s \Rightarrow s) \Rightarrow \neg\neg s$	*(1), (3), (MP)*
5. $s \Rightarrow (\neg\neg\neg s \Rightarrow s)$	*(A1)*
6. $\neg\neg\neg s \Rightarrow s$	*(2),(5),(MP)*
7. $\neg\neg s$	*(4),(6),(MP)*
8. $s \Rightarrow \neg\neg s$	*(2-7), (DT)*

We shall denote this rule $(\neg\neg I)$.

Example. *Lets prove* $\neg w \vdash w \Rightarrow u$*, by proving* $\neg w, w \vdash u$*.*

	$\neg \mathsf{w} \Rightarrow (\mathsf{w} \Rightarrow \mathsf{u})$	
	1. $\neg \mathsf{w}$	
	2. w	
	3. $(\neg \mathsf{u} \Rightarrow \neg \mathsf{w}) \Rightarrow ((\neg \mathsf{u} \Rightarrow \mathsf{w}) \Rightarrow \mathsf{u})$	*(A3)*
	4. $\neg \mathsf{w} \Rightarrow (\neg \mathsf{u} \Rightarrow \neg \mathsf{w})$	*(A1)*
	5. $\neg \mathsf{u} \Rightarrow \neg \mathsf{w}$	*(1),(4), (MP)*
	6. $(\neg \mathsf{u} \Rightarrow \mathsf{w}) \Rightarrow \mathsf{u}$	*(3),(5),(MP)*
	7. $\mathsf{w} \Rightarrow (\neg \mathsf{u} \Rightarrow \mathsf{w})$	*(A1)*
	8. $\neg \mathsf{u} \Rightarrow \mathsf{w}$	*(2),(7),(MP)*
	9. u	*(6),(8),(MP)*
	10. $\mathsf{w} \Rightarrow \mathsf{u}$	*(2-9),(DT)*
	11. $\neg \mathsf{w} \Rightarrow (\mathsf{w} \Rightarrow \mathsf{u})$	*(1-10), (DT)*

This is a proof of **the law of contradiction**, *denoed* (LC).

Example. *Consider the following proof.*

	$(\mathsf{s} \Rightarrow \mathsf{w}) \Rightarrow (\neg \mathsf{w} \Rightarrow \neg \mathsf{s})$	
	1. $\mathsf{s} \Rightarrow \mathsf{w}$	
	2. $\neg \mathsf{w}$	
	3. $\neg \mathsf{w} \Rightarrow (\neg \neg \mathsf{s} \Rightarrow \neg \mathsf{w})$	*(A1)*
	4. $\neg \neg \mathsf{s} \Rightarrow \neg \mathsf{w}$	*(2),(3),(MP)*
	5. $(\neg \neg \mathsf{s} \Rightarrow \neg \mathsf{w}) \Rightarrow ((\neg \neg \mathsf{s} \Rightarrow \mathsf{w}) \Rightarrow \neg \mathsf{s})$	*(A3)*
	6. $(\neg \neg \mathsf{s} \Rightarrow \mathsf{w}) \Rightarrow \neg \mathsf{s}$	*(4),(5),(MP)*
	7. $\neg \neg \mathsf{s}$	
	8. $\neg \neg \mathsf{s} \Rightarrow \mathsf{s}$	$(\neg \neg E)$
	9. s	*(7),(8),(MP)*
	10. w	*(1),(9),(MP)*
	11. $\neg \neg \mathsf{s} \Rightarrow \mathsf{w}$	*(7-10), (DT)*
	12. $\neg \mathsf{s}$	*(6),(11), (MP)*
	13. $\neg \mathsf{w} \Rightarrow \neg \mathsf{s}$	*(2),(12),(MP)*
	11. $(\mathsf{s} \Rightarrow \mathsf{w}) \Rightarrow (\neg \mathsf{w} \Rightarrow \neg \mathsf{s})$	*(1-10), (DT)*

This is the **law of contraposition** (LCP).

Example. *Lets prove* $\vdash (s \Rightarrow w) \Rightarrow ((\neg s \Rightarrow w) \Rightarrow w)$.

$(\mathsf{s} \Rightarrow \mathsf{w}) \Rightarrow ((\neg \mathsf{s} \Rightarrow \mathsf{w}) \Rightarrow \mathsf{w})$	
1. $\mathsf{s} \Rightarrow \mathsf{w}$	
2. $(\mathsf{s} \Rightarrow \mathsf{w}) \Rightarrow (\neg \mathsf{w} \Rightarrow \neg \mathsf{s})$	*(LCP)*
3. $\neg \mathsf{w} \Rightarrow \neg \mathsf{s}$	*(1),(2),(MP)*
4. $\neg \mathsf{s} \Rightarrow \mathsf{w}$	
5. $(\neg \mathsf{w} \Rightarrow \neg \neg \mathsf{s})$	*(4),(LCP)*
6. $\neg \neg \mathsf{s} \Rightarrow \mathsf{s}$	$(\neg \neg E)$
7. $(\neg \mathsf{w} \Rightarrow \neg \neg \mathsf{s}) \Rightarrow ((\neg \neg \mathsf{s} \Rightarrow \mathsf{s}) \Rightarrow (\neg \mathsf{w} \Rightarrow \mathsf{s}))$	*(CUT)*
8. $(\neg \neg \mathsf{s} \Rightarrow \mathsf{s}) \Rightarrow (\neg \mathsf{w} \Rightarrow \mathsf{s})$	*(5),(7),(MP)*
9. $\neg \mathsf{w} \Rightarrow \mathsf{s}$	*(6),(8),(MP)*
10. $(\neg \mathsf{w} \Rightarrow \neg \mathsf{s}) \Rightarrow ((\neg \mathsf{w} \Rightarrow \mathsf{s}) \Rightarrow \mathsf{w})$	*(A3)*
11. $(\neg \mathsf{w} \Rightarrow \mathsf{s}) \Rightarrow \mathsf{w}$	$(3), (10), (MP)$
12. w	*(9), (11), (MP)*
13. $(\neg \mathsf{s} \Rightarrow \mathsf{w}) \Rightarrow \mathsf{w}$	$(2-9), (DT)$
14. $(\mathsf{s} \Rightarrow \mathsf{w}) \Rightarrow ((\neg \mathsf{s} \Rightarrow \mathsf{w}) \Rightarrow \mathsf{w})$	$(1-10), (DT)$

This essentially proves that $(s \vee \neg s) \Rightarrow w$ *implies* w.

We shall verify that proofs never lead to contradiction.

Theorem 2.10. *If* $\vdash s$, *then* $\models s$.

Proof. This is a trivial proof by structural induction. Prove that all axioms are tautologies, and that the set of tautologies is closed under modus ponens. □

This theorem shows that there are some statements which are not provable in our system. In fact, if s is provable, then $\neg s$ is not provable. We call an axiom system like this **absolutely consistant**. We shall show that all tautologies are provable, which shows the system is **complete**. A complete system is effectively one in which all theorems which were meant to be able to be proved, are able to be proved.

Lemma 2.11. *Let* $x_1, \ldots, x_n \in \Lambda$ *be variables in* $s \in SL(\Lambda)$. *Let* f *be a truth assignment, and define* $s' = s$ *if* $f_*(s) = \top$, *or* $s' = \neg s$ *if* $f_*(s) = \bot$. *Then*

$$x_1', \ldots, x_n' \vdash s'$$

Proof. We prove by structural induction. If $s = x_1$, then $x_1' = s'$, and $s \vdash s$ is a trivial theorem. If $s = \neg w$, we consider two cases. If $w' = w$, then $s' = \neg\neg w$, and we have already shown $w \vdash \neg\neg w$, hence $x_1', \ldots, x_n' \vdash s'$. If $w' = \neg w$, then $s' = w'$, and the theorem is trivial. If $s = w \Rightarrow u$, then either $f_*(w) = \top$ and $f_*(u) = \top$, or $f_*(w) = \bot$. In the first case, we have $x_1', \ldots, x_n' \vdash u$, from which $x_1', \ldots, x_n' \vdash w \Rightarrow u$ follows. In the second case, $x_1', \ldots, x_n' \vdash \neg w$, and we have $\neg w \vdash (w \Rightarrow u)$. □

Corollary 2.12 (Completeness Theorem)**.** *If* $\vDash s$, $\vdash s$.

Proof. Let $x_1, \ldots, x_n$ be the variables in s. By the last lemma, we have

$$x_1, \ldots, x_n \vdash s \qquad x_1, \ldots, \neg\neg x_n \vdash s$$

By the deduction theorem

$$x_1, \ldots, x_{n-1} \vdash x_n \Rightarrow s \qquad x_1, \ldots, x_{n-1} \vdash \neg x_n \Rightarrow s$$

But then, since $\vdash (x_n \Rightarrow s) \Rightarrow ((\neg x_n \Rightarrow s) \Rightarrow s)$,

$$x_1, \ldots, x_{n-1} \vdash s$$

By induction, we find $\vdash s$. □

Before we finish our discussion of semantics, we note that there are many other axioms systems which can be used to define a propositional calculus (in the sense that they prove all tautologies). Most interesting is the axiom system whose only connective is the sheffer stroke, and whose only axiom schema is

$$(B|(C|D))|[\{E|(E|E)\}|[(F|C)|((B|F)|(B|F))]]$$

and whose rule of inference is to infer D from $B|(C|D)$, and B. Of course, it is outlandish to attempt proofs in such a system, hence why we did not attempt this chapter using the system.

2.5 Multivalued Logics and Independence

The most basic assumption of boolean logic is that there are two truth values, $\bot$ and $\top$. Everything is either true or false. But what about the statement "God exists"? Until the skys open up, it is impossible to determine

whether this statement is true or false. If we model the statement with boolean values, we must decide whether the statement is true and false. It is the principle of three-valued logic, invented by Jan Łukasiewicz, who decided that some statements have a third 'truth value', that the statement is undecidable. After this introduction, there is nothing stopping us from considering a system with even more 'truth values', which evaluate statements based on the degree of truth, representing false, possible, justifiable, likely, etc, or something more subtler.

In classical logic, proof systems are built to 'preserve truth'. All axioms are tautologies, and anything deduced from tautologies are tautologies. How then, are we to evaluate proof systems for multi-valued logics? When creating proof systems for multi-valued logics, we hope the value of a statement which follows from other statements is less than or equal to the value of the original statement. We shouldn't be able to prove some 'more true' than what we know. For instance, consider Sorites paradox

> (1) "A grain of sand is not a heap of sand".
> (2) "Adding a grain of sand to something which is not a heap does not make it into a heap"
> (3) "A grain cannot be turned into a heap by adding grains."

We fix this argument by considering it in a multi-valued logic system. For each k, consider the statements

> $(1.k)$ "k grains of sand do not make a heap".
> $(2.k)$ "Adding a grain to k grains of sand doesn't make $k+1$ grains a heap"
> $(1.k+1)$ "$k+1$ grains do not make a heap."

we are only allowed to conclude $(1.k+1)$ from $(1.k)$ and $(2.k)$ when $(1.k+1)$ is 'less true' than $(1.k)$ and $(2.k)$. As $k \to \infty$, if we continue the argument, then we find the statement eventually becomes completely false, which is true, for eventually a certain number of grains do become a heap. This book is not intended to talk about non-standard logics in detail. Instead, we use the notion of multivalued logic to attack the notion of independance. Are the axioms of our Hilbert system minimal – do some of the axioms apply some of the other axioms? The trick to finding whether an axiom is independent is to find a truth system in which the other axioms are suitable, but the removed axiom is not. It follows that any statements which follows from the axioms are not removed.

Let us start by discarding (A1) from our axiom system. We may only derive results from (A2), (A3), and (MP). Can we derive (A1) using these rules? We shall try and set a value system on this proof system which satisfies the decreasing property of proofs. We shall consider 'truth assignments' based on the table

x	y	$H_{\Rightarrow}(x,y)$	$H_{\neg}(x)$
0	0	0	1
0	1	2	1
0	2	2	1
1	0	2	1
1	1	2	1
1	2	0	1
2	0	0	0
2	1	0	0
2	2	0	0

With this definition, we may consider truth values of formulae. Note that $x \Rightarrow (y \Rightarrow x)$ evaluates to 2 when x is 1 and y is 2. We shall show any provable formula from (A2) and (A3) always takes the value zero. One calculates that this is true of any instance of (A2) and (A3). Furthermore, if s and $s \Rightarrow w$ is always zero, then w is always zero. But this shows that $x \Rightarrow (y \Rightarrow x)$ can never be proven using (A2) and (A3).

To prove that (A2) is independent of (A1) and (A3), take the truth table

x	y	$H_{\Rightarrow}(x,y)$	$H_{\neg}(x)$
0	0	0	1
0	1	2	1
0	2	1	1
1	0	0	0
1	1	2	0
1	2	0	0
2	0	0	1
2	1	0	1
2	2	0	1

Then everything proved from((A1) and (A3) takes the value zero always, yet (A2) takes the value 2 when s is 0, w is 0, and u is 1.

We prove the independence of (A3) using a different trick, related to the removal of (A3) in intuitionistic logic. If s is a statement, consider $h(s)$, which is defined by

$$h(x) = x \qquad h((x \Rightarrow y)) = (h(x) \Rightarrow h(y)) \qquad h((\neg x)) = x$$

Then h removes all negations signs from s. If s is an instance of (A1) or (A2), then s is a tautology, and $h(s)$ is a tautology. Furthermore, if $h(w \Rightarrow u)$, and $h(w)$ are tautologies, then $h(w \Rightarrow u) = h(w) \Rightarrow h(u)$ is a tautology. Yet

$$h((\neg s \Rightarrow \neg w) \Rightarrow ((\neg s \Rightarrow w) \Rightarrow s)) = (s \Rightarrow w) \Rightarrow ((s \Rightarrow w) \Rightarrow w)$$

which is not a tautology, and is thus an instance of (A3) which cannot be proved by (A1) and (A2).

Chapter 3

First Order Logic

It is logical to conclude that "Julie is a human" and "Laura is a human" from the general statement that "All women are human"? In Propositional logic, we are unable to model this deduction in any satisfactory manner. We shall attempt to design a formal system mimicking these derivations. This is predicate logic.

3.1 Language

The syntax of predicate logic is less homogenous, for our language must contain nouns, like "Julie" and "Laura", and separate words describing statements about these nouns. These are known as **terms** and **quantifiers** respectively.

Terms should be able to model certain definite nouns, such as "Julie" and "Laura", variables such as X and Y, which can stand for many definite nouns, and relational nouns, such as "The school X went to", a statement which describes a different object for each interpretation of X. Definite nouns are known as **constants**, and relational nouns are known as **functions**. Functions will be separated based on their **arity**, the number of arguments they take. "X's favourite Y"is a '2-ary' function, "X's birthday" is a '1-ary' function. We shall now define the terms formally. Let Λ be a set of variables, Δ a set of constants, and for each n, a set Ψ_n of n-ary functions.

Definition. The set of **terms** is the smallest set $T(\Lambda, \Delta, \{\Psi_n\})$ such that $\Lambda, \Delta \subset T(\Lambda, \Delta, \{\Psi_n\})$, and if $s_1, \ldots, s_n \in T(\Lambda, \Delta, \{\Psi_n\})$, and f is a function in Ψ_n, then $f(s_1, \ldots, s_n) \in T(\Lambda, \Delta, \{\Psi_n\})$.

It is fairly easy to construct truth functional statements from these term. In addition to the usual connectives of sentential logic, we also require **predicates**, which are functions of nouns representing a statement about those nouns. For instance, "X is a Human" is a predicate. Predicates, like functions, are separated based on arity. For each n, let Π_n be a set of n-ary predicates. Given $\Lambda, \Delta, \{\Psi_n\}$, and $\{\Pi_n\}$, we shall construct

Definition. Fix a set of variables Λ, a set of constants Δ, a family of functions $\{\Psi_n\}$, and a family of predicates $\{\Pi_n\}$. An **atomic formula** is a string of the form $P(t_1, \ldots, t_n)$, where $P \in \Pi_n$, and $t_1, \ldots, t_n \in T(\Lambda, \Delta, \{\Psi_n\})$. Then the first-order language $\mathrm{FO}(\Lambda, \Delta, \{\Psi_n\}, \{\Pi_n\})$ is defined to be the smallest set containing all atomic formulae, which is also closed under the logical operations $\wedge, \vee, \Rightarrow, \neg, \Leftrightarrow$, as in sentential logic, and if $x \in \Lambda$ is a variable, and s is a statement, then $(\forall x : s)$ and $(\exists x : s)$ are formulae in the language.

Its a bit of faff to verify that one may interpret each statement of the language in a unique way. Given the readers experience, we leave them to fill in the syntactical details when needed. A modification of the brackets lemma used in the previous chapter will aid in this task.

In sentential logic, one may substitute arbitrary formulas for arbitrary variables, and the meaning of the statement will not change. In first order logic, things are more complicated. Consider the statement

"there exists X, such that if X is a man, then X is mortal"

First off, we cannot replace the initial X, for when we replace it with a definite nount the statement becomes nonsense. We may replace the other X's, but this changes the meaning of the statement

"there exists X, such that if Laura is a man, then Laura is mortal"

The problem results because the instances of X are faulty. Another case results when we substitute X for Y in the formula

"there is Y such that if X is a man, then Y is a dog"

The resulting substitution is

"there is Y such that if Y is a man, then Y is dog"

We shall perform these such of substitutions. Let s be an arbitrary string in a first order language. An occurrence of a variable x is **bound** in S if it occurs in a subformula $(\forall x : w)$ or $(\exists x : w)$. An occurrence is **free** if it is not bound, and is **free for a formula** s if it does not occur in any subformula of the form $(\forall y : w)$ or $(\exists y : w)$, where y is a free variable in s. A substitution $w[s_1/x_1, \ldots, s_n/x_n]$ is only valid when each x_i is free for s_i. This avoids the interpretation problems above. In the first example, X is bound, so cannot be substituted. In the second X is free, but is not free for Y.

3.2 Interpretation

Languages are defined in terms of the subject manner we wish to study, but may be interpreted in many different ways. For instance, the axioms which define the logic of group theory may be interpreted relative to whichever group we interpret the axioms as agreeing with. We would hope that a statement is true if and only if it is true in every interpretation of the axioms. To begin discussing this, we must precisely define what we mean by interpretation, as formulated by Alfred Tarski.

An **interpretation** M of a first order language $\mathrm{FO}(\Lambda, \Delta, \{\Phi_n\}, \{\Pi_n\})$ is a **universe of discourse** $M(U)$, and an interpretation of the relations; for each constant $c \in \Delta$, we have an associated element $c_M \in U$, for each function $f \in \Phi_n$, we have a function $f_M : U^n \to U$, and for each proposition $P \in \Pi_n$, we have an n-ary relation P_M on elements of U.

In sentential logic, when we assign a truth value to a set of variables, we may extend the definition of truth to all formulas. When we assign a meaning to each variable in a first order language, we may define a meaning on all terms, and from these meanings, assign truth to statements in the corresponding language. Consider a particular interpretation of a first order language with variables Λ, and consider an assignment $f : \Lambda \to U$.

Define $f_* : T(\Lambda, \Delta, \{\Psi_n\}) \to U$, by the recursive formulation. Let x be an arbitrary variable, c an arbitrary constant, and g an arbitrary formula,

$$f_*(x) = f(x) \qquad f_*(c) = c_M \qquad f_*(g(t_1, \ldots, t_n)) = g_M(f_*(t_1), \ldots, f_*(t_n))$$

Using this definition, we may define whether a formula is satisfied in a model of a first order theory. We shall now define what it means for an assignment to **satisfy** a formula in an interpretation. For simplicity, write $f[a/x]$ for the assignment

$$f[a/x](y) = \begin{cases} f(y) & y \neq x \\ a & y = x \end{cases}$$

1. An assignment f satisfies $P(t_1, \ldots, t_n)$ if $P_M(f_*(t_1), \ldots, f_*(t_n))$ holds.
2. An assignment f satisfies $(\forall x : s)$ if s is satisfied by all $f[a/x]$, for all a in the universe of discourse. A statement satisfies a formula $(\exists x : s)$ if there is some a such that $f[a/x]$ satisfies s.
3. An assignment f satisfies $s \circ u$ or $\neg s$, where $\circ$ and $\neg$ are logical connectives, exactly how they were defined in sentential logic.

If M is an interpretation, then a formula s is **valid** for an interpretation, denoted $\models_M s$, if s is true under every assignment under M. A statement is false if it is true under no interpretation, or alternatively, if the negation of the statement is true under the interpretation. An interpretation is a **model** for a set of formulas Γ if every formula in Γ is true for the interpretation.

Lemma 3.1. *If $\models_M s$ and $\models_M s \Rightarrow w$, then $\models_M w$.*

Proof. If f satisfies s and $s \Rightarrow w$, then f satisfies w. □

Lemma 3.2. *If a formula s contains free variables $x_1, \ldots, x_n$, and two assignments f and g agree on the free variables, then f satisfies s if and only if g satisfies s.*

Lemma 3.3. *$\models_M (\exists x : s)$ if and only if $\models_M \neg(\forall x : \neg s)$.*

Proof. If an assignment f satisfies $(\exists x : s)$, then s is satisfied by some $f[a/x]$. But then f does not satisfy $(\forall x : \neg s)$ for $f[a/x]$ does not satisfy $\neg s$. Conversely, if an assignment f does not satisfy $(\exists x : s)$, then every $f[a/x]$ satisfies $\neg s$. □

Lemma 3.4. *$\models_M s$ if and only if $\models_M (\forall x : s)$.*

Proof. If $\models_M s$, then every assignment f satisfies s, so certainly every $f[a/x]$ satisfies s, and thus f satisfies $(\forall x : s)$, hence $\models_M (\forall x : s)$. Conversely, suppose $\models_M (\forall x : s)$. Then, every assignment satisfies $(\forall x : s)$, and thus in particular satisfies s. □

Let s contain free variables $x_1, \ldots, x_n$. The **closure** of s is the string $(\forall x_1 : (\forall x_2 : \ldots (\forall x_n : s)) \ldots))$. We then verify by induction that a formula is satisfied by an interpretation if and only if its closure is.

Theorem 3.5. *Consider a form of sentential logic, whose variables are all atomic formulas, and formulas of the form $(\forall x : s)$, and $(\exists x : s)$. Then if a statement is a tautology, then it is satisfied under all interpretations.*

Proof. The connectives of predicate logic not summarized above are exactly the connectives of sentential logic. Thus regardless of how f satisfies the formulas treated atomically in sentential logic, f will satisfy the complete tautology. □

Theorem 3.6. *If s contains no free variables, then either $\models_M s$ or $\models_M \neg s$.*

Proof. This follows from the fact that any two assignments that agree on the free variables of a formula agree on the satisfiability. Thus if there are no free variables, all assignments agree, and in particular all satisfy the formula or all do not satisfy the formula. □

Lemma 3.7. *If t and u are terms, x is a variable, and f is an assignment, then*

$$f[f_*(u)/x]_*(t) = f_*(t[u/x])$$

Proof. If t is a variable unequal to x, or t is a constant, then

$$f[f_*(u)/x]_*(t) = f(t) = f_*(t[u/x])$$

If $t = x$, then

$$f[f_*(u)/x]_*(t) = f_*(u) = f_*(t[u/x])$$

For a structural induction, let $t = g(t_1, \ldots, t_n)$. Then

$$\begin{aligned} f[f_*(u)/x]_*(t) &= g_M(f[u/x]_*(t_1), \ldots, f[u/x]_*(t_n)) \\ &= g_M(f_*(t_1[u/x]), \ldots, f_*(t_n[u/x])) = f_*(t[u/x]) \end{aligned}$$

Thus the theorem holds in general. □

Lemma 3.8. *f satisfies $s[u/x]$ if and only if $f[f_*(u)/x]$ satisfies s.*

Proof. If s is $P(t_1,\ldots,t_n)$. Then $s[u/x] = P(t_1[u/x],\ldots,t_n[u/x])$, and

$$P_M(f_*(t_1[u/x]),\ldots,f_*(t_n[u/x])) = P_M(f[f_*(u)/x]_*(t_1),\ldots,f[f_*(u)/x]_*(t_n))$$

Which shows that f satisfies $s[u/x]$ if and only if $f[f_*(u)/x]$ satisfies s. If s is formed by standard sentential connectives, the theorem is trivial. If $s = (\forall y : w)$, where $y \neq x$, then $s[u/x] = (\forall y : w[u/x])$, and by definition, f satisfies $s[u/x]$ if and only if $f[a/y]$ satisfies $w[u/x]$ for all a, which by induction implies that $f[f_*(u)/x][a/y]$ satisfies w for all a, so $f[f_*(u)/x]$ satisfies s. Similar results hold if s's primitive connective is the existential quantifier. □

Theorem 3.9. *For any formula s and term t free for x, $\models_M (\forall x : s) \Rightarrow s[t/x]$*

Proof. Let f be an assignment satisfying $(\forall x : s)$. Then $f[f_*(t)/x]$ satisfies s, so f satisfies $s[t/x]$. □

Theorem 3.10. *If s does not contain x as a free variable, then*

$$\models_M (\forall x : s \Rightarrow w) \Rightarrow (s \Rightarrow (\forall x : w))$$

Proof. If f satisfies $(\forall x : s \Rightarrow w)$ and s, then $f[a/x]$ satisfies $s \Rightarrow w$, and since s does not contain x as a free variable, $f[a/x]$ also satisfies s, so f satisfies $(\forall x : w)$. □

Given a specific interpretation with universe of discourse U, we enhance the first order language we are discussing to include U as constants. The interpretation naturally extends to this new language. Thus we can talk of whether our interpretation satisfies a formula

$$\models_M s[u_1/x_1,\ldots,u_n/x_n]$$

Where $u_1,\ldots,u_n \in U$ are formulas containing elements of U. If the free variables of a formula s are $x_1,\ldots,x_n$, then the set of $u_1,\ldots,u_n$ for which the equation above holds will be called the relation relative to s.

A statement is **logically valid** if it is true under all possible interpretations. A statement is **satisfiable** if it is true under at least one interpretation, and **contradictory** if it is false under every interpretation. A set of statements is satisfiable if they are all true under a single interpretation. A statement s is a **logical consequence** of a set of statements Γ if every interpretation which satisfies every statement of Γ also satisfies s.

3.3 First Order Formal Systems

We have the syntax and semantics to begin discussing formal systems in the first-order languages. We would like to find axioms which only find theorems satisfied by every model of the system, a **sound** axiom system. Even better, if we can find **complete** axioms, which can proved anything satisfied by every model of the system. Axioms are essential, because it turns out that determining whether a formula is satisfied by all models of the system is an uncomputable problem. We shall use an axiom consisting of five schemata, the original (A1), (A2), and (A3) found in predicate logic, as well as two new first order schema (A4) and (A5). Let s and w be formula, and t a term substitutable for a variable x. Then (A4) is

$$(\forall x : s) \Rightarrow s[t/x]$$

provided that x does not occur freely in s, we have (A5)

$$(\forall x : s \Rightarrow w) \Rightarrow (s \Rightarrow (\forall x : w))$$

Our rules of inference are modus ponens (MP), to infer w from s and $s \Rightarrow w$, as well as universal generalization (UG), inferring $(\forall x : s)$ from s. As in predicate logic, we shall let $\vdash s$ stand for s is a theorem of the system.

From the theorems we proved about the semantics of first order logic, we already know that this system is sound, a simple use of structural induction. Now we show that the system is complete. Our system includes all the rules of propositional calculus, which justifies a very useful property. Let s be a formula in first order logic, and replace it with a corresponding formula in sentential logic by replacing all the main occurences of the quantifiers $(\forall x : s)$ with variables separate from the rest of the variables occuring in s. Then if the formula obtained is a tautology, s is provable in first order logic.

Example. *Find easy example of tautology usage to insert here!*

The deduction theorem cannot be carried directly over to first order logic, since $s \vdash (\forall x : s)$ is always true, yet $\vdash s \Rightarrow (\forall x : s)$ cannot be a theorem of first order logic.

Example. *Consider the formula $P(x) \Rightarrow (\forall x : P(x))$. Take a model M whose universe consists of two elements a and b, and let P_M only be satisfied by a. Then, under the assignment $f(x) = a$, $P(x)$ is satisfied, but $(\forall x : P(x))$ is not.*

Part II

Computability

In 1931, Kurt Gödel proved all sufficiently complicated axiomatic systems had unprovable theorems, but a fundamental question remained; how was one to decide whether a theorem could be proved? It took a decade for Alonzo Church and Alan Turing to deduce the impossibility of such a claim. Fifty years later, 'theoretical computation' had become a common reality. We shall describe mathematical models of computation which have been developed to analyze the limitations of various computational methods.

Turing and Church's major breakthrough was precisely defining a 'computational procedure'. It is often the case that precise definitions give rise to easy proofs of the most surprising consequence. Philosophically, one should be able to define a procedure without reference to a computer, for humans computed long before microchips. On the other hand, models should reflect physical reality, since one needs a physical mechanism in order to computer, whether electronic or mental. If your computational model is too strong or too weak, it will not accurately represent our limitations.

We will begin by analyzing the automaton, a model of computation without stored memory. We will expand the amount of expression of the automaton by considering context-free grammars. Finally, we add memory by considering a Turing machine. It is the Church Turing thesis that this is the ultimate model of computation – any real world computation can be modelled as an action on a Turing machine. From this model, and with the hypothesis of Church and Turing, we can make precise, philisophically interesting statements about the nature of computation in the real world.

As in mathematical logic, the objects of study are strings of symbols over a certain alphabet. One studies the notion of computation syntactically. One of the main ideas of computability theory is that a mental decision can be modelled as a **decision problem** – find a computational model which will 'accept' certain strings over an alphabet. Suppose our problem is to verify whether the addition of two numbers is correct. We are given a, b, and c, and we must decide whether $a + b = c$. Our symbol set is $\{0, 1, \ldots, 9, :\}$, and we wish to model a computation which accepts all strings of the form "$a : b : c$", where a, b, and c are decimal strings for which $a + b = c$. Thus we must design machine to accept strings in a specified language, and determining whether a problem is solvable reduces to studying the structure of languages.

As a more dynamic discipline than mathematical logic, we need more operations on strings to obtain languages from other languages. We obviously need concatenation, but also **reversal**, which will be denoted s^R. These operations are extended to languages by applying the operations on a component by component basis:

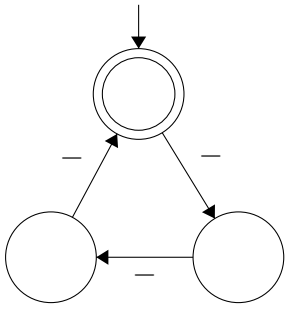
$$S \circ W = \{s \circ w : s \in S, w \in W\} \qquad S^R = \{s^R : s \in S\}$$

A **palindrome** is a string s for which $s^R = s$. If Σ is a set of strings, we shall let $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$.

Chapter 4

Finite State Automata

Our initial model of computability is a computer with severely limited, finite amount of memory. Surprisingly, we shall still be able to compute a great many things. The idea of this model rests on explicitly representing memory as a finite amount of states, whose behaviour is uniquely determined by the state upon which it stands at a point of time. We can represent this process via a state diagram. Suppose we would like to describe an algorithm determining if a number is divisible by three. We shall represent a number by a string of dashes. For instance, "$-----$" represents the number 5. We describe the algorithm in a flow chart below



The algorithm proceeds as follows. Begin at the top node. we proceed clockwise around the triangle, moving one spot for each dash we see. If, at the end of our algorithm, we end up back at the top node, then the number of dashes we have seen is divisible by three. The basic idea of the finite automata is to describe computation via these flow charts – we follow a string around a diagram, and if we end up at a 'accept state', then

we accept the string. A mathematical model for this description is a finite state automaton.

A **deterministic finite state automaton** is a 5-tuple $(Q, \Sigma, \Delta, q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet, $q_0 \in Q$ is a start state, $F \subset Q$ are the accept states, and $\Delta : Q \times \Sigma \to Q$ is the transition function. A finite state machine 'works' exactly how our original algorithm worked. Draw a directed graph whose nodes are states, and draw an edge between a state q and each related state $\Delta(q, \sigma)$, for each symbol $\sigma \in \Sigma$. Take a string $s \in \Sigma^*$. We begin at the start state q_0. Sequentially, for each symbol in s, we follow the directed edge from our current state to the state on the edge related to the current symbol in s. If, we end at an accept state in F, then s is an 'accepted' string. Formally, we define this method by extending Δ to $Q \times \Sigma^*$. We define, for $s \in \Sigma^*$, $t \in \Sigma$,

$$\Delta(q, \varepsilon) = q \qquad \Delta(q, st) = \Delta(\Delta(q, s), t)$$

A state machine M **accepts** a string s in Σ^* if $\Delta(s, q_0) \in F$. We call the set of all accepting strings the **language** of M, and denote the set as $L(M)$. A subset of Σ^* which is a language of a deterministic finite state automata is known as a **regular language**.

Example. *Consider $\Sigma = \{-\}$. Then the set of all 'dashes divisible by three' is regular, as in the introductory diagram. Formally, take*

$$Q = \mathbf{Z}_3 \qquad \Delta(x, -) = x + 1 \qquad q_0 = 0 \qquad F = \{0\}$$

then $(Q, \Sigma, \Delta, q_0, F)$ recognizes dashes divisible by three. The 'graph' of the automata is exactly the graph we've already drawn.

Arithmetic is closed under certain operations. Given two numbers, we can add them, subtract them and multiplication, and what results is still a number. In the theory of computation, the operations have a different flavour, but are nonetheless just as important. We shall find that all regular languages can be described from very basic languages under certain compositional operators, under which the set of regular languages is closed.

Theorem 4.1. *If $A, B \subset \Sigma^*$ are regular languages, then $A \cup B$ is regular.*

Proof. let $M = (Q, \Sigma, \Delta, q_0, F)$ and $N = (R, \Sigma, \Gamma, r_0, G)$ be automata recognizing A and B respectively. We shall define a finite automata recognizing $A \cup B$. Define a function $(\Delta \times \Gamma) : (Q \times R) \times \Sigma \to (Q \times R)$, by letting

$$(\Delta \times \Gamma)(q, r, \sigma) = (\Delta(q, \sigma), \Gamma(r, \sigma))$$

Consider

$$H = \{(q, r) \in S : q \in F \text{ or } r \in G\}$$

We contend that

$$((Q \times R), \Sigma, \Delta \times \Gamma, (q_0, r_0), H)$$

recognizes $A \cup B$. By induction, one verifies that for any $s \in \Sigma^*$,

$$(\Delta \times \Gamma)(q, r, s) = (\Delta(q, s), \Gamma(r, s))$$

Thus $(\Delta \times \Gamma)(q_0, r_0, s) \in H$ if and only if $\Delta(q_0, s) \in F$ or $\Gamma(r_0, s) \in G$. □

Theorem 4.2. *If A is a regular language, then A^c is regular.*

Proof. If $M = (Q, \Sigma, \Delta, q_0, F)$ recognizes A. Then define a new machine $N = (Q, \Sigma, \Delta, q_0, F^c)$. The transition $\Delta(q_0, s)$ is in F^c if and only if $\Delta(q_0, s)$ is not in F. □

Corollary 4.3. *If A and B are regular languages, then $A \cap B$ are regular.*

Proof. $A \cap B = (A^c \cup B^c)^c$. □

4.1 Non Deterministic Automata

An important concept in computability theory is the introduction of non-determinism. Deterministic machines must follows a set protocol when understanding input. Non deterministic machines can execute one of many different specified protocols. If any of the protocols accepts the input, then the entire machine accepts the input. Thus non-deterministic machines are said to multitask, for they can be seen to run every protocol specified at once, checking one of a great many protocols to see a pass. An alternative viewpoint is that the machines make a lucky guess – they always seem to choose the write protocol which results in an accepted string.

A **non-deterministic finite state automaton** is a 5-tuple $(Q, \Sigma, \Delta, q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet, q_0 is the start state, $F \subset Q$ are the accept states, and $\Delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ is the non-deterministic transition function.

In a non-deterministic finite state automata, we **accept** a string s if $s = s_1 \dots s_n$, where each $s_i \in \Sigma_\varepsilon$, and there are a sequence of states $t_0, \dots, t_{n+1}$, with $q_0 = t_0$ and $t_n \in F$, such that $t_{k+1} \in \Delta(t_k, s_k)$. The set of accepting strings of a machine M form the language $L(M)$. We draw a graph with nodes Q, and with directed edges v to w if $w \in \Delta(v, \Sigma_\varepsilon)$. We begin at q_0. For a string s, we attempt to find a path from q_0 to an accept state, by following edges whose corresponding symbol is in s (or whose symbol is ε, in which we get for free). The string is accepted if such a path is possible. Some call non-deterministic methods a lucky guess methods, since they always make a lucky guess of which deterministic path to take to accept a string.

There is a nicer criterion of acceptance than described above, which is easier to work with in proofs. First, assume there are no ε-transitions in a machine M; that is, $\Delta(q, \varepsilon) = \varnothing$ for all states q. We may then extend $\Delta : Q \times \Sigma \to \mathcal{P}(Q)$ to $\Delta : \mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(Q)$ recursively by

$$\Delta(X, \varepsilon) = X \qquad \Delta(X, st) = \Delta(\Delta(X, s), t)$$

where $\Delta(X, s) = \{\Delta(x, s) : x \in X\}$. A string s is accepted by M if and only if an accept state is an element of $\Delta(q_0, s)$. If $s = s_1 \dots s_n$, and there are $t_0, \dots, t_n$ with $t_0 = q_0$, t_n an accept state, and $t_{k+1} \in \Delta(t_k, s_k)$, then by induction one verifies that $t_n \in \Delta(q_0, s)$. Conversely, an induction on s verifies that if $q \in \Delta(q_0, s)$, and if $s = s_1 \dots s_n$, then there is a state q' with $q' \in \Delta(q_0, s_1 \dots s_{n-1})$, $q \in \Delta(q', s_n)$. But this implies that if there is an accept state in $\Delta(q_0, s)$, then s is accepted by M. We can always perform this trick, for we may always remove ε transitions.

Lemma 4.4. *Every non-deterministic automata is equivalent to a non deterministic automata without ε transitions, in the sense that they both recognize the same language.*

Proof. Consider a non-deterministic automata.

$$M = (Q, \Sigma, \Delta, q_0, F)$$

Define a state u to be ε reachable from t if there is a sequence of states $q_0, \dots, q_n$ with $q_0 = t$, $q_n = u$, and $q_{i+1} \in \Delta(q_i, \varepsilon)$. Let $E(u)$ be the set of all

states ε reachable from u. Define

$$N = (Q, \Sigma, \Delta', q_0, F)$$

Where

$$\Delta'(q,s) = \begin{cases} \bigcup_{u \in E(q)} \Delta(u,s) & s \neq \varepsilon \\ \varnothing & s = \varepsilon \end{cases}$$

Then it is easily checked that $L(N) = L(M)$, for we may skip over ε transitions in N. □

It seems to be a much more complicated procedure to find if a string is accepted by a non-deterministic automata, but it turns out that every non-deterministic automata can be converted into a deterministic automata. The proof relies on the fact that we may exploit the operations of non-determinism, described using power sets of a set.

Theorem 4.5. *A non-deterministic finite state automata language is regular.*

Proof. Let $M = (Q, \Sigma, \Delta, q_0, F)$ be a non-deterministic automata. Assume M has no ε transitions, without loss of generality. Let

$$N = (\mathcal{P}(Q), \Sigma, \Gamma, \{q_0\}, \{S \in \mathcal{P}(Q) : S \cap F \neq \varnothing\}$$

where

$$\Gamma(S,t) = \Delta(S,t)$$

We have already verified this deterministic machine recognizes $L(M)$, for $\Delta(\{q_0\}, s)$ contains an accept state if and only if s is accepted. □

It is now fair game to use non-deterministic automata to understand regular languages, for the language of every non-deterministic automata is regular.

Theorem 4.6. *If A and B are regular languages, then $A \circ B$ is regular.*

Proof. Let $M = (Q, \Sigma, \Delta, q_0, F)$ and $N = (R, \Sigma, \Gamma, r_0, G)$ be deterministic automata accepting A and B respectively. Without loss of generality, assume Q is disjoint from R. Consider the non-deterministic machine

$$O = (Q \cup R, \Sigma, \Pi, q_0, G)$$

Define

$$\Pi(s,t) = \begin{cases} \{\Delta(s,t)\} & s \in Q, \text{ and } t \neq \varepsilon \text{ or } s \notin F \\ \{\Delta(s,t), r_0\} & s \in F, t = \varepsilon \\ \{\Gamma(s,t)\} & s \in R \\ \varnothing & \text{otherwise} \end{cases}$$

We quickly verify that if $s \in L(M)$ and $w \in L(N)$, then $sw \in L(O)$. If $v \in L(O)$, there is some substring k such that $r_0 \in \Pi(q_0, k)$ (for this is the only path from q_0 to G). If we choose k to be the shortest such string, then k must also be an accept string in $L(M)$, since any substring of k cannot map to states in R, and thus k must map via the ε transition from an accept state of M. If $v = kr$, then $\Pi(r_0, r)$ is an accept state in N, so $r \in L(N)$. Thus $v \in L(M) \circ L(N)$. □

Theorem 4.7. *If A is a regular language, A^* is regular.*

Proof. Let $M = (Q, \Sigma, \Delta, q_0, F)$ be a deterministic language which accepts A. Form a non-deterministic automata $N = (Q \cup \{i\}, \Sigma, \Gamma, i, F)$, where

$$\Gamma(q,s) = \begin{cases} \{\Delta(q,s)\} & s \neq \varepsilon \\ \{q_0\} & s = \varepsilon, q = i \\ \{i\} & s = \varepsilon, q \in F \\ \varnothing & \text{otherwise} \end{cases}$$

Then $L(N) = A^*$, for if $s = a_1 \dots a_n$, with $a_i \in A$, then we may go from i to q_0, then to an accept state via a_1, return to i with a ε transition, and continue, so s is accepted. If we split a string accepted to $L(N)$ into when ε transitions to i are used, then we obtain strings in A. □

In turns out that the operations of concatenation, union, and 'kleene starification' are enough to describe all regular languages. Thus we may take an algebraic approach to understanding regular languages, using the symbology of regular expressions.

4.2 Regular Expressions

Automata are equivalent to a much more classical notion of computation – regular expressions.

Definition. A **regular expression** over an alphabet Σ is the smallest subset of $(\Sigma \cup \{\varnothing, \cup, *, (,)\})^*$ such that

1. $\varnothing$ is a regular expression, as are $s \in \Sigma^*$.
2. If λ and γ are regular expressions, then so are $\lambda^*, (\lambda \cup \gamma)$, and $(\lambda \circ \gamma)$.

Every regular expression describes a language. A string $s \in \Sigma^*$ is recognized by an regular expression λ if

1. $\lambda \in \Sigma^*$, and $s = \lambda$.
2. $\lambda = (\lambda \cup \gamma)$, and s is recognized by λ or by γ.
3. $\lambda = \gamma \circ \delta$, and $s = wl$, where w is recognized by γ, and δ recognizes l.
4. $\lambda = \gamma^*$, and $s = \varepsilon$ or $s = s_1 \dots s_n$, where s_i is recognized by γ.

The regular language corresponding to a regular expression λ is $L(\lambda)$, the set of strings recognized by t. The set of all regular expressions on an alphabet Σ will be denoted $R(\Sigma)$.

It is a simple consequence of our discourse that every language recognized by a regular expression *is actually* a regular language. In fact, we can show that every regular language is described by a regular expression. We shall describe an algorithm for converting a finite state automaton to a regular expression. We shall make a temporary generalization, by allowing non deterministic finite automata to have regular expressions in their transition functions. A **generalized non-deterministic finite state automaton** is a 5-tuple $M = (Q, \Sigma, \Delta, q_0, f_0)$, where $\Delta : Q - \{f_0\} \times Q - \{q_0\} \to R(\Sigma)$ is the generalized transition function, and q_0 is the start state, f_0 is the end state. A generalized automaton accepts $s \in \Sigma^*$ if we may write $s = s_1 \dots s_n$, and there are a sequence of states $q_0, \dots, q_n$ where $q_n = f_0$, and $\Delta(q_i, q_{i+1})$ recognizes s_i.

Theorem 4.8. *Any generalized finite-state automaton describes the language of a regular expression.*

Proof. If the generalized automaton has two states, then there is only one transition from start state to begin state, and this transition describes a regular expression for the automaton. We will reduce every automaton to this form by induction. Suppose an automaton has n states. Fix a state $q \in Q$, which is neither the beginning or accepting state. Define a new automaton

$$N = (Q - \{q\}, \Sigma, \Delta', q_0, f_0)$$

where $\Delta'(a,b) = (\Delta(a,b) \cup \Delta(a,q)\Delta(q,q)^*\Delta(q,b))$ Then N is equivalent to M, and has one fewer state, and is thus equivalent to a regular expression. □

Corollary 4.9. *Every regular language is described by a regular expression.*

Proof. Clearly, every DFA and NFA is equivalent to a generalized NFA, for by adding new states, we may ensure no states map back to the start state, and that there is only one end state. □

4.3 Limitations of Finite Automata

We've discovered a menagerie of different problems we can solve with finite automata, but it has already been foreshadowed that better machines await. Here we discover methods which attack languages, showing that they cannot be recognized by regular expressions, not finite automata.

Theorem 4.10 (Pumping Lemma)**.** *Let L be a regular language. Then there is a number p, called the pumping length, such that any $s \in L$ which satisfies $|s| \geqslant p$, then we may write $s = wuv$, where $|u| > 0$, $|wu| \leqslant p$, and $wu^iv \in L$ for all $i \geqslant 0$.*

Proof. Let L be a regular language, and M a deterministic automata recognizing L with p states. Let s be a string with $|s| \geqslant p$, with $s \in L(M)$. Write $s = s_1 \dots s_n$ and let $q_k = \Delta(q_0, s_1, \dots, s_k)$. Then we obtain $|s| + 1$ states $q_0, q_1, \dots, q_{|s|}$. By the pidgeonhole principle, since q_i equals some q_j, for $i < j$. Let $w = s_1 \dots s_i$, $u = s_{i+1} \dots s_j$, $v = s_{j+1} \dots s_{|s|}$. Then $\Delta(\Delta(q_0, w), u) = \Delta(q_0, w)$, so

$$\Delta(q_0, wu^iv) = \Delta(q_0, wuv)$$

So $wu^iv \in L$ for all i. □

Example. *$L = \{0^k 1 0^k : k \geqslant 0\}$ is not regular. If it was regular, it would have a pumping length p. Since $0^p 1 0^p$ is in L, so we may write $0^p 1 0^p = wuv$, where $|wu| \leqslant p$, and $wu^i v \in L$. Then $u = 0^k$, for $k > 0$, and $wv = 0^{p-k} 1 0^p \in L$, which is clearly not in the language, contradicting regularity.*

Example. *$L = \{1^{n^2} : n \in \mathbf{N}\}$ is not regular. Suppose we had a pumping length p. Then $1^{p^2} \in L$, so there is $0 < k \leqslant p$ with $1^{p^2+k} \in L$. But*

$$(p+1)^2 = p^2 + 2p + 1 > p^2 + k$$

] And there is no perfect square between p^2 and $(p+1)^2$, a contradiction.

Example. *The language $L = \{0^i 1^j : i > j\}$ is not regular. If we had a pumping lenght p, then $0^p 1^{p-1} \in L$. But then there is $0 < k \leqslant p$ such that $0^{p+(i-1)k} 1^{p-1} \in L$ for all natural numbers i. In particular, for $i = 0$, we find $0^{p-k} 1^{p-1} \in L$ But $p - k \leqslant p - 1$, a contradiction.*

There is a much more mathematically elegant and complete way of separating regular languages from non-regular ones, discovered by John Myhill and Anil Nerode. Consider an alphabet Σ, and a particular language $L \subset \Sigma^*$. Call two strings a and b in Σ^* L-indistinguishable if $az \in L$ if and only if $bz \in L$ for any $z \in \Sigma^*$. This forms an equivalence relation on Σ^*. We shall define the index of L, denoted Ind L, to be the cardinality of the partition.

Theorem 4.11 (Myhill-Nerode)**.** *L is regular if and only if Ind $L < \infty$, and Ind L is the number of states of the smallest finite state machine to recognize L.*

Proof. Let M be a deterministic finite state machine recognizing L with n states, transition function Δ, and start state q_0. Let $a_1, \ldots, a_{n+1}$ be $n+1$ strings in Σ^*. We claim that at least one pair is indistinguishable. if we take $q_i = \Delta(q_0, a_i)$, then some $q_i = q_j$. These two strings are then indistinguishable in L. Conversely, suppose that Ind $L < \infty$. Let $A_1, \ldots, A_n$ be the equivalence classes of Σ^*. If $s \in A_i$ is contained in L, then every other $w \in A_i$ is in L, for otherwise s and w can be distinguished. Build a finite state machine whose states are A_i, whose start state is $[\varepsilon]$, and whose transition function is

$$\Delta([s], t) = [st]$$

$[s]$ is accepted if and only if $s \in L$. This finite state machine recognizes s. □

In some circumstances, the Myhill Nerode theorem is very powerful.

Example. *For $\Sigma = \{a, b, \ldots, z\}$, consider the set L of words w whose last letter has not appeared before. For example, the words "apple", "google", "k", and ε are in L, but the words "potato" and "nutrition" are not. Is this language regular? We apply Myhill Nerode. If the letters in one word are different to the letters in another word, these words are distinguishable. If the letters in one word are the same as the letters in another word, and both are not accepted or both are not accepted, these words are indistinguishable. Thus the index of the language in consideration is the same as the number of different subsets of the set of letters in a word, counted twice for repeated and non-repeated characters. Subtracting one from the fact that ε need only be counted once, we find that $2 \cdot 2^{26} - 1 = 2^{27} - 1$. Since this is finite, the language is regular, and this is the minimal number of sets in a finite state machine recognizing the language.*

4.4 Function representation

Decision problems are sufficient to model a large variety of computational models, but for completeness, we should at least mention how we determine whether problems with multiple outputs can be. The counterpart of a finite-state automata is a finite-state transducer. A **finite state transducer** is a 5 tuple $(Q, \Sigma, \Lambda, \Delta, \Gamma, q_0)$, where Q is the set of states, Σ is the input alphabet, Λ is the output alphabet, $\Delta : Q \times \Sigma \to Q$ is the transition function, $\Gamma : Q \times \Sigma \to \Lambda_\varepsilon$ is the transduction function, and q_0 is the start state. Each finite-state transducer M gives rise to a function $f_M : \Sigma^* \to \Lambda^*$, defined by

$$f_M(\varepsilon) = \varepsilon \qquad f_M(st) = f_M(s) \circ \Gamma(\Delta(q_0, s), t)$$

A **regular function** is one computable by a finite state transducer.

Example. *Consider an alphabet $\mathbf{F}_2$ and consider the function f taking and returning strings over $\mathbf{F}_2$, inverting the strings on even positions, and leaving the strings on the odd positions. Then f is a regular function, for it may be computed by the automata below.*

Chapter 5

Context Free Languages

Finite state machines are useful on a specific set of problems, but have limitations. We would like our notion of computability to decide on a much more complicated set of problems. Thus we must define new classes of computable languages.

5.1 Context Free Grammars

Definition. A **Context Free Grammar** is (V, Σ, R, S), where V is a set of variables, Σ is a character set, disjoint from V, R is a relation between V and $(V \cup \Sigma)^*$, and $S \in V$ is the start variable. We call elements of R derivation rules, and write $(a, s) \in R$ as $a \to s$.

A string of the form uvw is **directly derivable** from uAw if $A \to v$ is a derivation rule. The smallest transitive relation containing the 'directly derivable' relation is the derivability relation. To reiterate, a string u is **derivable** from w if there is a sequence of direct derivations

$$w \to s_0 \to \cdots \to s_n \to u$$

Such a sequence is known as a **derivation**. The language of a grammar is the set of all strings in Σ^* derivable from the start state S. As in finite automata, the language of a grammar G will be denoted $L(G)$. A language is **context-free** if it is the language of a context free grammar.

Lemma 5.1. *Let $G = (V, \Sigma, R, S)$ and $H = (V, \Sigma, R', S)$ be two different context free languages. If every $(v, s) \in R'$ is a derivation in G, then $L(H) \subset L(G)$.*

Proof. If we can show that every direct derivation in H is a derivation in G, then all derivations in H are derivations in G, since derivations form the smallest transitive relation containing the direct derivations in H, and the derivations in G certainly satisfy this. If w is directly derived from s in H, then there is a rule (B, u), $s = rBt$, and $w = rut$. There is a sequence $s_0 \dots s_n$ deriving u from B in G. But then $(rs_0t) \dots (rs_nt)$ is a derivation of w from s, so w is derivable from s in G. □

A leftmost derivation is a derivation which always swaps out the leftmost variable. A string is **ambiguous** if it has two different leftmost derivations. A grammar is ambiguous if its language contains an ambiguous string. Ambiguity is unfortunate when parsing a language, since it means we may be able to interpret elements of the language in two different ways.

Example. *First order logic can be defined as an unambiguous grammar. To develop the language, we took a bottom up approach, but a top up approach can also be taken. We must take a finite alphabet to develop the language, which we take to be*

$$\{(,), \forall, \exists, \neg, \wedge, \vee, \Rightarrow, x, f, P, 0, 1, \dots, 9\}$$

we cannot take an 'infinite number of variables', in the sense of an infinite number of symbols, for then formal language theory does not apply. We must instead assume our variables, predicates, and functions are themselves words in a finite alphabet. For instance, we will enumerate our variables

$$\Lambda = \{x, x_0, x_{00}, \dots, x_{0000000000}, \dots\}$$

The trick to forming functions and predicates is to put n ones after an f or a P to denote that is is an n-ary function, so

$$\mathcal{F}^n = \{f^{111\dots11}, f^{111\dots11}{}_0, \dots, f^{111\dots11}{}_{000000000}, \dots\}$$

$$\mathcal{P}^n = \{P^{1111\dots11}, \dots\}$$

Then we may form predicate logic as a context free grammar. The variables are easiest to form

$$X \to x \mid X_0$$

Terms are tricky to define because we must form functions as well. The trick is to introduce new variables Y and Z which add enough terms to the function.

$$T \to X \mid f^1 U)$$
$$U \to V(T \mid {}^1 U, T$$
$$V \to \varepsilon \mid V_0$$

Finally, we form the formulas of the calculus. Again, the only trick part are the atomic formulae

$$F \to (F \wedge F) \mid (F \vee F) \mid (\neg F) \mid (F \Rightarrow F) \mid (\forall x : F) \mid (\exists x : F) \mid P^1 Z)$$
$$Y \to Z(T \mid {}^1 Y, T$$
$$Z \to \varepsilon \mid Z_0$$

We showed the formulas are derived unambiguously, but this took a lot of hard work. It is impossible to find a general procedure to decide whether a language is ambiguous, which is what makes verifying ambiguity so difficult.

It is useful to put grammars in a simple form for advanced theorems. A grammar (V, Σ, R, S) is in **Chomsky Normal Form** if the only relations in R are of the form

$$S \to \varepsilon \quad A \to BC \quad A \to a$$

where $A, B, C \in V$ and $B, C \neq S$, and $a \in \Sigma$.

Theorem 5.2. *Every context-free language can be recognized by a context-free grammar in Chomsky normal form.*

Proof. We shall reduce any context free grammar $G = (V, \Sigma, R, S)$ to a context free grammar in normal form in a systematic fashion, adding each restriction one at a time.

1. **No derivation rules map to the start variable**:
   Create a new start variable mapping onto the old start variable.

2. **There are no ε-transition rules except from the start variable**:
   Define a variable $A \in V$ to be nullable if we may derive ε from A. Let W be the set of all nullable variables. Define a new language $G'(V, \Sigma, R', S)$ such that, if $A \to A_1 \dots A_n$ is a derivation rule in G, and

$A_{i_1}, \ldots, A_{i_m}$ are nullable, then we add 2^m new rules to G' by removing some subset of the A_{i_k}. Then $L(G') = L(G) - \{\varepsilon\}$, so that if $\varepsilon \in L(G)$, we need only add an ε rule to S to make the two languages equal.

We will prove that if A is a variable in G', then A can derive $w \in \Sigma^*$ in G' if and only if it can derive it in G and $w \neq \varepsilon$. One way is trivial, the other a proof by induction on the length of the derivation. Suppose we have a derivation in G

$$A \to s_0 \to \cdots \to s_n \to w$$

Let $s_0 = A_1 \ldots A_n$. Then each A_i derives w_i in G, where $w = w_1 \ldots w_n$. We can choose such a derivation to be shorter than the derivation of G. But this implies that A_i derives w_i in G', provided $w_i \neq \varepsilon$. Let $w_{i_1} \ldots w_{i_m} \neq \varepsilon$. Then $m \neq 0$, since $w \neq \varepsilon$. We have a corresponding production rule $A \to A_{i_1} \ldots A_{i_n}$ in G', since the other variables are nullable. Thus, by induction, A can derive w.

3. **There are no derivation rules $A \to B$, where B is a variable**:
Call B unitarily derivable from A if there are a sequence of derivation rules

$$A \to V_1 \to \cdots \to V_n \to B$$

Define a new grammar $G' = (V, \Sigma, R', S)$. If B is directly derivable from A, and B has a derivation rule $B \to s$, then G' has a production rule $A \to s$, provided that s is not a variable. Then G' has no rules of the form $A \to B$, and generates the same language. This is fairly clear, and left to the reader to prove.

4. **Every rule is of the form $A \to AB$ or $A \to a$**:
If we have a rule in G of the form $A \to s$, where $s = s_1 \ldots s_n$, and $s_{i_1}, \ldots, s_{i_m} \in \Sigma$, then add new unique variables $A_{s_{i_k}}$ for each $s_i \in \Sigma$, and replace the rule with new rules of the form

$$A \to s_1 \ldots A_{s_{i_1}} \ldots A_{s_{i_m}} \ldots s_n$$

$$A_{s_{i_m}} \to s_{i_m}$$

Thus we may assume every rule of the form $A \to A_1 \ldots A_n$ (where we may assume $n \geqslant 2$) only maps to new variables. But then we may add new variables $V_2 \ldots V_{n-1}$, and swap this rule with rules of the form

$$A \to V_{n-1} A_n$$

$$V_k \to V_{k-1} A_k$$
$$V_2 \to A_1 A_2$$

Now every derivation rule is in the correct form, and we have reduced every grammar to Chomsky normal form. □

Chomsky normal form allows us to prove a CFG pumping lemma.

Theorem 5.3. *If L is a context free language, then there is $p > 0$, such that if $s \in L$, and $|s| \geqslant p$, then we may write $s = uvwxy$, where $|vwx| \leqslant p$, $v, x \neq \varepsilon$, and for all $i \geqslant 0$, $uv^i wx^i y \in L$.*

Proof. Let A be the language of the grammar G, which we assume to be in Chomsky normal form. Let there be v variables in G. If the parse tree of $s \in A$ has height k, then $|s| \leqslant 2^{k-1}$, which follows because the tree branches in two except at roots, so there is at most 2^{k-1} roots. If $|s| \geqslant 2^{3v}$, then every parse tree of s has height greater than v. Pick a particular parse tree of smallest size. There is a sequence of variables $A_1, A_2, \ldots, A_{3v}$, such that A_i is the parent of A_{i+1}. Because of how many variables there are, some variable must occur at least 3 times (for otherwise we may remove the variables in pairs, to conclude that $3v - 2v = v$ variables contain no variables, a contradiction). What's more, they much occur within a height of $3v$ of each other. Let $A_i = A_j = A_k$, for $i < j < k$. Let A_i produce aA_jb, let A_j produce xA_ky, and let A_k produce r. Write $s = maxrybn$. By virtue of the minimality of the tree, we may assume that a or b is nonempty, and one of x or y is nonempty. First, if both ax and yb are assumed non-empty, then we may pump these strings up, and have proved our lemma. So suppose ax is empty. Then b and y are nonempty, and $s = mrybn$. Since A_i produces A_jb, and A_j produces A_ky, A_i may be pumped to produce A_jb^i, A_j may be pumped to produce A_ky^i, and mry^ib^in is in the context free language, so we have non-empty strings to pump. The proof is simlar if by is empty, for then a and x are non-empty. The constraint $|vxy| \leqslant k$ is satisfies for yb and ax, for the A_i and A_j lie within $3v$ of each, so the string produces in this production is at most as long as 2^{3v}, which is less than or equal to the pumping length. □

5.2 Pushdown Automata

Regular languages have representations as the languages of regular expressions or as finite automata. Context-free languages also have dual representations, as 'machines' or as abstraction operations. It is good to represent a language as a machine for it may hint as to what hardware capabilities a computer must have to be able to solve problems related to the languages. The key machine component for a context-free language is a stack. A pushdown automata is a finite state automata with the addition of a stack.

Definition. A **(non-deterministic) pushdown automata** is a tuple $(Q, \Sigma, \Lambda, \Delta, q_0, F)$, where Q is a finite set of states, Σ is an alphabet, Λ is the stack alphabet, $\Delta : Q \times \Sigma_\varepsilon \times \Lambda_\varepsilon \to \mathcal{P}(Q \times \Lambda_\varepsilon)$ is the state transition function, $q_0 \in Q$ is the start state, and $F \subset Q$ are the accept states.

It turns out that deterministic pushdown automata are less powerful than non-deterministic automata, so we do not discuss deterministic automata. It is interesting to note that the languages of deterministic automata are connected to unambiguous grammars, though we will not have time to discuss this further.

Let us describe a pushdown automata intuitively. The automata has a stack of symbols from Λ, which it can push and pull from when deciding how to move through the machine. A stack is a string in Λ^*. Thus, formally, a string s is **accepted** by a push-down automata M if there are a sequence of states $q_0, \ldots, q_n$, and stacks $w_0, \ldots, w_n \in \Lambda^*$ such that $q_n \in F$, $w_0 = \varepsilon$, and if we write $w_i = w\lambda$, with $\lambda \in \Lambda_\varepsilon$, then $w_{i+1} = w_i\lambda'$, with $(q_{i+1}, \lambda') \in \Delta(q_i, \lambda)$. Thus we pop and pull off the rightmost character in the string when moving between states.

Pushdown automata have enough versatile memory to recognize context free languages. The stack can 'remember' variables it has yet to parse, and check when symbols are used. We shall allow a mild generalization of pushdown automata, which can push multiple symbols to the stack at a time. This is fine, without loss of generality, because we could have instead introduced new states that take nothing from the stack, and push the symbols on one at a time. We shall also assume a pushdown automata

starts with a \$ symbol at the bottom of its stack, which is fine, because we could have added another start state to the automata which pushes the \$ on as we begin running the machine.

Theorem 5.4. *Every context free language is accepted by a pushdown automata.*

Proof. Consider a context free language (V, Σ, R, S). Consider a pushdown automata $(Q, \Sigma, \Lambda, \Delta, q_0, F)$, with the stack language $\Lambda = \Sigma \cup V$, and Q just two states q_0 and f_0. For each derivation $A \to s \in R$ we have

$$(q_0, s) \in \Delta(q_0, \varepsilon, A)$$

And for each $a \in \Sigma$, we have

$$(q_0, \varepsilon) \in \Delta(q_0, a, a)$$

And a finale transition

$$(f_0, \varepsilon) \in \Delta(q_0, \varepsilon, \$)$$

It is clear that this automata parses the context free language. □

A converse also holds, so that pushdown automata are equivalent computers of context free languages. To do this, we assume that the automata pushes everything off its stack before it finishes, and has a single accept state f_0. In addition, we shall assume that a state only pops and pulls in one action, and doesn't do both at the same time. Adding additional states means this is no loss of generality.

Theorem 5.5. *Each pushdown automata language is context free.*

Proof. The gist of our approach is as follows. Let $(Q, \Sigma, \Gamma, \delta, q_0, \{f_0\})$ be a pushdown automata. We shall define a context grammar with variables A_{pq}, with $p, q \in Q$. This variable should be able to generate all possible strings which can start in p with an empty stack, and end up in q with an empty stack. Our start variable will then be A_{q_0, f_0}. The first rules are most basic

$$A_{pp} \to \varepsilon$$

Such a path may end up empty halfway through the path, so we have these rules, for each $p, q, r \in Q$,

$$A_{pq} = A_{pr} A_{rq}$$

We can also pop something on the stack, and save it for a long time later. If $t \in \Sigma$, and $(r,t) \in \Delta(p,a,\varepsilon)$, and $(q,\varepsilon) \in \Delta(s,b,t)$, then we add the derivation rule

$$A_{pq} = aA_{r,s}b$$

We claim these rules describe all possible derivations we could make in the pushdown automata. It is clear that all such derivations in this context language are accepted in the pushdown automata.

We shall prove that if we can move from p to q using a string x, both with an empty stack, then $A_{pq} \to x$. This is done by induction on the number of steps to accept the string in the automata. If we do this in one step, then the string is empty, and we have a rule $(q,\varepsilon) \in \Delta(p,\varepsilon)$, or the string consists of a single letter, and we have a rule $(q,\varepsilon) \in \Delta(p,t)$. In the first case, we have a derivation $A_{p,q} \to \varepsilon$, and in the second, we have a derivation

$$A_{p,q} \to tA_{q,q}\varepsilon \to t\varepsilon\varepsilon = t$$

Now consider a machine that runs for a length n. Suppose the stack empties at some state k, after running through x_1 of the string, for $x = x_1x_2$. Then we have, by induction, a derivation

$$A_{pq} \to A_{pk}A_{kq} \to x_1x_2$$

Thus we may assume that the stack never empties except at beginning and end. Then the first action must be to push a symbol t to the stack, and the last action to remove t. If we move from p to r in the first action by reading a, and from s to q in the last action by reading b, then we may write $x = acb$, and by induction, we have the derivation

$$A_{pq} \to aA_{rs}b \to acb = x$$

Thus we have verified the equivalence of the pushdown automata and context free language. □

Pushdown automata are easy to connect to their finite state cousins.

Corollary 5.6. *Every regular language is context free.*