Theory of Computation

Todd Gaugler

December 14, 2011

Contents

1 Mathematical Background **5**
1.1 Overview 5
1.2 Number System 5
1.3 Functions 6
1.4 Relations 6
1.5 Recursive Definitions 8
1.6 Mathematical Induction 9

2 Languages and Context-Free Grammars **11**
2.1 Languages 11
2.2 Counting the Rational Numbers 13
2.3 Grammars 14
2.4 Regular Grammar 15

3 Normal Forms and Finite Automata **17**
3.1 Review of Grammars 17
3.2 Normal Forms 18
3.3 Machines 20
3.3.1 An NFA λ 22

4 Regular Languages **23**
4.1 Computation 24
4.2 The Extended Transition Function 24
4.3 Algorithms 26
4.3.1 Removing Non-Determinism 26
4.3.2 State Minimization 26
4.3.3 Expression Graph 26
4.4 The Relationship between a Regular Grammar and the Finite Automaton 26
4.4.1 Building an NFA corresponding to a Regular Grammar 27
4.4.2 Closure 27
4.5 Review for the First Exam 28
4.6 The Pumping Lemma 28

5 Pushdown Automata and Context-Free Languages **31**
5.1 Pushdown Automata 31
5.2 Variations on the PDA Theme 34
5.3 Acceptance of Context-Free Languages 36

5.4 The Pumping Lemma for Context-Free Languages . . . 36
5.5 Closure Properties of Context- Free Languages . . . 37

6 Turing Machines 39
6.1 The Standard Turing Machine . . . 39
6.2 Turing Machines as Language Acceptors . . . 40
6.3 Alternative Acceptance Criteria . . . 41
6.4 Multitrack Machines . . . 42
6.5 Two-Way Tape Machines . . . 42
6.6 Multitape Machines . . . 42
6.7 Nondeterministic Turing Machines . . . 43
6.8 Turing Machines as Language Enumerators . . . 44

7 Turing Computable Functions 47
7.1 Computation of Functions . . . 47
7.2 Numeric Computation . . . 48
7.3 Sequential Operation of Turing Machines . . . 49
7.4 Composition of Functions . . . 50

8 The Chomsky Hierarchy 51
8.1 Unrestricted Grammars . . . 51
8.2 Context- Sensitive Grammars . . . 52
8.2.1 Linear-Bounded Automata . . . 52
8.2.2 Chomsky Hierarchy . . . 53

9 Decidability and Undecidability 55
9.1 Church-Turing Thesis . . . 55
9.2 Universal Machines . . . 55
9.3 The Halting Problem for Turing Machines . . . 56
9.4 Problem Reduction and Undecidability . . . 56
9.5 Rice's Theorem . . . 56

10 μ-Recursive Functions 57

11 Time Complexity 59

12 $\mathcal{P}, \mathcal{NP}$, and Cook's Theorem 61

Chapter 1

Mathematical Background

1.1 Overview

The Chaimsky Hierarchy of Languages (on page 339) is broken up into the following types:

Type	Language	Grammar	Machine
0	Recursively Enumerable	Unrestricted Phrase Structure	Turing Machine
1	Contex-Sensitive	Context-Sensitive	Linear Bounded
2	Context-Free	Context-Free	Pusmh-Down
3	Regular	Regular	Finite Autmomaton.

1.2 Number System

The natural numbers, denoted $\mathbb{N} = \{0, 1, 2, ...\}$ will be important in this class, along with the integers $\mathbb{Z}$, the rational numbers $\mathbb{Q}$, and the real numbers $\mathbb{R}$. The irrational numbers, $\overline{\mathbb{Q}}$ or $\mathbb{R} - \mathbb{Q}$.

Definition

A **set** is a collection of elements where no element appears more than once.

Some common examples of sets are the empty set, $\{\}$ and singletons, which are sets that contain exactly one element. The notion of a subset of some set $S = \{s_1, s_2, ...s_n\}$ is a new set $R = \{r_1, r_2, ...r_n\}$, and $R \subseteq S$ if for some j, $r_i = s_j$. A proper subset, denoted $R \subsetneq S$ is a subset that does not contain all the elements of S. The notion of a **power set**, denoted $\mathcal{P}(S)$ is the set of all subsets of S. Given some set S with n elements, the cardinality (or number of elements in that set) of $\mathcal{P}(S)$ is 2^n.

Given two sets A, B we can talk about their union $A \cup B = \{c, | c \in A \text{ or} c \in B\}$. Similarly, intersection can be described as: $A \cup B = \{c | c \in A \text{ and } c \in B\}$. Union and intersection are commutative. When taking the difference, $A - B$, we see that this operation is not commutative. These operations are known as binary operations.

Another important operation is the notion of the unary operation **complementation**. The complement of a set S can be written S', or S^c which considers the universe $\mathcal{U}$ in which S is contained, and $S' = \mathcal{U} - S$.
Example. Take the odd positive natural numbers. The compliment of the odd numbers depends on the universe in which you are placing these odd positive natural numbers. If we allowed the universe to be the integers $\mathbb{Z}$, the compliment of this set would contain all odd and negative even integers.

Definition

Demorgans law: $\overline{A \cup B} = \overline{A} \cap \overline{B}$ and $\overline{A \cap B} = \overline{A} \cup \overline{B}$

1.3 Functions

A function $f : X \to Y$ is unary, whereas the function $f(x_1, x_2)$ is called a binary function. This notion can be extended to the n case. A **total function** is a function which is defined for the entire domain. Many of the functions in this course will be of the form: $f : \mathbb{N} \to \mathbb{N}$. However, looking at the function $f(n) = n - 1$, we see that this can't possibly be defined for the entire domain of $\mathbb{N}$, since if you plug 0 into this function, the image under f of 0 is not contained in $\mathbb{N}$. This such function is known as a **partial** function. However, we could redefine f such that $f(n) = n - 1$ for all $n \geq 1$, and if $n = 0$, $f(n) = 0$. This then becomes a total function.

Two other important properties of function are onto functions and one-to-one functions. If a function is $1 - 1$, this means that $f(a) = f(b) \Rightarrow a = b$. A function f is onto if its image is all of the set to which it is mapping. A function is **bijective** if it is both onto and one-to-one.

1.4 Relations

You can define $x \in \mathbb{N}$ as being in a relationship with y in the following way: $[x, y]$. Unlike a function, an element x can be in relation with many other "outputs". These relationships are subsets of the Cartesian product $\mathbb{N} \times \mathbb{N}$.

Example. Let $A = \{2, 3\}$ and let $B = \{3, 4, 5\}$. The Cartesian product is defined as follows:

$$A \times B = \{(a, b) | a \in A, b \in B\}$$

The cardinality $|A \times B| = |A| \times |B|$.

Definition

Cardinality denotes the number of elements in some given set. A set has **finite** cardinality if the number of elements it contains is finite. A set has countably infinite cardinality if there exists a bijection from that set to the natural numbers. A set has **uncountable** cardinality if it contains an infinite number elements, but there does not exist a bijection with that set and $\mathbb{N}$.

Suppose we think of $[0, 1) \subsetneq \mathbb{R}$. Any such number must be a decimal number, less than 1, that has some sequence of numbers that corresponds to binary numbers. We will show that this interval has uncountable Cardinality through a proof by contradiction.

Proof. You can set up each number in this interval as:

$$\begin{aligned} r_0 &= 0.b_{a_1} b_{a_1} b_{a_3} \ldots. \\ r_1 &= 0.b_{b_1} b_{b_1} b_{b_3} \ldots. \\ r_2 &= 0.b_{c_1} b_{c_1} b_{c_3} \ldots. \\ r_3 &= 0.b_{d_1} b_{d_1} b_{d_3} \ldots. \end{aligned}$$

And then you represent the new number:

$$r_{new} = b^c_{a_1} b^c_{b_2} b^c_{c_3} \ldots$$

Which can be shown to not have been in our original list. [1] □

It can also be shown that total unary functions $f : \mathbb{N} \to \mathbb{N}$ are also of infinite cardinality thought a similar argument using the following table:

	1	2	3	4	5 ...
$f_0(n)$	$f_0(0)$	$f_0(1)$	$f_0(2)$	$f_0(3)$	$f_0(4)$
$f_1(n)$	$f_1(0)$	$f_1(1)$	$f_1(2)$	$f_1(3)$	$f_1(4)$
$f_2(n)$	$f_2(0)$	$f_2(1)$	$f_2(2)$	$f_2(3)$	$f_2(4)$

$$\vdots$$

[1] The complement of a binary number will simply be the value of that number +1

It can also be shown that $\mathcal{P}(\mathbb{N})$ is of uncountable cardinality. The proof of this is in our book.

Another important notion for this class is the notion of a **partition**. A set S can be partitioned into subsets $A_1, A_2, ...A_n$ such that $S = A_1 \cup A_2 \cup A_3 ... \cup A_n$ and that $A_i \cap A_j = \emptyset$ if $i \neq j$. The last property states that A_i, A_j are **disjoint** if $i \neq j$. This brings us back to the idea of an equivalence relation.

Definition

An equivalence relation is a relation (I'll say that a is related to b by writing $a \star b$) that satisfies the following:

1. $a \star a$
2. $a \star b \Rightarrow b \star a$
3. $a \star b, b \star c \Rightarrow a \star c$.

Some examples of an equivalence relation include:

1. Connectedness in a graph
2. Equivalence of vectors
3. Isomorphisms of groups, rings, etc.

The Barber's Paradox: "There is a barber that shaves everyone that cannot shave themselves. So, does he shave himself?"

1.5 Recursive Definitions

For a recursive definition, we need the following:

1. A basis
2. A recursive step
3. A closure- if we can reach a result by starting with the basis and a finite number of applications of the recursive step.

Example. Suppose you have $m + n$, the addition of two numbers. So first, we define $m + 0 = m$. Now, we define our recursive definition as $m + S(n) = S(m + n)$. Think of it in this way:

$$\begin{aligned} m + 0 &= m \\ m + 1 &= S(m + 0) \\ m + 2 &= S(m + 1) \\ &\vdots \end{aligned}$$

Which effectively defines addition, where $S(n)$ is the "successor" of n.
Example. Using the equivalence relation "less than", we can first say that $[0,1]$, and that if $[m,n]$, then $[m,S(n)]$. Similarly, $[S(m),S(n)]$ is in the relation. This is an example of another recursive definition.

1.6 Mathematical Induction

Looking at the sum $1+2+3+4....+n$ we can prove inductively that this sum equals $\frac{(n)(n+1)}{2}$.

Proof. Our base case when $n=0$ holds true for this formula, as does when $n=1$. To proceed, we now look at the inductive hypothesis, which says we will assume that indeed this formula works, and that we now need to prove that this works for the $n+1^{th}$ case. To show this, add 1 to n, plug it into the formula:

$$\frac{(n+1)(n+2)}{2} = \frac{(n^2+3n+2)}{2} = \frac{n(n+1)+2n+2}{2} = \frac{(n)(n+1)}{2} + (n+1)$$

which tells us that the $n+1^{th}$ case holds. □

We can show using strong induction that given a tree, $|v| = |E| + 2$.

Proof. Given some tree, you can break it up into two separate pieces by separating it into two pieces via one edge. Then by the strong inductive hypothesis,

$$|v_1| = |E_1| + 1, \quad |v_2| = |E_2| + 1$$

and we know that

$$|V| = |v_1| + |v_2| = (|E_1| + 1) + (|E_2| + 1) + 1 = |E| + 1$$

Where $|E| + 1 = |E_1| + |E_2| + 1$ □

Chapter 2

Languages and Context-Free Grammars

2.1 Languages

Definition

There exists a set that we call the **Alphabet** $\Sigma = \{a, b, c\}$ and, λ =denotes "the empty string".

And we have the following identity for the element λ:

$$a = \lambda a = a\lambda$$

And there exists the notion of an operation called **concatenation**, which is the notion we all learned in $CS111$. This operation is **NOT** commutative. In other words, $ab \neq ba$. However, concatenation is associative- notice that the following are equivalent:

$$a(bc) = (ab)c$$

Where the '(' indicate the operation of concatenation. We can also talk about things like **substrings** and **prefixes**, which are fairly self-evident. Another operator to think about is the **reversal** operator, which does the following:

$$a^R = a, \quad word^R = drow, \quad (uv)^R = v^R u^R$$

The last equality can be shown by mathematical induction:

Proof.

the base case, $|v| = 0(v = \lambda)$
$(u\lambda)^R = u^R = \lambda u^R = \lambda^R u^R$
now the case in which $|v| = 1$:
$(ua)^R = a(u^R) = a^R u^R$
using the inductive hypothesis, we assume $(uv)^R = v^R u^R$ for $|v| = n$
we need to prove that $(uw)^R = w^R u^R$ for $|w| = n + 1$, which we can represent as:
$(uva)^R = a(uv)^R = av^R u^R = a^R u^R v^R$
and this is the same as:
$(va)^R u^R = w^R u^R$ and $|va| = |w| = n + 1$

□

Definition

If we write a^*, this denotes the set that includes $\{\lambda, a, a^n\}$. This operation is called the **Kleene Star**.
Example. Given a word w,

$$w^* = \{\lambda, w, ww, www, ...\}$$

We also talk about the **union** in the context of words. We can say:

$$u \cup v = \text{"either the word 'u', or the word 'v' "}$$

Using union, we can also speak about examples like $u \cup v \cup w$... We can put these operations in the context of a **Regular Expression**, which includes $\lambda, a \in \Sigma,$ concatenation, Kleene Star (*), and Union $\cup$. However, a regular expression does not have a concept of "intersection", or "and". We will allow a "+" super script, where w^+ denotes "one more copy of w". This is allowed in a regular expression because it is just a combination of two operations already existent in a regular expression; $a^+ = aa^*$.

Definition

A **Language** L is the set of words over some alphabet Σ. A language can be empty, can be infinite, and can have one word.

Example. one example of an infinite language can be the constructed by allows $\Sigma = \{a, b, c\}$. Our language L then consists of all 1, 2, 3, 4.... letter words, and this generates an infinite number of words even though the length of the words has to remain finite. This is denoted Σ^*.

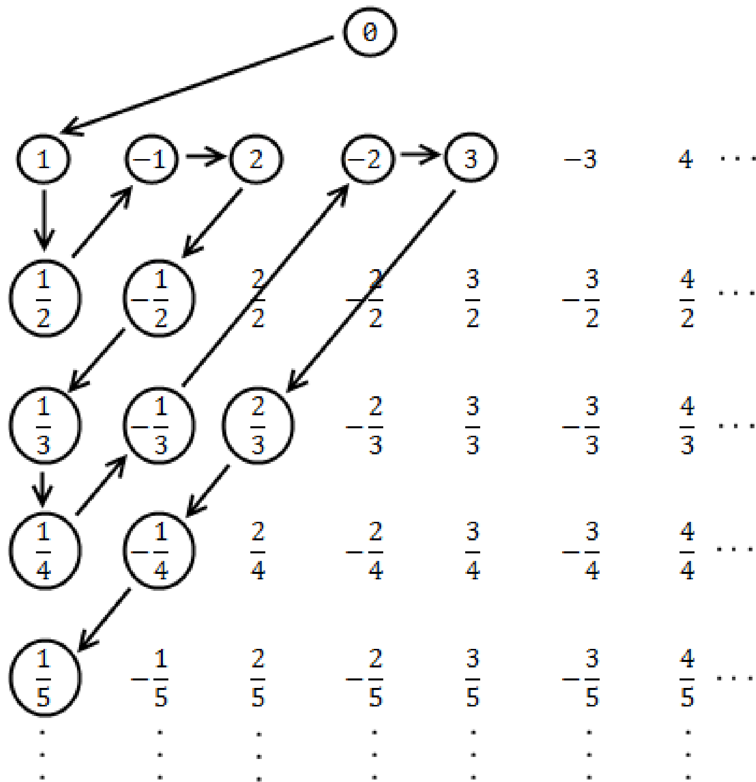
We can also talk about the set of all languages of Σ (this is a good multiple choice question).
Example. The set of all languages over some alphabet Σ is like the power set $\mathcal{P}$ of our alphabet Σ^*, and is denoted $\mathcal{P}(\Sigma^*)$. Now while the cardinality of the $|P(S)| = 2^{|S|}$ when $|S|$ is finite, noting that Σ^* is infinite, this won't help us here. However, similar to the argument that the power set of the natural numbers was uncountable, we can use a similar diagonalization argument to show that $|P(S)|$ is uncountable.

To review, the following are used to denote certain operations:

$L_1 \cup L_2$	union
$L_1 L_2$	concatenation
L^*	Kleene Star
λ	$\emptyset$

2.2 Counting the Rational Numbers

We can use the following diagonalization 'snake' to find a correspondence between $\mathbb{N}$ and $\mathbb{Q}$:



A different picture was given in class, but they are equivalent. However, the irrational numbers

$\mathbb{R} - \mathbb{Q}$ are **not** countable, and no such diagonalization argument can be shown for them.

2.3 Grammars

Think about the way people talk- there exists some structure in which we all communicate. In English, we use $< nouns >$ and $< verbs >$ and have rules to determine what is a 'noun' or a 'verb'. We are using the rules of **grammar** and applying them. This is very similar to the way in which Compilers view Code.

Definition

A **Grammar** has four components:

1. Variables
2. an alphabet
3. production rules
4. start symbols

This can be written as (V, Σ, P, S). **Variables** are usual capital letters, i.e. $V = \{A, B, C, ..\}$.These are also known as **non-terminal** letters, as opposed to the alphabet, which are terminal symbols. Using the special symbol $S \in V$, we apply a sequence of production rules to S, and this will eventually produce a word, $\in \Sigma^*$. A **production rule** for a context-free grammar is the following: $V \to (V \cup \Sigma)^*$, where you take a Variable then change them to a sequence of variables and elements in the alphabet.
Example.

$$A \to aA, \quad B \to bCaB \quad C \to \lambda \quad D \to b$$

Let's consider the case in which we have $\Sigma = \{a, b\}$, S, and $P : \{S \to aSa, \quad S \to bSb, \quad S \to \lambda\}$, $V = \{S\}$. And since $S \to aSa, \to aa$ We know that $S \overset{*}{\Rightarrow} aa$. Notice that this generates palindromes over $\{a, b\}$, where these palindromes are of even length. For example,

$$S \to aSa \to aaSaa \to aaaSaaa \to .... \to \underbrace{aaaaaa...}_{evenlength}$$

This can be remedied by saying $P : S \to \{aSa, bSb, \lambda, a, b, \}$, of you could allow P to map S to some other variables that map strictly to either a or b. From now on, we will use a vertical bar $|$ to denote 'or' in the map P.

Now suppose you want to generate words of the form $L = \{a^m b^n | n, n \geq 0\}$. How would you do this?

$$P : S \to \{AB|\lambda\}, \quad P : A \to \{aA|\lambda\}, \quad P : B \to \{bB|\lambda\}$$

this application is called **the recursive rule**. The act that sends a variable to the null string is called the **null rule**. The variables of this grammar would be $V = \{S, A, B\}$, which correspond to our production rules.

Similarly, if we wanted the following : $L = \{a^m b^n | m = n \geq 0\}$. This can be done through the following production rules:

$$P : S \rightarrow \{aSb|\lambda\}$$

2.4 Regular Grammar

Definition

A **Regular Grammar** has:

1. $A \rightarrow \lambda$
2. $A \rightarrow a$
3. $A \rightarrow aB$

This is the generic form for a Regular grammar, which would be written as:

$$V \rightarrow \lambda \cup \Sigma \cup \Sigma V$$

A **context-free grammar** is written generally as:

$$V \rightarrow (V \cup \Sigma)^*$$

A language is regular if the words can be represented by a regular expression or is generated by a by a Regular Grammar or is accepted by a Finite Automaton (DFA, NFA, NFA-λ)

Chapter 3

Normal Forms and Finite Automata

The following are the notes that we took on the 13th of September.

3.1 Review of Grammars

Recall that a grammar is defined as follows:

$$G = (V, E, P, \Sigma)$$

Where:

1. V denotes the variables
2. Σ represents the alphabet
3. P represents the production
4. S represents the designated 'start symbol'

If we have a **Context-Free Grammar**, we have the general form:

$$V \to (V \cup \Sigma)^*$$

Where a Regular grammar looks like the following:

$$V \to \lambda \cup \Sigma \cup \Sigma V$$

This has to do with what i called the **verification of grammars**. We can talk about the language of the grammar, $L(G)$. Take the following example.
Example. $P : S \to aSa|bSb|\lambda$ could be our production rule, which means our language would look like even length palindromes over $\{a, b\}$. More formally, we are claiming that $L(G) =$ even length palindromes over $\{a, b\}$. To prove as such, we need to show that everything produced under this rule gives us such a word. This is fairly easy to see. Another way to do this would be to show that every such palindrome would be generated through this production.
Remark. A sentence is in sentential form when it is still on its way to being fully produced; it has variables left in it.

Definition

The notion of **ambiguity** is related to the idea that it is more than possible to derive the same words via different sequences based on your production rules. This means that equivalent words would be found at different parts of a 'tree' that relates the choice of production rule used to generate a word.
Example. using the following production rules:

$$S \to aS|Sa|\lambda, \quad \text{there are many ways to generate the word 'aaaa'}$$

Sometimes we try to eliminate ambiguity, and this is through a **left-most derivation**.
Example.

$$S \overset{*}{\Rightarrow} ABbBa^1$$

at this point, we can look for rules as to where to map the variable A, and can apply one of those rules to the A. Another idea is to look for rules as to where we can map B; either way we need to make that choice. However, in a left-most derivation, we always use rules for the **left-most** variable first. The proves that if a word can be derived using a grammar, there exists a left-most derivation. And much more obviously, a left-most derivation implies that there is a derivation. Notice that if there exists a derivation, there exists some sequence of steps where

$$S \Rightarrow ... \Rightarrow w$$

If this was **NOT** true, then at some point, we had to use a variable that was not the left-most variable. Let's call this

$$S \Rightarrow ... \Rightarrow u_1Au_2Bu_3 \Rightarrow ... \Rightarrow w$$

in other words, to get to w, we had to break the rule at that middle point there, where B had to be processed befor we got to A. However, this makes no sense, since we would still have to process the A at some point to get that w - ignoring it temporarily does not solve our problem. While this isn't a solid proof, it gives us the intuitive idea for something more vigorous.

Example. 'Jack was given a book by Hemingway' means that either Jack was given a book written by Hemingway, or that the book was handed to him by Hemingway, or that Hemingway handed him a book written by himself. This is an example of the ambiguity of grammar.

3.2 Normal Forms

The first notion of a normal form is of a **Grammar Transformation**. Given some grammar $G = (V, \Sigma, P, S)$ and a second grammar $G' = (V_2, \Sigma_2, P_2, S_2)$ we can ask if they are equivalent, or if they generate the same set of words.

[1] where the star denotes an unspecified number of steps

We need two things to establish $L(G) = L(G')$:

1. Every word in G can be generated by G'
2. Every word in G' can be generated by G

Noticing immediately that the alphabets are the backbone of the words generated, it seems natural to claim that the languages must have the same alphabets. However, this is not true- on occasion, some languages do not display certain letters (perhaps some languages do not have production rules that allow certain letters to appear). For example, $L = \{\}$ is one such example. However, it is true that when letters appear in one language, they must also appear in the other. Grammar transformation is the process under which we take a grammar and modify it by adding/changing variables and production rules to transform it so that we get a grammar in a form we like, and retain the same language.

The first undesirable property of a grammar are called $\lambda-$rules. If λ is included in the language, then we **must** have some sort of λ rule to get λ. Thus, we need the rule $S \to \lambda$. This does not mean that we can also have $A \to \lambda$, since S is our start- symbol. How can we remove λ-rules? What you can do is replace all variables $B \to \lambda$, you can work backwards and remove all λs. For example, suppose we had

$$S \to aA|aBa|b|\lambda \qquad B \to \lambda|bB \qquad A \to Aa|\lambda$$

we could switch this to the following:

$$S \to aA|aB|b|\lambda|a \qquad B \to bB|b \qquad A \to Aa|a$$

this is one method to remove λ-rules. Another undesirable property of grammars is called **a chain rule**. This looks like the following:

$$A \to B \qquad B \to C$$

we end up combining these two to the following:

$$A \to C \quad B \to C, \quad A \to B$$

we also have things called **useless symbols**

Definition

The **Chaomsky Normal Form** is an example of a normal form where all his rules obey the following forms:

$$S \to \lambda \text{ if there exists a } \lambda \text{ rule}$$

we could have

$$A \to a$$

and we could have

$$A \to BC$$

the only catch is that $B, C \in \{V - \{S\}\}$, which says that these variables should not be S. This is because we want a non-recursive start symbol. One way of getting rid of a start symbol S would be to define a new start symbol

$$S' \to S$$

and that way, anything else that would be illegal for the start symbol would be legal, since it isn't a start symbol anymore.

We will show that given something in Chomsky normal form and using the CYK algorithm, we can generate a machine.

There exists another type of normal form, called the **Greibach Normal Form**, which includes the following:

$$S \to \lambda, \qquad A \to a, \qquad A \to aA_1A_2A_3...A_n$$

one last undesirable feature is the eliminating of **direct left recursion**, which gives us something along the lines of

$$A \to B \text{ [more variables]}$$

we would like to move forward to our final product, not to more variables. Notice that Greibach normal form avoids this by requiring that when adding more variables, at least one letter must be included.

3.3 Machines

The machines we will talk about are called **finite state machines**. They are not physical, they are abstract processors of strings that make decisions based on the strings they take as inputs. A **state** denotes that there are situations under which the machine can process strings. In giving an analogy, we can think of a newspaper machine. Suppose a newspaper costs \$.60, and the machine accepts nickels dimes and quarters. The input can then be thought of as a sequence of characters representing money values. Suppose a man comes and inputs *dqdd*, which is only worth \$.55. Naturally, the machine won't accept this string. However, if he put in the string *qnqn*, the machine would then accept the money and give the man a newspaper. This is illustrated by the following picture:

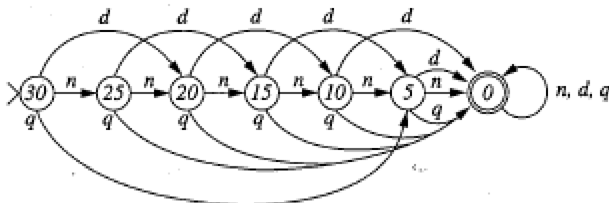


FIGURE 5.1 State diagram of newspaper vending machine.

Where the machine has some 'start state', in which it expects $.60, and an accepting, or final state, where it has recieved the money it needs and need nothing else.

The finite state machine we'll talk about are deterministic finite automatons (DEA)s, and are defined as follows:

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

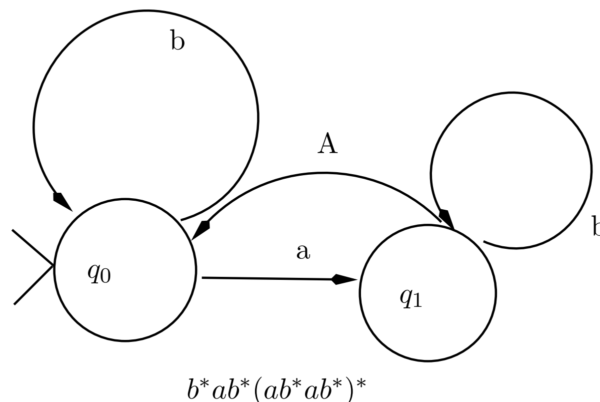
1. Q stands for the set of states.
2. Σ is the alphabet.
3. δ is the transition function
4. q_0 is the start state, which $\in Q$.
5. F is the set of final state(s), $F \subseteq Q$.

We know that $L(M) : \emptyset$ and $L(M) : \Sigma^*$. A transition function does the following:

$$\delta : Q \times \Sigma \to Q$$

this function is a total function. It takes in some input, looks at some input and the current state, and determines which state to go to next.

the following is a regular expression for the strings accepted by a machine:



$$b^*ab^*(ab^*ab^*)^*$$

Definition

Whereas a deterministic finite automaton does not permit functions to map to multiple images, A **non-deterministic finite automaton** (NDF or NFA) is the same as a deterministic finite automaton except that it has a different (but still total) transition function:

$$\delta : Q \times \Sigma \to \mathcal{P}(Q)$$

for example,

$$\delta(q_1, a) = \{q_2, q_3\}$$

notice that the output is a single set that contains multiple states. A set of states can be one of three things:

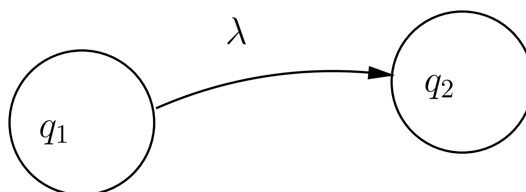
1. Empty $\emptyset$
2. A singleton $\{q_i\}$
3. Multiple elements $\{q_1, q_2, ...q_n\}$

So, a valid function for a non-deterministic finite automaton could be the following

$$\delta(q_1, b) = \{\}$$

3.3.1 An NFA λ

This implies that there is a λ transition from one state to the next where λ is the empty string.



To accommodate this, we allow the following definition:

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \rightarrow \mathcal{P}(Q)$$

Chapter 4

Regular Languages

Recall that a DFA is a machine denoted by $M = (Q, \Sigma, \delta, q_0, F)$, where the transition function looks like the following:

$$\delta : Q \times \Sigma \to Q$$

Also recall that an NFA is a machine denoted by $M = Q, \Sigma, \delta, q_0, F)$ where the transition function looks like:

$$\delta : Q \times \Sigma \to \mathcal{P}(Q)$$

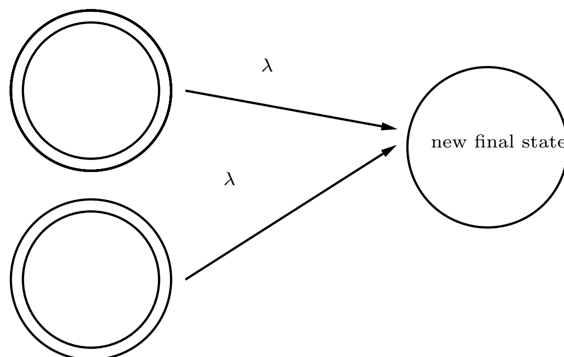
and in an NFA-λ, our transition function looks like:

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \to \mathcal{P}(Q)$$

we have a lemma that says:
Lemma 1. *A finite state machine can be made into an equivalent machine of the following:*

1. *The start state is of degree zero*
2. *The final state has out degree zero*
3. *The machine has only one final state*

Proof. It is fairly easy to see that through the use of a λ-transition one can take a machine's final state, map it to a new final state that satisfies these conditions, do the same with the start state, and we are done. This new machine mimics the structure of a regular language. ☐



4.1 Computation

Computation is the process of starting a machine, doing computations, and determining whether or not you are at a final start or not when finished. The process looks like the following:

$$[Q_i, aw] \mapsto_M [q_j, w]$$

where the 'w' stands for the 'word', and this map has processed the 'a'. Sometimes these blocks are called a **machine configurations.** A computation is the process of going from one machine configuration to the next. Just as we use a '*' over the $\Rightarrow$ to indicate that something can be derived in some amount of steps, we use the same symbol between machine configurations. Notice that this is only possible if

$$\delta(q_i, a) = q_i$$

4.2 The Extended Transition Function

For our purposes, an extended function is for a DFA. The difference between a extended transition function and a transition function is that a transition function acts on one character. Meanwhile, the extended transition function:

$$\hat{\delta}(q_i, w) = q_i, \qquad \hat{\delta} : Q \times \Sigma^* \to Q$$

takes entire words and maps them to states. Defining the extended transition function through a recursive definition, we use the following basis:

$$\hat{\delta}(q_i, \lambda) = q_i$$

and the basis

$$\hat{\delta}(q_i, a) = \delta(q_i, a)$$

now inductively, we can say

$$\hat{\delta}(q_i, wa) = \delta(\hat{\delta}(q_1, w), a), \qquad |w| = n, |wa| = n + 1$$

inductively, it can be shown that $\hat{\delta}(q_0, w) = q_i$ yields a unique path from q_0 to q_i.
Example. One of the languages from the homework was $L = \{a^n b^n | n > 0\}$. Looking at the language $L = \{a^n b^n | 0 < n \leq m\}$, we can define this under the production rule:

$$S \to ab|a^2b^2|...|a^m b^m$$

Looking at that language $L = \{aabb\}$ defined as a regular grammar as:

$$S \to aB \quad B \to b|aC \quad C \to bD \quad D \to b$$

we know there has to be a machine that mimics this structure. Such a machine can be seen in figure (4.1).

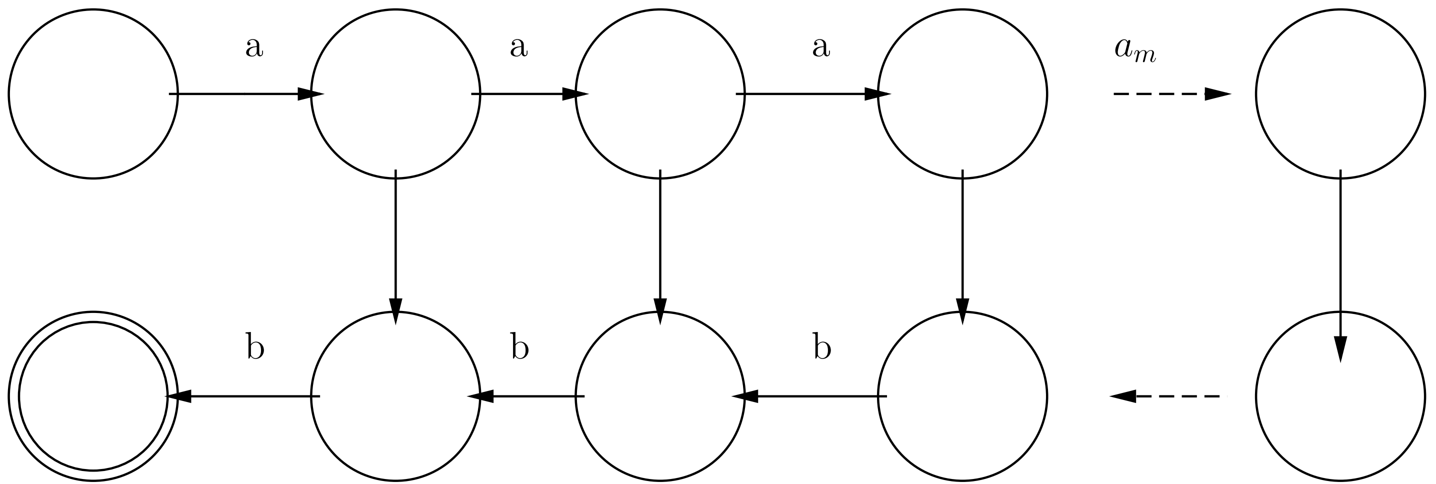


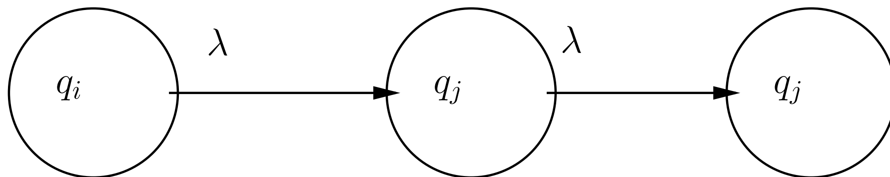
Figure 4.1: $L = \{a^n b^n | 0 < n \leq m\}$

Definition

This is 5.6 from the book: The λ-closure is 'all the places you can go when processing the empty string'. It is written as the following:

$$\lambda - closure(q_i)$$

and through a recursive definition, $q_i \in \lambda - closure(q_i)$, $q_j \in \lambda - closure(q_i)$ if $q_* \in \lambda - closure(q_i$ and $q_j \in \delta(q_*, \lambda)$



The **input transition function** looks like the following:

$$t(q_i, a) = \bigcup_{q_j \in \lambda - closure(q_i)} \lambda - closure\,(\delta((q_j, a))$$

4.3 Algorithms

4.3.1 Removing Non-Determinism

Recall that we have DFA, NFA, NFA-λ. Now what is true, is that any NFA covers anything a DFA could do-but the less obvious discussion is that the reverse is also true.

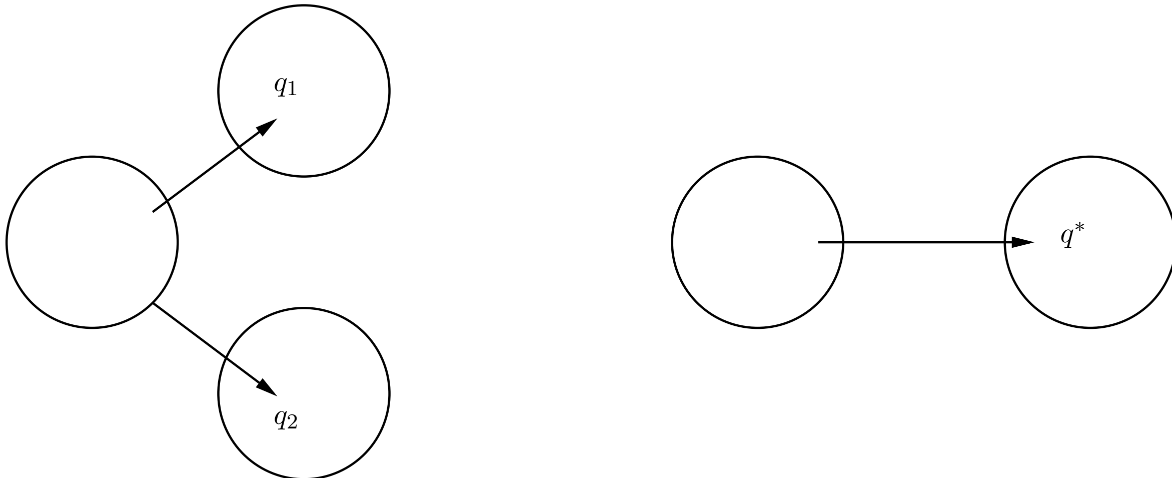


Figure 4.2: The idea is to construct a new state that represents both of these states, and remove the indeterminism.

4.3.2 State Minimization

The idea is to remove states that are 'redundant', which we call **indistinguishable** states. The idea is if states represent the same sequence of characters, they serve the same person, and can be minimized.

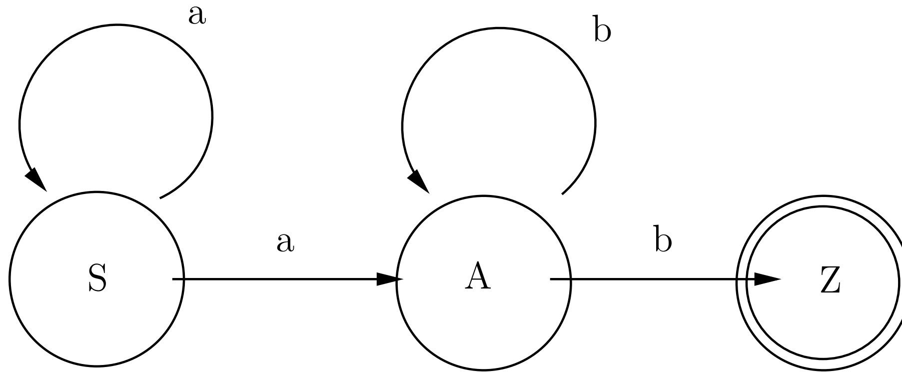
4.3.3 Expression Graph

The idea of an expression graph is to say that it might be easier to deal with regular expressions as opposed to long sequences of characters.

4.4 The Relationship between a Regular Grammar and the Finite Automaton

If we have a regular grammar, how do we build a machine that accepts that grammar, and vice-versa. Given the language:

$$S \to aS|aA \qquad A \to bA|b$$



a regular expression for this language can be written as a^+b^+.

$$Q = \{V \cup \{Z\} | where \;\; Z \notin V \text{ in Production Rules}\}\{V\text{otherwise}\}$$

where the transition function will be:

$$\delta(A, a) = B, \quad A \to aB \in P \qquad \delta(A, a) = Z \to A \to a \in P$$

and the final state will be:

$$F = \left| \begin{array}{c} A|A \to \lambda \in P \cup \{Z\} \text{ if } Z \in Q \\ A|A \to \lambda \in \text{ if } Z \notin Q \end{array} \right.$$

this creates and arbitrary machine for an arbitrary regular grammar. Now, we would like to go from a machine to a regular grammar.

4.4.1 Building an NFA corresponding to a Regular Grammar

Starting with a regular grammar corresponding to an NFA, we need:

$$\begin{aligned} & V = Q \\ & \Sigma \\ & P = q_i \to aq_j \in P \text{ if } \delta(q_i, a) = q_j \\ & \qquad q_i \to \lambda \in P \text{ if } q_i \in F \\ & S \end{aligned}$$

4.4.2 Closure

Will the output of a binary operation acting on two members of a set be in that set? Looking at languages, the compliment of a language L_1 is:

$$\overline{L}_1 = \Sigma^* - L_1$$

to do this on a machine, you change the final state of your machine from F to $Q - F$. Now, everything that would have been rejected will be accepted, and vice-versa.

4.5 Review for the First Exam

Just as a trick, notice that a DFA is also a NFA and an $NFA - \lambda$, they are all equivalent.

We have said that $L = \{a^n b^n | n \geq 0\}$ is non regular. Now suppose that we bound n above by n- is this regular? It turns out that it is- having a finite selection gives us a regular language. We have already seen what this looks like in terms of a Machine.

4.6 The Pumping Lemma

Let L be a language that is accepted by a DFA with K states. Let z be any string in L with

$$length(z) \geq k$$

Then, z can be written as uvw with length

$$length(uv) \leq k, \quad length(v)$$

and $uv^i w \in L$ for all $i \geq 0$. It can be used to prove that languages are not regular. However, it **cannot** be used to prove that a language is regular.
Example. Let $L = \{a^i | i \text{ is prime}\}$. We have K states, and consider some word a^n that will be broken down into uvw, as our lemma says, then can be 'pumped'. We'll do the following:

$$uv^{i+1}w$$

and consider the length of this word, which gives us:

$$length(uv^{i+1}w) = |u|+|v^{i+1}|+|w| = |u|+(n+1)|v|+|w| = |u|+|v|+n|v|+|w| = n+n|v| = n(1+|v|)$$

now what this means, as that the length of this string is n the length of that word $v + 1$. We know that this is a composite number, since it is the product of two numbers (neither of which is 1, since $|v| > 0$). Thus, this language is not regular.
Example. Let $L = \{a^m |$ m is a perfect square$\}$ Notice that the length of a^{k^2} is clearly k^2, which is a perfect square. In this case the number of states will be K. According to the pumping lemma,

$$a^K = uvw \quad |v| > 0$$

as before, we consider

$$|uv^2w| = |uvw| + |v| = K^2 + |v| \leq K^2 + K < K^2 + K + K + 1 = (K+1)^2$$

but at the same time, we know:

$$K^2 < |uv^2w| < (K+1)^2$$

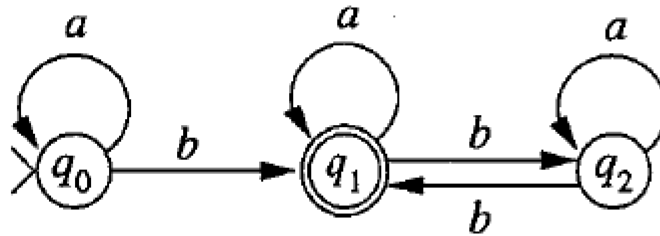
thus, $|uvw|$ can't be a perfect square. This means that our assumption is incorrect- that our language is regular. Thus, this language is not regular.

The Pumping lemma is necessary for a language to be regular, but is not sufficient. The following is section 6.7:

Theorem 2. ***The Myhill-Necode Theorem:*** *Given $u, v \in \Sigma^*$, where $uw \in L$ whenever $vw \in L$, and $uw \notin L$ whenever $vw \notin L$. We then call u and v* ***indistinguishable****- they basically behave in the same way. They behave nicely with reflexive, symmetric, and transitive properties.*
Example. Consider $\{a^ib^i|i \geq 0\}$ and the word a^ib^j. Now looking at $a^ib^ia^ib^j$, this word does not behave as something inside of the language-but, we are saying that a^i and a^j are not indistinguishable- since depending on what is put after them, sometimes you will have something in the language and sometimes you won't. I.e.,

$$a^ib^i \in \text{L} \qquad a^jb^i \notin L$$

thus a^i and a^j are distinguishable and are in separate equivalence classes.
Example. (page 212) the regular expression for this is: $a^*ba^*(ba^*b)^*$



Notice that one equivelance class is going to be a^*, from which we can see at the beginning. A second will be $a^*ba^*(ba^*ba^*)^*$, and the third will be $a^*ba^*ba^*(ba^*ba^*)^*$. This is an example of a finite number of equivalence classes.

Chapter 5

Pushdown Automata and Context-Free Languages

Recall that we decided that regular languages are generated by regular grammars, and accepted by finite automata. However, also recall that there were certain limitations to what a finite automata could accept: For example, the language $\{a^i b^i \ |i \geq 0\}$ could not have been accepted by a deterministic finite automaton because it had no ability to "remember" what amount of elements had already been accepted. This is our motivation for things called **Pushdown Automata**

5.1 Pushdown Automata

For a machine to accept the language $\{a^i b^i | i \geq 0\}$, it needs to have the ability to record the processing of any finite number of $a's$. The restriction of having finitely many states does not allow the automatons we discussed previously to do such a thing. Thus, we define a new type of automaton that augments the state-input transitions of a finite automaton with the ability to utilize unlimited memory.

A **pushdown stack**, or simply a stack, is added to a finie automaton to construct a new machine known as a pushdown automaton (PDA). stack operations affect only the top item of the stack: a **pop** removes the top element from the stack, and a **push** places and element on the top of the stack. Formally, we have the following definition:

Definition

A **pushdown automaton** is a sextuple:

$$(Q, \Sigma, \Gamma, \delta, q_0, F)$$

Where:

1. Q is a finite set of states

2. Σ is a finite set called the **input alphabet**
3. Γ is a finite set called the **stack alphabet**
4. q_0 is the start state
5. $F \subseteq Q$ is a set of final states,
6. δ is a transition function of the form:

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{\lambda\}) \to Q \times (\Gamma \cup \{\lambda\})$$

Notice that this tells us that a PDA has two alphabets, one from which the input strings are built and a stack alphabet Γ whose elements are stored on top of the stack. The stack is represented as a string of stack elements, the elements on the top of the stack is the leftmost symbol in the string. We use capital letters to represent Stack elements. The notation $A\alpha$ represents a stack with A as the top element, and an empty stack is denoted λ. The computation ofa PDA starts with the machine in state q_0, the input on the tape, and the stack empty. Notice that we have the following map for a transition function:

$$\delta(q_i, a, A) \to \{[q_j, B], [q_k, C]\}$$

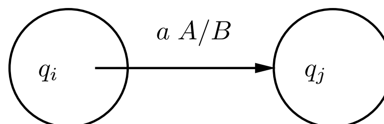
which indicates that two transitions are possible when the automaton is in state q_i, scanning an a with A on top of the stack. The transition:

$$[q_j, B] \in Im(\delta)$$

causes the machine to do all of the following:

1. Change the state from q_i to q_j
2. Process the symbol a
3. Remove A from the top of the stack
4. Push B onto the stack

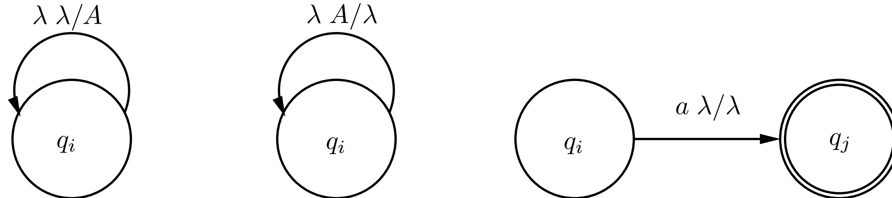
A pushdown automaton can also be depicted by a state diagram, where the labels on the arcs indicate both the input and the stack operation. For example, the transition $\delta(q_i, a, A) = \{[q_j, B]\}$ is represented by the following:



Here, the / symbol indicates replacement: A/B indicates that A is replaced by B.

Whenever λ occurs as an argument in the stack position of the transition function, the transition is applicable whenever the current state and input symbol match those in transition regardless of

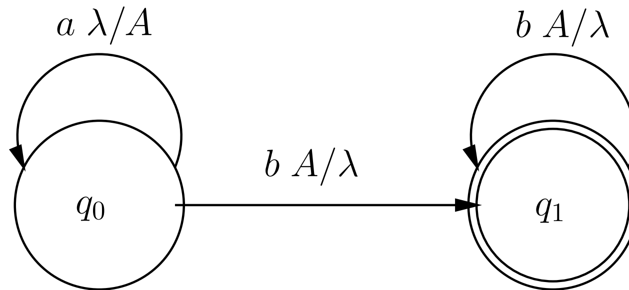
the status of the stack. In other words, the transition $[q_j, B] \in \delta(q_i, a, \lambda)$ is applicable whenever a machine is in state q_i; the application of the transaction will cause the machine to enter q_j and to add B to the top of the stack. Think of it in terms of removing nothing, and pushing on an element of Γ. A similar application can be used in the other direction; one can remove an element of the stack and push on λ. Also, one can do nothing to the stack- in which case, the machine acts like a finite deterministic automaton. Illustrated below are such transition.



A PDA configuration is represented by the triple $[a_i, w, \alpha]$, where q_i is the machine state, w is the unprocessed input, and α is the stack. The notation

$$[q_i, w, \alpha] \rightarrow_M [q_j, v, \beta]$$

indicates that the configuration $[q_j, v, \beta]$ can be obtained from $[q_i, w, \alpha]$ by a single translation of the PDA. as with before, $\rightarrow_M^*$ represents a sequence of transitions. We now have the following PD representation that can accept the language $\{q_i b^i \; |i \geq 0\}$: which give us a nice natural transition



to the following definition:

Definition

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a PDA. A string $w \in \Sigma^*$ is **accepted** by M if there is a computation:

$$[q_0, w, \lambda] \rightarrow^* [q_i, \lambda, \lambda]$$

where $q_i \in F$. The**language** of M , denoted $L(M)$ is the set of all strings accepted by M.

Definition

A PDA is **deterministic** if there is at most one transition that is applicable for each combination of state, input symbol, and stack top.

In previous chapters, we showed that deterministic and nondeterministic finite automata accepted the same family of languages. Nondeterminism was a useful design feature but did not increase the ability of the machine to accept languages. This is not the case for pushdown automata. For example, there is no deterministic PDA that accepts the language $L = \{ww^R \,|w \in \{a, b\}^*\}$.

5.2 Variations on the PDA Theme

Pushdown automata are often defined in a manner that differs from our traditional definition. There exist several altercations that preserve the set of accepted languages. Along with changing the state, a transition in a PDA is accompanied by three actions: popping the stack, pushing a stack element, and processing an input symbol. A PDA is called **atomic** if each transition causes only one of these actions to occur. Transitions in an atomic PDA have the form:

1. $[q_j, \lambda] \in \delta(q_i, a, \lambda)$
2. $[q_j, \lambda] \in \delta(q_i, \lambda, A)$
3. $[q_j, A] \in \delta(q_i, \lambda, \lambda)$

Clearly, every atomic PDA is a PDA in the sense of our definition Moreover, we have a method to construct an equivalent atomic PDA from an arbitrary PDA:
Theorem 3. *Let M be a PDA then there is an atomic PDA M' with $L(M') = L(M)$.*

Proof. This can be shown by taking all the non atomic transitions of M and replacing them by sequences of atomic transitions. For example, let $[q_j, B] \in \delta(q_i, a, A)$ be a transition of M. The atomic equivalent requires two new states, p_1, p_2, and the transitions:

$$
\begin{aligned}
&[p_1, \lambda] \in \delta(q_i, q, \lambda) \\
&\delta(p_1, \lambda, A) = \{[p_1, \lambda]\} \\
&\delta(p_2, \lambda, \lambda) = \{[q_j, B]\}
\end{aligned}
$$

□

An extended transition is an operation on a PDA that pushes a string of elements rather than juts a single element, onto the stack. The transition $[q_j, BCD] \in \delta(q_i, a, A)$ pushes BCD onto the stack with B becoming the new stack top. A PDA containing extended transitions is called an **extended PDA**. The apparent generalization does not increase the set of languages accepted by pushdown automata. Each extended PDA can be converted into an equivalent PDA in the sense of our definition.

To construct a PDA from an extended PDA, extended transitions are transformed into a sequence of transitions each of which pushes a single stack element. To achieve the result of an extended transition that pushes k elements requires $k-1$ additional states. We have the following theorem:
Theorem 4. *Let M be an extended PDA. Then there is a PDA M' such that $L(M) = L(M')$.*
Example. We can construct a standard PDA, an atomic PDA, and an extended PDA to accept the language $L = \{a^i b^{2i} \mid i \geq 1\}$. As might be expected, the atomic PDA requires more transitions and the extended PDA requires fewer transitions than the equivalent standard PDA.6

Definition

By a previous definition, an input string is accepted if there is a computation that processes the entire string and terminates in an accepting state with an empty stack. This type of acceptance is referred to as acceptance **by final state and empty stack**. Defining acceptance in terms of the final state or in the configuration of the stack alone does not change the set of languages recognized by pushdown automaton. A string w is accepted **by final state** if there exists a computation

$$[q_0, w, \lambda] \to_m^* [q_i, \lambda, \alpha]$$

where q_i is an accepting state and $\alpha \in \Gamma$. That is, a computation that processes the input and terminates in an accepting state. The contents of the stack at termination are irrelevant with acceptance by final state. A language accepted by final state is denoted L_F.

Lemma 5. *Let L be a language accepted by a PDA $M' = (Q \cup \{q_f\}, \Sigma, \Gamma, \delta', q_0, \{q_f\}, F)$ with acceptance defined by final state. Then there is a PDA that accepts L by final state and empty stack.*

Proof. Intuitively, a computation in M' that accepts a string should be identical to one in M except for the addition of transitions that empty the stack. □

Lemma 6. *Let L be a language accepted by PDA $M = (Q, \Sigma, \Gamma, \delta, q_0)$ and with acceptance defined by empty stack- that is, there is a computation that take an input string and gets to a state wherein the stack is empty, and there are no restrictions on the halting state. There is a PDA that accepts L by final state and empty stack.*

Combining these two lemmas give us the following theorem:
Theorem 7. *The following three conditions are equivalent:*

1. *The language L is accepted by some PDA.*
2. *There is a PDA M_1 with $L_F(M_1) = L$.*
3. *There is a PDA M_2 with $L_E(M_2) = L$.*

5.3 Acceptance of Context-Free Languages

Theorem 8. *Let L be a context-free language. There is then a PDA that accepts L.*

Proof. First, we need to show that every context-free language is accepted by an extended PDA. This follows from taking a language in Greibach normal form, and then constructing a PDA that accepts all words in that language. It then follows that we can construct a standard PDA that accepts that context-free language. □

Alternatively, we have the following theorem:
Theorem 9. *Let M be a PDA. Then there is a context-free grammar G with $L(G) = L(M)$.*

5.4 The Pumping Lemma for Context-Free Languages

The Pumping lemma for regular languages tells us that sufficiently long strings in a regular language have a substring that can be 'pumped' any number of times while still remaining in the language. There is an analogous lemma for context free languages.
Theorem 10. *Let L be a context-free language. There is a number k, depending on L, such that any string $z \in L$ with $length(z) > k$ can be written as $z = uvwxy$ where:*

1. *$length(vwx) \leq k$*
2. *$length(v) + length(x) > 0$*
3. *$uv^iwx^iy \in L$ for $i \geq 0$*

The Proof will be omitted, but we have the following examples:
Example. The language $L = \{a^ib^ic^i | i \geq 0\}$ is not context free. Assume the opposite. By the pumping lemma, the string $z = a^kb^kc^k$ can be decomposed into substrings $uvwxy$ that satisfy some repetition properties. Consider the possibilities for the substrings v and x. If either of these contains moer than one type of terminal symbol, then uv^2wx^2y contains a b preceeding an a or a c preceding a b. In either case, the string is not in L.

By the previous observation, v, x must be substrings of one of a^k, b^k, or c^k. Since at most one of the strings v, x is null, uv^2wx^2y increases the number of at least one, maybe two, but not all three types of terminal symbols. This implies that $uv^2wx^2y \notin L$. Thus, L is not context-free, since it does not satisfy the pumping lemma.
Example. The language $L = \{a^ib^ja^ib^j \mid i, j \geq 0\}$ is not context- free. Let k be the number specified by the pumping lemma and let $z = a^kb^ka^kb^k$. Assume there is a decomposition $uvwxy$ of z that satisfies the pumping lemma. By the first conditionn of the lemma, $length(vwx)$ can be at most k. This implies that vwx is a string containing only one type of terminal or the concatenation of two such strings. That is,

1. $vwx \in a^*$ or $vwx \in b^*$
2. $vwx \in a^*b^*$ or $vwx \in b^*a^*$

By an argument similar to the one in the previous example, the substrings v, x must only contain one type of terminal. Pumping v and x increases the number of $a's$ or $b's$ in only one of the

substrings in z. Since there is no decomposition of z satisfying the conditions of the pumping lemma, we conclude that L is not context free.

Example. The language $L = \{w \in a^* \mid length(w) \text{ is prime}\}$ is not context free. Assume that L is context free, and let n be a prime greater than k. The string a^n must have a decomposition $uvwxy$ that satisfies the conditions of the pumping lemma. Let $m = length(u) + length(w) + length(v)$. The length of any string uv^iwx^iy is $m + i(n - m)$. In particular, $length(wv^{n+1}wx^{n+1}y) = m + (n + 1)(n - m) = n(n - m + 1)$. Both of the terms in the preceding product are natural numbers greater than one, and as a result, this string is not in L. Thus, L is not context-free.

5.5 Closure Properties of Context- Free Languages

Operations that preserve context-free languages provide another tool for proving that languages are context free.

Theorem 11. *The Family of context-free languages is closed under the operations of union, concatenation, and Kleene star.*

Theorem 12. *The set of context-free languages is not closed under intersection or complementation.*

Theorem 13. *Let R be a regular language and L be a context-free language. Then, $R \cap L$ is a context- free language.*

Example. The langauge $L = \{ww \mid w \in \{a, b\}^*\}$ is not context-free but $\overline{L}$ is. First we show that L is not context-free using a proof by contradiction. Assume that L is context free. Then, by one of our above theorems,

$$L \bigcap a^*b^*a^*b^* = \{a^ib^ja^ib^j \mid i, j \geq 0\}$$

is context free, and we know that it isn't - contradicting our assumption. To show that $\overline{L}$ is context free, we construct two context-free grammars G_1, G_2 such that $L(G_1) \cup L(G_2) = \overline{L}$. What one can do is to let G_1 generate all even-length strings in $\{a, b\}^*$ and G_2 generate all odd length strings in $\{a, b\}^*$. Both of these can be shown to be context free, in which case we are done.

Chapter 6

Turing Machines

6.1 The Standard Turing Machine

The Turing machine is a finite-state machine in which a transition prints a symbol on a 'tape'. The tape head may move in either direction, allowing the machine to read and manipulate the input as many times as desired.

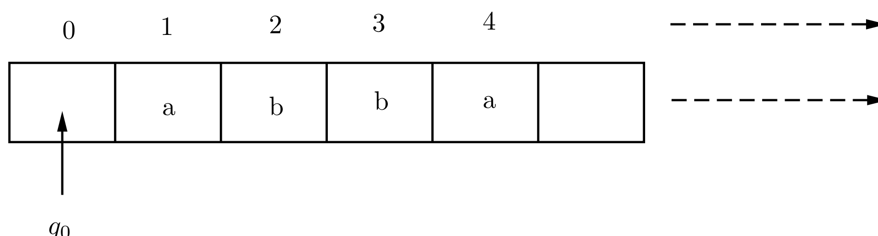
Definition

A **Turing Machine** is a quintuple $M = (Q, \Sigma, \Gamma, \delta, q_0)$ where Q is a finite set of states, Γ is a finite set called the tape alphabet, Γ contains a special symbol B that represents a blank, Σ is a subset of $\Gamma - \{B\}$ called the **input alphabet**, δ is a partial function from

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

called the transition function, and q_0 is a distinguished state called that start state.

The tape of a Turing machine has a left boundary, and extends indefinitely to the right. Tape positions are numbered by the natural numbers, with the leftmost position numbered zero. Each tape position contains one element from the tape alphabet.



Every computation starts with the machine in state q_0 and the tape head scanning the leftmost position. The input, a string from Σ^* is written on the tape beginning at position one. A transition consists of three actions: changing the state, writing a symbol on the square scanned by the tape head, and moving the tape head. The direction of the movement is specified by the final component of the transition. An L indicates a movement to the left, where R indicates a movement to the right.

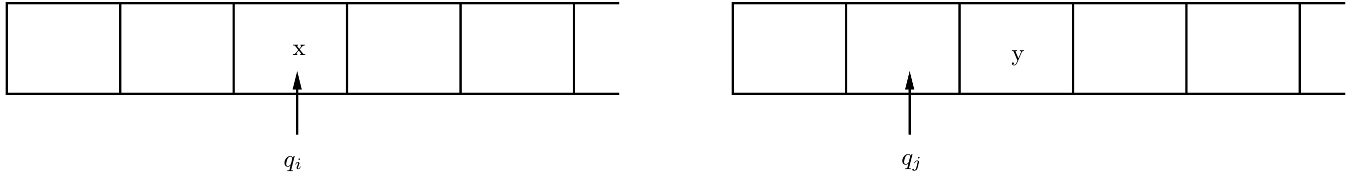


Figure 6.1: Illustration when $\delta(q_i, x) = [q_j, y, L]$

A computation halts when it encounters a state, symbol pair for which no transition is defined. A transition from tape position zero may specify a move to the left of the boundary of the tape. When this occurs, the computation is said to **terminate abnormally**. The Turing machine in our definition is deterministic, that is, at most one transition is specified for every combination of state and tape symbol. The one-tape deterministic Turing machine with initial conditions above is refered to as the **standard Turing machine**.

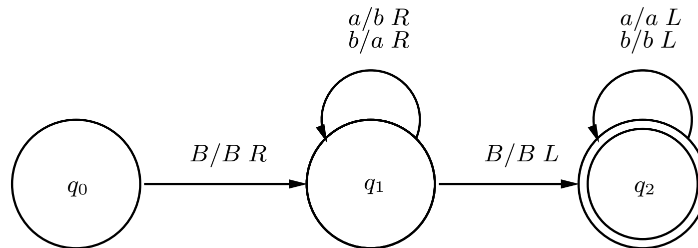


Figure 6.2: Turing machines can be graphically represented by a state diagram.

Example. There is a Truing machine with input alphabet $\{a, b\}$ that produces a copy of the input string. That is, the computation that begins with BuB ends with $BuBuB$. Try to work out how this is possible, the answer is example 8.1.2 in the textbook.

6.2 Turing Machines as Language Acceptors

The interesting thing about Turing machines is that they can be used to accepted languages and to compute functions. The result of a computation can be defined in terms of the state in which the computation terminates or the configuration of the tape at the end of the computation.

Unlike finite-state and pushdown automata, a Turing machine need not read the entire input string to accept the string. A Turing machine augmented with final states is a sextuple:

$$(Q, \Sigma, \Gamma, \delta, q_0 m F)$$

where $F \subseteq Q$ is the set of final states of the Turing machine.

Definition

Let $M = (Q, \Sigma, \Gamma, \delta, q_0 mF)$ be a Turing machine. A string $u \in \Sigma^*$ is **accepted by final state** if the computation of M with input u halts in a final state. A computation that terminates abnormally rejects the input regardless of the state in which the machine halts. The language of M, usually denoted $L(M)$ is the set of all strings accepted by M.

A language accepted by a Turing machine is called a **recursively enumerable language**. The ability of a Turing machine to move in both directions and process blanks introduces the possibility that the machine may not halt for a particular input. Thus there are three possible outcomes for a Turing machine computation: it may halt and accept the input string; half and reject the string; or it may not halt at all. Because of the last possibility, we will sometimes say that a machine M recognizes L if it accepts L, but does not necessarily halt for all input strings. The computations of M identify the strings L but may not provide answers for strings not in L. A language accepted by a Turing machine that halts for all input strings is said to be **recursive**.

6.3 Alternative Acceptance Criteria

Using our prior definition, the acceptance of a string by a Turing machine is determined by the state of the machine when the computation halts. Alternative approaches to defining acceptance will be presented in this section.

Definition

Let $M = (Q, \Sigma, \Gamma, \delta, q_0 mF)$ be a Turing machine. A string $u \in \Sigma^*$ is **accepted by halting** if the computation of M with input u halts normally.

Turing machines designed for acceptance by halting are used for language recognition. The computation for any input not in the language will not terminate. The next theorem shows that any language recognized by a machine that accepts by halting is also accepted by a machine that accepts by final state:
Theorem 14. *The following statements are equivalent:*

1. *The language L is accepted by a Turing machine that accepts by final state.*
2. *The language L is accepted by a Turing machine that accepts by halting.*

6.4 Multitrack Machines

A multitrack tape is one in which the tape is divided into tracks. A tape position in an n-track tape contains n symbols from the tape alphabet. The machine reads and entire tape position. Multiple tracks increase the amount of information that can be considered when determining the appropriate transition. A tape position in a two-track machine is represented by the ordered pair $[x, y]$ where x is the symbol in track 1, and y is the symbol in track 2.

The states, input alphabet, tape alphabet, initial state, and final states of a two track machine are the same as in the standard Turing machine. A two-track transition reads and rewrites the entire tape position. A transition of a two track machine is written as follows:

$$\delta(q_i, [x, y]) = [q_j, [z, w], d]$$

where $d \in \{L, R\}$.
Theorem 15. *A language L is accepted by a two-track Turing machine if and only if it is accepted by a standard Turing machine.*

6.5 Two-Way Tape Machines

A Turing machine with a two-way tape is identical to the standard model except that the tape extends indefinitely in both directions. Since a two-way tape has no left boundary, the input can be placed anywhere on the tape. A"ll other tape positions are assumed to be blank. "The tape head is initially positioned on the blank to the immediate left of the input string. The advantage of a two way tape is that the Turing machine designed need not worry about crossing the left boundary of the tape.

It turns out that this new type of Turing machine is completely the same as a regular Turing machine:
Theorem 16. *A language L is accepted by a Turing machine with a two-way tape if and only if it is accepted by a standard Turing machine.*

6.6 Multitape Machines

A k-tape machine has k tapes and k independent tape heads. The states and alphabets of a multitape machine are the same as in a standard Turing machine. The machine reads the tapes simultaneously but has only on state- this is depicted by attaching each of the independent tape heads to a single control indicating the current state.

A transition in a multitape machine may:

1. Change the state
2. Write a symbol on each of the tapes
3. Independently reposition each of the tape heads

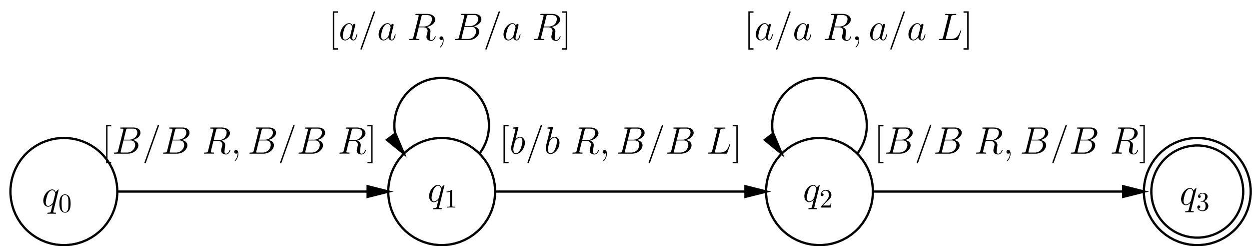
The repositioning consists of moving the tape head one square to the right, one square to the left, or leaving it at its current position. A transition of a two-tape machine scanning x_1 on tape 1 and x_2 on tape 2 is written:

$$\delta(q_i, x_1, x_2) = [q_j; y_1, d_1; y_2, d_2]$$

where $x_i, y_i \in \Gamma$ and $d_i \in [L, R, S]$. This transition causes the machine to write y_i on tape i. The symbol d_i represents the direction of the movement of tape head i.

The input to a multitape machine is placed in the standard position on tape 1. All the other tapes are assumed to be blank. The tape heads originally scan the leftmost position of each tape. A multitape machine can be represented by a state diagram in which the label on an arc specifies the action of each tape.

The advantages of multitape machines are the ability to copy data between tapes and to compare strings on different tapes. For example,



accepts the language $\{a^iba^i \mid i \geq 0\}$. A computation with input string a^iba^i copies the leading $a's$ to tape 2 in state q_1, and when the b is read on tape 1, the computation enters the state q_2 to compare the number of $a's$ on tape 2 with the $a's$ after the b on tape 1.

A standard Turing machine is a multitape Turing machine with a single tape. Consequently, every recursively enumerable language is accepted by a multitape machine.
Theorem 17. *A language L is accepted by a multitape Turing machine if and only if it is accepted by a standard Turing machine.*

6.7 Nondeterministic Turing Machines

A nondeterministic Turing machine may specify any finite number of transitions for a given configuration. The components of a nondeterministic machine with the exception of the transition function are identical to those of the standard Turing machine. Transitions in a nondeterministic machine are defined by a function from $Q \times \Gamma$ to subsets of $Q \times \Gamma \times \{L, R\}$. An input string is accepted by a nondeterministic Turing machine if there is at least one computation that terminates in an accepting state. The existence of other computations that halt in non-accepting states or fail to halt altogether is irrelevant.
Example.

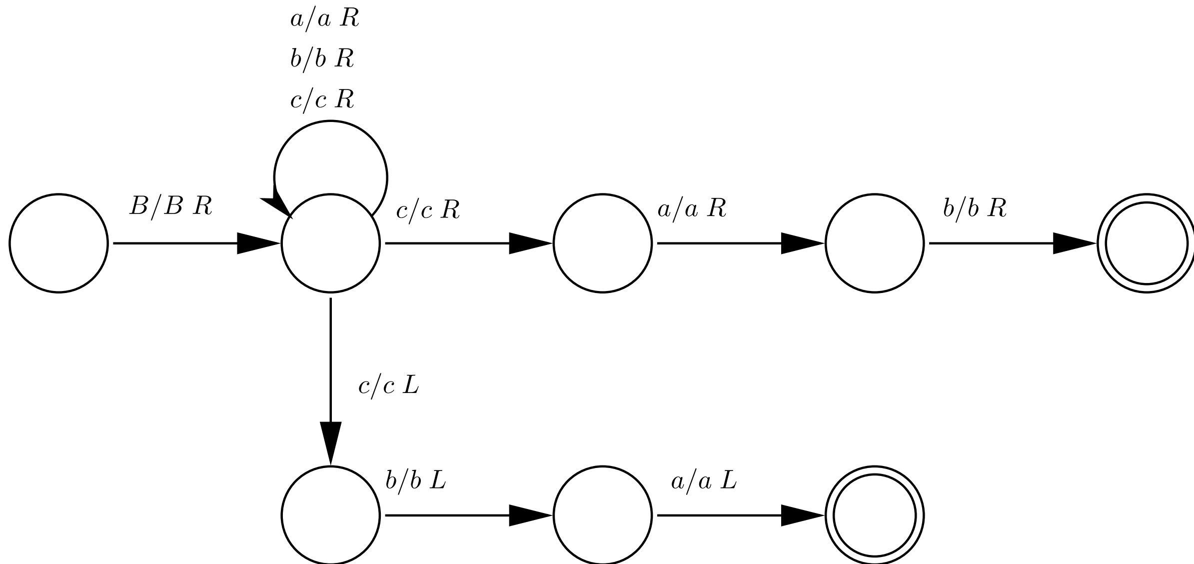


Figure 6.3: This machine accepts strings containing a c preceeded or followed by ab.

6.8 Turing Machines as Language Enumerators

In the previous sections, Turing machines have been used as language acceptors: A machine is given an input string, and the result of the computation indicates the acceptability of the input. Turing machines may also be designed to enumerate a language. The computation of such a machine sequentially produces an exhaustive listing of the elements of the language. An enumerating machine has no inputs, its computation continues until it has generated every string in the language.

Like Turing machines that accept languages, there are a number of equivalent ways to define an enumerating machine. We will use a k-tape deterministic machine, $k \geq 2$ as the underlying Turing machine model in the definition of enumerating machines. The first tape is the output tape and the remaining tapes are work tapes. A special tape symbol # is used on the output tape to separate the elements of the language generated during the computation. The machines in this section perform two things, acceptance and enumeration. To distinguish them, a machine that accepts a language will be denoted M and an enumerating machine will be denoted E.

Definition

A k-tape Turing machine $E = (Q, \Sigma, \Gamma, \delta, q_0)$ **enumerates** the language K if:

1. The computation begins with all tapes blank
2. With each transition, the tape head on tape 1 (the output tape) remains stationary or moves to the right

3. At any point in the computation, the nonblank portion of tape 1 has the form

$$B\#u_1\#u_2\#...\#u_k\# \quad \text{or} \quad B\#u_1\#u_2\#...\#u_k\#v$$

   where $u_i \in L$ and $v \in \Sigma^*$.

4. A string u will be written on tape 1 preceded and followed by # if and only if $u \in L$.

Theorem 18. *Let L be a language enumerated by a Turing machine E. Then there is a Turing machine E' that enumerates L and each string in L appears only once on the output tape of E'.*

Proof. The idea is essentially to add one more tape that acts as a 'is this word already on the output tape'? Checker. □

Theorem 19. *If L is enumerated by a Turing machine, then L is recursively enumerable.*

Chapter 7

Turing Computable Functions

7.1 Computation of Functions

A function $f : X \to Y$ is a mapping that assigns at most one value from the set Y to each element of the domain X. Adopting a computational viewpoint, we refer to the variables of f as the input of the function. The definition of a function does not specify $f(x)$. Turing machines will be designed to compute the values of functions. The domain and range of a function computed by a Turing machine consist of string over the input alphabet of the machines.

A Turing machine that computes a function has two distinguished states: the initial state q_0 and the halting state q_f. A computation begins with a transition from state q_0 that positions the tape head at the beginning of the input string. The state q_0 is never reentered; its sole purpose is to initiate the computation. All computations that terminate do so in state q_f with the value of the function written on the tape beginning at position one.

Definition

A deterministic one-tape Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_f)$ computes the unary function $f : \Sigma^* \to \Sigma^*$ if

1. There is only one transition from the state q_0 and it has the form $\delta(q_0, B) = [q_i, B, R]$.
2. There are no transitions of the form $\delta(q_i, x) = [q_o, y, d]$ for any $q_i \in Q$, $x, y\ in \Gamma$ and $d \in \{L, R\}$
3. THere are no transitions of the form $\delta(q_f, B)$
4. The computation with input u halts in the configuration $q_f BvB$ whenever $f(u) = v$;
5. The computation continues indefinitely whenever $f(u) \uparrow$

---

a Function is said to be **Turing Computable** if there is a Turing machine that computes it. A

Turing machine that computes a function f may fail to halt for an input string u. In this case, f is undefined for u. Thus Turing machines can compute both total and partial functions.

An arbitrary need not have the same domain and range. Turing machines can be designed to compute functions from Σ^* to a specific set R by designating an input alphabet Σ and range R. Condition 4 is then interpreted as requiring the string v to be an element of R.

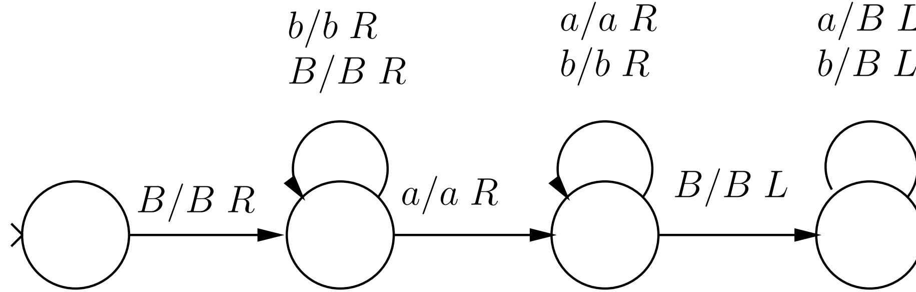


Figure 7.1: This Turing machine computes the partial function f from $\{a, b^*\}$ to $\{a, b\}^*$.

Where the function f is defined as follows:

$$f(u) = \begin{cases} \lambda & \text{if } u\text{u contains an } a \\ \uparrow & \text{otherwise} \end{cases}$$

7.2 Numeric Computation

We now turn to things called **number-theoretic functions**. A number theoretic function takes the form:

$$f : \mathbb{N} \times \mathbb{N} \times ... \times \mathbb{N} \to \mathbb{N}$$

Example. The function $f(n) = n^2$ is a unary number-theoretic function.

The transition from symbolic to numeric computations requires only a change of perspective, since we can represent numbers by strings of symbols. The input alphabet of the Turing machine is determined by the representation of natural numbers used in the computation. We will represent the number n by the string 1^{n+1}. This is called the unary notation of n, and is denoted $\overline{n}$. When numbers are encoded using the unary representation, the input alphabet for a machine that computes a number theoretic function is the singleton set $\{1\}$.

A k-variable total number-theoretic function $r : \mathbb{N} \times \ltimes \times ... \times \mathbb{N} \to \{0, 1\}$ defines a k-ary relation R on the domain of the function. The relation is defined by:

$$[n_1, n_2, ..., n_k] \in R \text{ if } r(n_1, n_2, ..., n_k) = 1$$
$$[n_1, n_2, ..., n_k] \notin R \text{ if } r(n_1, n_2, ..., n_k) = 0$$

The function r is called the **characteristic function** of the relation R, and a relation is Turing computable if its characteristic function is Turing computable.
Example.

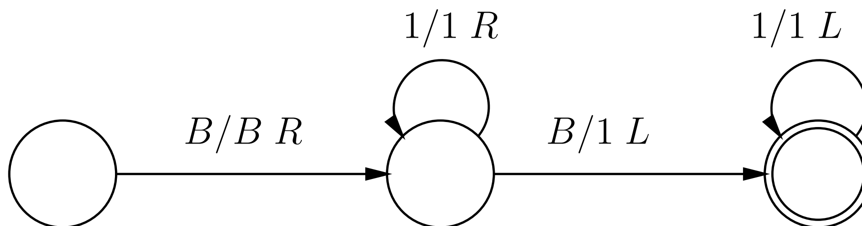


Figure 7.2: This Machine computes the successor function s(n) = n+1

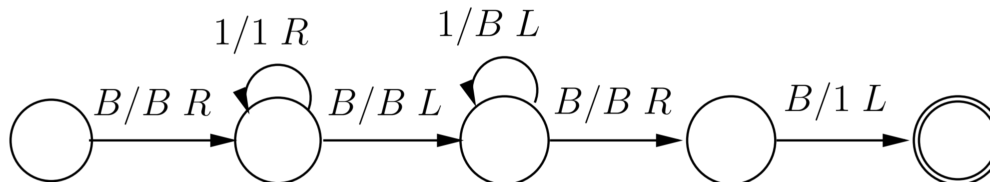


Figure 7.3: The zero function, $z(n) = 0$

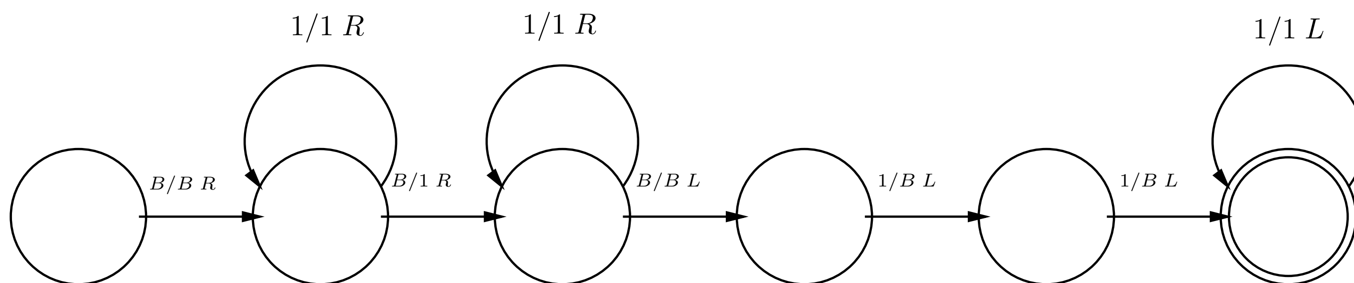
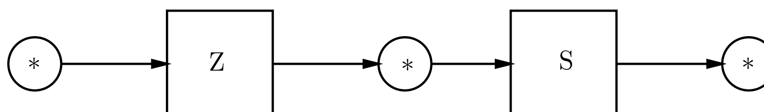


Figure 7.4: This Turing Machine computes the binary function defined by the addition of natural numbers.

7.3 Sequential Operation of Turing Machines

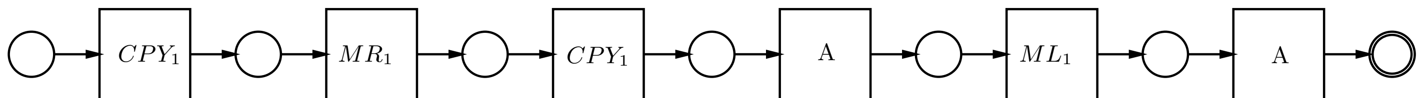
Turing machines designed to accomplish a single task can be combined to construct machines that perform complex computations. Intuitively, the combination is obtained by running the machines sequentially. The result of one computation becomes the input for the succeeding machine. For example, by running the zero function then running the successor function, we get $f(n) = 1$ for all n. We denote the sequential combination of two machines in the following way:



There are certain sequences of actions that frequently occur in a computation of a Turing machine. Machines can be constructed to perform these recurring tasks. These machines are designed in a manner that allows them to be used as components in more complicated machines. We call machines constructed to perform a single simple task a **macro**. A few are listed here:

1. MR_i (move right) requires a sequence of at least i natural numbers to the immediate right of the tape at the initiation of a computation.
2. ML_k (move left)requires a sequence of at least i natural numbers to the immediate left of the tape at the initiation of a computation.
3. FR and FL are the 'fine' macros, which move the tape head into a position to process the first natural number to the right or left of the current position.
4. E_k (erase) erases a sequence of k natural numbers and halts with the tape head in its original position.
5. CPY_k and $CPY_{k,i}$ produce a copy of the designated number of integers. The segment of the tape on which the copy is produced is assumed to be blank- $CPY_{k,i}$ expects a sequence of $k+1$ numbers followed by a blank segment large enough to hold a copy of the first k.
6. T (translate) is the translate macro, and changes the location of the first natural number to the right of the of the tape head.

The macros and previously constructed machines can be used to design a Turing machine that computes the function $f(n) = 3n$:



7.4 Composition of Functions

Composition of functions is the exact same as in linear algebra. We have the following though:

Definition

Let g, h be unary number-theoretic functions. The **composition** of h with g, denoted $h \circ g$ is a function f that satisfies the following:

$$f(x) = \begin{cases} \uparrow & \text{if } g(x) \uparrow \\ \uparrow & \text{if } g(x) = y \text{ and } h(y) \uparrow \\ h(y) & \text{if } g(x) = y \text{ and } h(y) \downarrow \end{cases}$$

Theorem 20. *The Turing computable functions are closed under the operation of composition.*

Chapter 8

The Chomsky Hierarchy

8.1 Unrestricted Grammars

The components of a phase-structure grammar are the same as those of the regular and context-free grammars studied in chapter 3. A phase-structure grammar consists of a finite set V of variables, an alphabet Σ, a start variable, and a set of rules. A rule has the form $u \to v$ where u, v can be any combination of variables and terminals, and defines a permissible string of transformation. The application of a rule to a string z is a two step process consisting of:

1. Matching the left-hand side of the rule to a substring of z, and
2. Replacing the left-hand side with the right-hand side.

The unrestricted grammars are the largest class of phase-structure grammars. There are no constraints on a rule other than requiring that the left-hand must not be null.

Definition

An **unrestricted grammar** is a quadruple (V, Σ, P, S) where V is a finite set of variables; Σ is a finite set of terminal symbols, P is a set of rules, and S is a distinguished element of V. A production of an unrestricted grammar has the form $u \to v$ where $u \in (V \cup \Sigma)^+$ and $v \in (V \cup \Sigma)^*$. The sets V and Σ are assumed to be disjoint.

Example. The unrestricted grammar with $V = \{S, A, C\}, \Sigma = \{a, b, c\}$ and rules:

$$
\begin{aligned}
&S \to aAbc \mid \lambda \\
&A \to aAbC \mid \lambda \\
&Cb \to bC \\
&Cc \to cc
\end{aligned}
$$

with a start symbol S generates the language $\{a^i b^i c^i \mid i \geq 0\}$. The rule $Cb \to bC$ allows the final C to pass through the $b's$ that separate it from the $c's$ at the end of the string.
Theorem 21. *Let $G = (V, \Sigma, P, S)$ be an unrestricted grammar. Then $L(G)$ is a recursively enumerable language.*

8.2 Context- Sensitive Grammars

The context-sensitive grammars represent an intermediate step between context-free and unrestricted grammar. No restrictions are placed on the left handed side of a production, but the length of the right hand side is required to be at least that of the left.

Definition

A phase-structure grammar $G = (V, \Sigma, P, S)$ is called **context-sensitive** if each rule has the form $u \to v$ where $u \in (V \cup \Sigma)^+$, and $length(u) \leq length(v)$.

Example. The following rules generate the language $\{a^i b^i c^i \mid i > 0\}$, and satisfy the conditions of a context-sensitive rules.

$$S \to aAbc \mid abc$$
$$A \to aAbC \mid abC$$

Theorem 22. *Every context-sensitive grammar is recursive.*

8.2.1 Linear-Bounded Automata

Restricting the amount of tape that a Turing machine can use decreases the capabilities of a Turing machine. A linear-bounded automaton is a Turing machine in which the amount of available tape is determined by the length of the input string. The input alphabet contains two symbols, $<$ and $>$, that designate the left and right boundaries of the tape.

Definition

A **linear-bounded automaton** (LBA) is a structure $M = (Q, \Sigma, \Gamma, \delta, q_0, <, >, F)$. Where $Q, \Sigma, \Gamma, \delta, q_0$, and F are the same as for a nondeterministic Turing machine. The symbols $<$ and $>$ are distinguished elements of Σ.

Theorem 23. *Let L be a context-sensitive language. Then there is a linear bounded automaton M with $L(M) = L$.*
Theorem 24. *Let L be a language accepted by a linear bounded automaton. Then $L - \{\lambda\}$ is a context-sensitive language.*

8.2.2 Chomsky Hierarchy

Chomsky numbered the four families of grammars that make up a heirarchy. It is structured as follows:

Grammars	Languages	Accepting Machines
Type 0 Grammars, Phase-structure Grammars, Unrestricted Grammars	Recursively Enumerable	Turing machine, nondeterministic Turing machine
Type 1 Grammars, Context-Sensitive Grammars	Context- Sensitive	Linear-Bounded Automata
Type 2 Grammars ,	Context- Free	Pushdown Automata
Type 3 Grammars , Regular Grammars, Left-linear Grammars, Right-linear Grammars	Regular	Deterministic finite automata, nondeterministic finite automata

Chapter 9

Decidability and Undecidability

A **decision problem P** is a set of related questions, each of which has a yes or no answer. The decision problem of determining if a number is a perfect square consists of the following questions:

$$p_0: \text{ Is 0 a perfect square?}$$
$$p_1: \text{ Is 1 a perfect square?}$$
$$p_2: \text{ Is 2 a perfect square?}$$

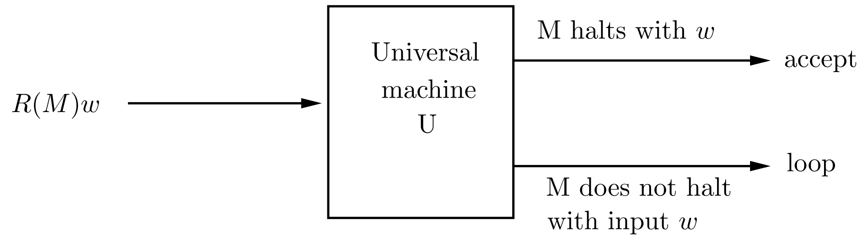
A decision is said to be **decidable** if it has a solution. A solution to a decision problem is an algorithm that determines the appropriate answer to every questions $p \in \mathbf{P}$.

9.1 Church-Turing Thesis

Theorem 25. *There is an effective procedure to solve a decision problem if and only if there is a Turing machine that halts for all input strings and solves the problem.*
Theorem 26. *A decision problem P is partially solvable if and only if there is a Turing machine that accepts precisely the instances of P whose answer is yes.*
Theorem 27. *A function f is effectively computable if and only if there is a Turing machine that computes f.*

9.2 Universal Machines

A **universal Turing machine** is designed to simulate the computations of an arbitrary Turing machine M. To do so, the input to the universal machine must contain a representation of the machine M and the string w to be processed by M. For simplicity, we will assume that M is a standard Turing machine that accepts by halting. The action of a universal machine U is depicted by:
Theorem 28. *The language $L_H = \{R(M)w \mid \ M \text{ halts with input } w \ \}$.*



9.3 The Halting Problem for Turing Machines

The halting problem may be formulated as follows: Given an arbitrary Turing machine M with input alphabet Σ and a string $w \in \Sigma^*$, will the computation of M with input w halt?

Theorem 29. *The halting problem for Turing Machines is undecidable.*

9.4 Problem Reduction and Undecidability

Reduction was introduced a few chapters ago as a tool for constructing solutions to decision problems. A decision problem P is reducible to Q if there is a Turing computable function that transforms instances of P into instances of Q, and the transformation preserves the answer to the problem instance of P.

Theorem 30. *There is no algorithm that determines whether an arbitrary Turing machine halts when a computation is initiated with a blank tape.*

9.5 Rice's Theorem

Theorem 31. *If $\mathbb{P}$ is a nontrivial property of recursively enumerable languages, then $L_{\mathbb{P}}$ is not recursive.*

Chapter 10

μ-Recursive Functions

A family of intuitively computable number-theoretic functions, known as the primitive recursive functions, is obtained from the basic functions:

1. The successor function $s : s(x) = x + 1$
2. The zero function $z : z(x) = 0$
3. The projection function $p_i^{(n)} : p_i^{(n)}(x_1, x_2, ...x_n) = x_i, \;\; 1 \leq i \leq n$

Definition

Let g and h be total number-theoretic functions with n and $n+2$ variables, respectively. The $n+1$ variable function f defined by:

$$f(x_1, ...x_n, 0) = g(x_1, ..., x_n)$$
$$f(x_1, ...x_n, y + 1) = h(x_1, x_2, ...x_n, y, f(x_1, ...x_n, y))$$

Is said to be obtained from g and h by **primitive recursion**

Where x_i is said to be the parameters for a definition by primitive recursion, and y is said to be the **recursive variable**. The algorithm for computing $f(x_1, ..., x_n, y)$ whenever g, h are available follows from our definition. We have the following:

$$f(x_1, ...x_n, 0) = g(x_1, ...x_n)$$

And since $f(x_1, ...x_n, y)$ is obtained from h by using the parameters $x_1, ...x_n$, treating y as a recursive variable, and $f(x_1, ...x_n, y)$ as the previous value of the function, we have a definition for f.

Definition

A function is **primitive recursive** if it can be obtained from the successor, zero, and projection functions by a finite number of applications of composition and primitive recursion.

Example. $f = add$ can be calculated from $g(x) = x$ and $h(x, y, z) = z + 1$ by the following:

$$add(x, 0) = g(x) = x$$
$$add(x, y + 1) = h(x, y, add(x, y)) = add(x, y) + 1$$

Theorem 32. *Every primitive recursive function is Turing computable*

Theorem 33. *Let g be a primitive recursive function and f a total function that is identical to g for all but a finite number of input values. Then f is primitive recursive.*

Chapter 11

Time Complexity

Definition

Let $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$ be one-variable number-theoretic functions.

1. The function f is said to be of order g if there exists a positive constant c and a natural number n_0 such that $f(n) \leq c\dot{g}(n)$ for all $n \geq n_0$.
2. The set of all functions of order g is denoted $O(g) = \{f | f \text{ is of order } g\}$

When f is of order g we say that g provides an asymptotic upper bound on f.
Example. Let $f = n^2, g = n^3$. Then, $f \in O(g)$ but $g \notin O(f)$.

If f has the same rate of growth as g, then g is said to be an asympotically tight bound on f. The set:

$$\Theta(g) = \{f \mid f \in O(g) \text{ and } g \in O(f)\}$$

Theorem 34. *Let f be a polynomial of degree r. Then:*

1. $f \in \Theta(n^r)$
2. $f \in O(n^k)$ *for all* $k > r$
3. $f \notin O(n^k)$ *for all* $k < r$

Definition

Let M be a standard Turing machine. The **time complexity** of M is the function

$$tc_M : \mathbb{N} \to \mathbb{N}$$

such that $tc_M(n)$ is the maximum number of transitions processed by a computation of M when initiated with an input string of length n. In other words, this definition of time complexity measures the worst-case performance of the Turing machine.

Theorem 35. *Let L be the language accepted by a $k-$track deterministic Turing machine M with time complexity $tc_M(n)$. Then L is accepted by a standard Turing machine M' with time complexity $tc_{M'}(n) = tc_M(n)$.*

Theorem 36. *Let L be the language accepted by a k-tape deterministic Turing machine M with time complexity $tc_M(n) = f(n)$. Then L is accepted by a standard Turing machine N with time complexity $tc_N(n) \in O(f(n)^2)$.*

Theorem 37. *Let M be a k-tape Turing machine , $k > 1$ that accepts L with $tc_M(n) = f(n)$. For any constant $c > 0$, there is a k-tape machine N that accepts L with $tc_N(n) \leq [cf(n)] + 2n + 3$.*

Corollary 38. *Let M be a one-tape Turing machine that accepts L with $tc_M(n) = f(n)$. For any constant $c > 0$, there is a two-tape machine N that accepts L with $tc_N(n) \leq [cf(n)] + 2n + 3$.*

Chapter 12

$\mathcal{P}, \mathcal{NP}$, and Cook's Theorem