EECS 510 Lecture Notes

Andrew Cousino

May 12, 2022

Chapter 1

Preliminaries

Definition. An *alphabet* is a finite, nonempty set of symbols usually denoted by Σ.

Example 1.1. Common alphabets include the following.

- $\Sigma = \{0, 1\}$ is the binary alphabet.
- The set of all UTF-8 characters, or the set of all printable UTF-8 characters.

$\square$

Definition. A *string* (or sometimes *word*) is a finite sequence of symbols chosen from an alphabet. The *empty string*, denoted by ϵ, is the string with zero symbols. The *length of a string* s is denoted by $|s|$ and is the length of the sequence. The *concatenation of strings* s_1 and s_2 is denoted by $s_1 \cdot s_2$ or just $s_1 s_2$.

Note the empty string ϵ is a string over all alphabets, and is the identity for concatenation: $\epsilon s = s\epsilon = s$ for all strings s.

Example 1.2. From the binary alphabet, $\Sigma = \{0, 1\}$, we have the following strings 01011 which has length 5, ϵ which has length 0, and 1111 which has length 4. $\square$

Definition. If Σ is an alphabet, then we recursively define Σ^n as $\Sigma^0 := \{\epsilon\}$ and $\Sigma^{n+1} := \Sigma^n \cdot \Sigma = \{s \cdot c \mid s \in \Sigma^n, c \in \Sigma\}$.

Example 1.3. For the binary alphabet, $\Sigma = \{0, 1\}$, we have the following.

- $\Sigma^0 = \{\epsilon\}$
- $\Sigma^1 = \{0, 1\}$
- $\Sigma^2 = \{00, 01, 10, 11\}$
- $\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$
- Σ^n are the strings of length n over the binary alphabet

$\square$

Definition. The *set of all strings over an alphabet* Σ, or the *Kleene closure of* Σ, is denoted as

$$\Sigma^* := \bigcup_{n=0}^{\infty} \Sigma^n = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots.$$

The *set of all non-empty strings over an alphabet* Σ is denoted as

$$\Sigma^+ := \bigcup_{n=1}^{\infty} \Sigma^n = \Sigma^1 \cup \Sigma^2 \cup \cdots.$$

Definition. A subset of Σ^* is called a *language*.

Example 1.4. Examples of languages include the following.

- Σ^*, $\emptyset$, $\{\epsilon\}$ are three distinct languages for any alphabet Σ.
- The language of all strings consisting of any number of zeros followed by a one, $\{0^n1 \mid n \geq 0\} = \{1, 01, 001, 0001, \dots\}$.
- The language of all strings consisting of n zeros followed by n ones, $\{0^n1^n \mid n \geq 0\} = \{\epsilon, 01, 0011, \dots\}$.
- The language of legal C programs is a language over UTF-8.

□

Chapter 2

Finite Automata

2.1 Deterministic Finite Automata

2.1.1 Transition Diagrams

We start by presenting the transition diagrams for two different deterministic finite automata.

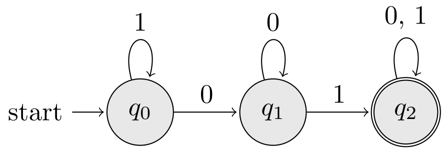


Figure 2.1: A DFA diagram

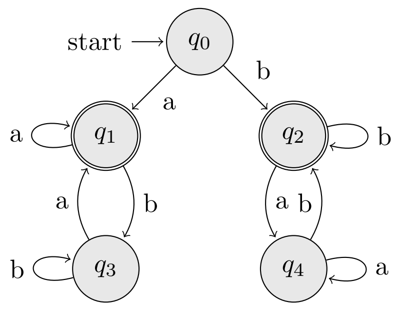


Figure 2.2: Another DFA diagram

Notice how (1) all the states and transitions are labelled, (2) there is an arrow labelled *Start* pointing to the starting state, and (3) the final/accepting states are denoted with a double circle while the non-accepting states have only a single circle.

To process a string, as an example 1011, through a diagram, in this case figure 2.1, we begin at the initial state, q_0. Follow the transition corresponding to the first symbol, which in this case is 1, to the next state, q_0. Continue to follow the transition matching the next symbol in the string until there are no more symbols. If we end in an accepting state, as we do in this example, the string is accepted into the language of the DFA. Otherwise, it is said to be rejected. Some other strings that are in this language are 01, 101, and 110100; other strings that are not in this language are ϵ, 10, and 111000.

Current state	Current symbol	Next state
q_0	1	q_0
q_0	0	q_1
q_1	1	q_2
q_2	1	q_2

Table 2.1: Running 1011 through figure 2.1

2.1.2 Transition Table

The transition table for figure 2.1 is the following. We have a row for every state, and a column for every symbol of the alphabet. The starting state is indicated by the arrow, and the accepting states are starred. The table gives all values of the transition function, which is a function from pairs of a state and a symbol to the set of states.

	0	1
$\rightarrow q_0$	q_1	q_0
q_1	q_1	q_2
$*q_2$	q_2	q_2

Table 2.2: Table corresponding to figure 2.1

	a	b
$\rightarrow q_0$	q_1	q_2
$*q_1$	q_1	q_3
$*q_2$	q_4	q_2
q_3	q_1	q_3
q_4	q_4	q_2

Table 2.3: Table corresponding to figure 2.2

2.1.3 Formal Definition

Definition. A *deterministic finite automata (DFA)* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ which consists of the following.

1. Q is a finite, non-empty set of *states*
2. Σ is an alphabet of *input symbols*
3. $\delta : Q \times \Sigma \rightarrow Q$ is a *transition function*
4. $q_0 \in Q$ is the *initial state*
5. $F \subseteq Q$ is the set of *accepting states.*

2.1.4 Extending the Transition Function

We would like to run entire strings through a DFA rather than look at single transitions on a single symbol. So what we want is a function $\hat{\delta}$ that operates on entire strings, i.e. $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$. We shall define it recursively in the natural way. But first we illustrate this natural definition with an example.

Example 2.1. Again, we will run 1011 through the DFA in figure 2.1. We start with the base case of $\hat{\delta}(q_0, \epsilon) = q_0$, which is the transition from the initial state on the empty string. The recursive cases will take one more step from the previous step.

1. $\hat{\delta}(q_0, \epsilon) = q_0$
2. $\hat{\delta}(q_0, 1) = \delta(\hat{\delta}(q_0, \epsilon), 1) = \delta(q_0, 1) = q_0$
3. $\hat{\delta}(q_0, 10) = \delta(\hat{\delta}(q_0, 1), 0) = \delta(q_0, 0) = q_1$
4. $\hat{\delta}(q_0, 101) = \delta(\hat{\delta}(q_0, 10), 1) = \delta(q_1, 1) = q_2$
5. $\hat{\delta}(q_0, 1011) = \delta(\hat{\delta}(q_0, 101), 1) = \delta(q_2, 1) = q_2$

□

The base case will be $\hat{\delta}$ operating on the empty string ϵ: for any $q \in Q$, $\hat{\delta}(q, \epsilon) \coloneqq q$. So on an empty input, we follow no transitions by staying put. For the recursive case, we decompose an arbitrary non-empty string w into its last symbol a and the initial slice v, so $w = va$. Then

$$\hat{\delta}(q, w) = \hat{\delta}(q, va) \coloneqq \delta(\hat{\delta}(q, v), a).$$

Note that $p \coloneqq \hat{\delta}(q, v)$ is another state, obtained by starting at q and transitioning according to the input v. We can now say that $\hat{\delta}(q, va) = \delta(\hat{\delta}(q, v), a) = \delta(p, a)$.

2.1.5 Language of a DFA

Definition. The *language of a DFA* $A = (Q, \Sigma, \delta, q_0, F)$ is denoted by $L(A)$ and defined as

$$L(A) \coloneqq \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}.$$

For any language L, if there is a DFA A for which $L = L(A)$, then L is said to be a *regular language*. We will also say that L is the *language accepted by DFA A*.

Example 2.2. The language of the DFA in figure 2.1 is

$$\{w \mid w \text{ contains 01 as a substring}\}.$$

The language of figure 2.2 is

$$\{w \mid w \text{ begins and ends with } a \text{ or begins and ends with } b\}.$$

□

2.2 Nondeterministic Finite Automata

A *nondeterministic* finite automata (NFA) is like the deterministic variety except with the ability to explore multiple states "in parallel". Alternatively, NFAs can be said to "guess" a property about the input, such as guess about being at the beginning/end of the input.

Example 2.3. The following is a diagram of an NFA which recognizes all inputs that end in 01. The transition table for this diagram is as follows. □

As the above example suggests, the transition function δ for an NFA will take a state and a symbol and return a *set of states* rather than one single state. This means that a transition can go to either many states, exactly one state, or precisely no states (and "die").

Definition. A *nondeterministic finite automata (NFA)* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, like its DFA counterpart, with the one exception that the transition function $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ maps a state and a symbol into the power set of Q.

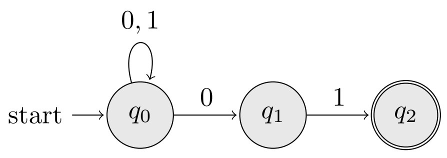


Figure 2.3: An NFA that accepts inputs ending in 01

	0	1
$\to q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

Table 2.4: The transition table for this NFA

2.2.1 Extending the Transition Function

Once again, we will recursively extend the transition function δ of an NFA from a function that operates on individual symbols to a function $\hat{\delta}$ that operates on whole strings. This time, however, it will be more complicated due to the nondeterminism. Let us start with an example to motivate the construction.

Example 2.4. We shall run the string 00101 through the NFA in figure 2.3.

1. $\hat{\delta}(q_0, \epsilon) = \{q_0\}$
2. $\hat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$
3. $\hat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
4. $\hat{\delta}(q_0, 001) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$
5. $\hat{\delta}(q_0, 0010) = \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
6. $\hat{\delta}(q_0, 00101) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$.

□

The base case $\hat{\delta}(q, \epsilon) \coloneqq \{q\}$ remains virtually the same. The recursive case for $w = ua$ where a is a symbol and u is the initial slice of the string w and for any state q,

$$\hat{\delta}(q, w) = \hat{\delta}(q, ua) \coloneqq \bigcup_{p \in \hat{\delta}(q,u)} \delta(p, a).$$

In other words, suppose $\hat{\delta}(q, u) = \{p_1, \ldots, p_n\}$, then $\hat{\delta}(q, w) = \delta(p_1, a) \cup \cdots \cup \delta(p_n, a)$.

2.2.2 The Language of an NFA

Example 2.5. Building off the previous example using the NFA of figure 2.3, the language of this NFA should accept the string 00101 as it is a string that ends in 01. Since $\hat{\delta}(q_0, 00101) = \{q_0, q_2\}$, and q_2 is the only accepting state. For an arbitrary string w, the language of this NFA contains w if and only if $\hat{\delta}(q_0, w) \cap \{q_2\} \neq \emptyset$. □

Definition. Given an NFA $A = (Q, \Sigma, \delta, q_0, F)$, the *language of the NFA A* is $L(A)$ and is defined as

$$L(A) \coloneqq \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}.$$

For any language L, if $L = L(A)$, then we will also say that the *language is accepted by the NFA A*.

2.2.3 Equivalence of Deterministic and Nondeterministic Finite Automata

Example 2.6 (Subset construction)**.** We will start by using a technique called the "subset construction" on the NFA in figure 2.3 to build an equivalent DFA. In general, the goal of the subset construction is to build a DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ from a given NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ such that $L(D) = L(N)$. Here is how the construction goes:

- $Q_D \coloneqq \mathcal{P}(Q_N)$ is the set of subsets of Q_N. However, many of these states in Q_D will not be accessible/reachable from the initial state, and these can be discarded.
- The input alphabets are the same for the two automata.
- For each subset $S \subseteq Q_N$ and input symbol $a \in \Sigma$, we follow δ_N from every state in S taking the a transition:
$$\delta_D(S, a) \coloneqq \bigcup_{p \in S} \delta_N(p, a).$$
- The initial state for D is the singleton set containing only the initial state for N.
- $F_D \coloneqq \{S \subseteq Q_N \mid S \cap F_N \neq \emptyset\}$ is the collection of subsets of Q_N which contain one or more accepting states.

This produces the following transition table. The inaccessible/unreachable state sets can be eliminated from

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Table 2.5: Subset construction from the NFA in figure 2.3

the table. Of the eight state sets, only three are reachable from the initial state: $\{q_0\}$, $\{q_0, q_1\}$, $\{q_0, q_2\}$. $\square$

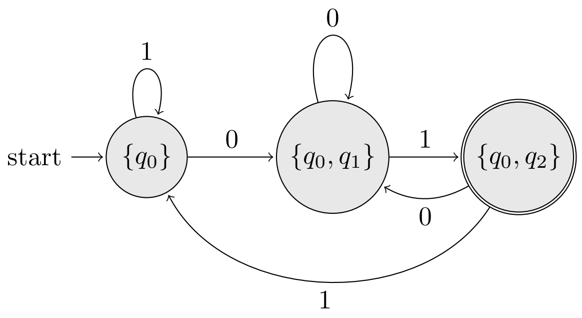


Figure 2.4: Pruned DFA constructed from the NFA in figure 2.3

Theorem 2.1. *Given an NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$, building the DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ using the above subset construction, then $L(D) = L(N)$.*

Proof. First, we will prove by induction on $|w|$ that

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w).$$

Base: For $|w| = 0$, it follows that $w = \epsilon$. By the definitions of $\hat{\delta}$., both $\hat{\delta}_D(\{q_0\}, \epsilon)$ and $\hat{\delta}_N(q_0, \epsilon)$ are $\{q_0\}$.
Ind($|w|$): Now assume that $|w| = n + 1$ and decompose $w = ua$ into some tailing symbol a and string u of length n such that $S \coloneqq \hat{\delta}_D(\{q_0\}, u) = \hat{\delta}_N(q_0, u)$.

$$\hat{\delta}_D(\{q_0\}, ua) = \bigcup_{p \in S} \delta_N(p, a) = \hat{\delta}_N(q_0, ua)$$

By induction, we have that $\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}(q_0, w)$ for all strings w. Now to prove that $L(D) = L(N)$, we will show that every element of one is an element of the other. Given $w \in L(D)$, we have by definition that $\hat{\delta}_D(\{q_0\}, w) \in F_D$. F_D recall is the set of all subsets of states that have some state belonging to F_N. In other words, $\hat{\delta}_D(\{q_0\}, w) \cap F_N \neq \emptyset$. And since we have shown that $\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$, we have that $\hat{\delta}_N(q_0, w) \cap F_N \neq \emptyset$. This is exactly the condition we need to say that $w \in L(N)$. Finally, we need to show that given $w \in L(N)$, then $w \in L(D)$. If $w \in L(N)$, then $\hat{\delta}_N(q_0, w) \cap F \neq \emptyset$. Consequently, $\hat{\delta}_N(q_0, w) = \hat{\delta}_D(\{q_0\}, w)$ is a set of states that shares an element with F_N, which is to say it is an element of F_D. With $\hat{\delta}_D(\{q_0\}, w) \in F_D$, we can now say that $w \in L(D)$. This completes the proof that $L(D) = L(N)$. □

Theorem 2.2. *A language L is accepted by some DFA if and only if it is accepted by some NFA.*

Proof. (If) Given a language L which is accepted by some NFA N, which is to say that $L = L(N)$, our previous theorem says that the subset construction will produce a DFA D such that $L = L(N) = L(D)$. Therefore, L is also accepted by D.

(Only if) Given a language L which is accepted by some DFA $D = (Q, \Sigma, \delta_D, q_0, F)$, we need to construct an equivalent NFA N. Basically, everything in D is going to serve the same role in N. So $N = (Q, \Sigma, \delta_N, q_0, F)$ is going to have the same states, the same initial state, and the same accepting states. The only thing that changes, albeit slightly, is $\delta_N(q, a) \coloneqq \{\delta_D(q, a)\}$. It is an easy matter to prove by induction on $|w|$ that $\hat{\delta}_N(q, w) = \{\hat{\delta}_D(q, w)\}$, which we will leave to the reader. Also left to the reader is to further show that any string accepted by D is also accepted by N and the converse is true too. This means $L = L(D) = L(N)$. □

What we have just shown is that DFAs and NFAs have equivalent power and their accepting languages are the regular languages.

Example 2.7 (Worse case for subset construction)**.** The following NFA which accepts only those strings with a 1 in the nth position.

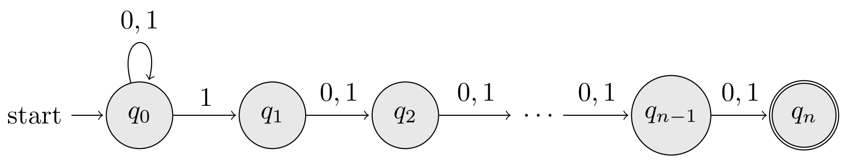


Figure 2.5: An NFA that has an equivalent DFA with no fewer than 2^n states

The equivalent DFA has to remember the last n symbols that it has read, which amounts to 2^n different states needed. □

2.3 Finite Automata with Epsilon-Transitions

An ϵ-transition is one which can be triggered without consuming any input. We shall focus on ϵ-NFAs which are NFAs that have been enhanced with ϵ-transitions.

Example 2.8. Consider matching on any floating point number in a programming language. There can be an optional $+/-$ sign, followed by a string of digits, followed by a ., and finally another string of digits. Note that one of the two strings of digits can be omitted without both being omitted together. In other words, we need to be able to accept strings such as $+3.14$, $.414$, and $-1.$ while rejecting such strings as . and 1.0.1.

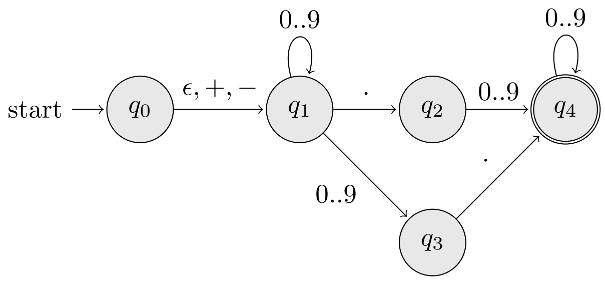


Figure 2.6: An ϵ-NFA to accept a float

□

2.3.1 Formal Notation for Epsilon-NFAs

Definition. An ϵ-NFA is a regular NFA with the transition function augmented to operate on a state and a symbol or ϵ, $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$. Note that ϵ cannot be a symbol in Σ.

Example 2.9. The transition table for the ϵ-NFA for figure 2.6 is the following. □

	$\epsilon, +, -$	.	0..9
$\to q_0$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\{q_2\}$	$\{q_1, q_3\}$
q_2	$\emptyset$	$\emptyset$	$\{q_4\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
$*q_4$	$\emptyset$	$\emptyset$	$\{q_4\}$

Table 2.6: Transition table for figure 2.6

2.3.2 Epsilon-Closures

The ϵ-closure of a state is the original state along with any state reachable from the original state via ϵ-transitions.

Definition. For an ϵ-NFA A with transition function δ and any state q of A, the *ϵ-closure of state q* is $\textsc{eclose}(q)$ is recursively defined with a base case of $q \in \textsc{eclose}(q)$ and a recursive case of $\delta(p, \epsilon) \subseteq \textsc{eclose}(q)$ for every $p \in \textsc{eclose}(q)$. The *ϵ-closure of a set of states S* is $\textsc{eclose}(S) \coloneqq \bigcup_{q \in S} \textsc{eclose}(q)$.

2.3.3 Extended Transitions and Languages for Epsilon-NFAs

The extended transition function $\hat{\delta}$ for an ϵ-NFA $E = (Q, \Sigma, \delta, q_0, F)$ is recursively defined as follows: the base case is $\hat{\delta}(q, \epsilon) \coloneqq \textsc{eclose}(q)$, and the recursive case for $w = ua$ where a is the last symbol of the given string w is

$$\hat{\delta}(q, w) = \hat{\delta}(q, ua) \coloneqq \textsc{eclose}\left(\bigcup_{p \in \hat{\delta}(q,u)} \delta(p, a) \right).$$

In other words, let $\hat{\delta}(q,u) \coloneqq \{p_1,\dots,p_m\}$ be all the states reachable from q using the input u taking any ϵ-transitions along the way, and let $\bigcup_{k=1}^{m} \delta(p_k,a) \coloneqq \{r_1,\dots,r_n\}$. Then $\hat{\delta}(q,w) = \text{ECLOSE}(\{r_1,\dots,r_n\}$.

Example 2.10. We will compute $\hat{\delta}(q_0, 3.14)$ for the ϵ-NFA of figure 2.6.

1. $\hat{\delta}(q_0,\epsilon) = \{q_0,q_1\}$
2. $\hat{\delta}(q_0,3) = \text{ECLOSE}\,(\delta(q_0,3) \cup \delta(q_1,3)) = \text{ECLOSE}(\emptyset \cup \{q_1,q_3\}) = \text{ECLOSE}(q_1) \cup \text{ECLOSE}(q_3) = \{q_1,q_3\}$
3. $\hat{\delta}(q_0,3.) = \text{ECLOSE}\,(\delta(q_1,.) \cup \delta(q_3,.)) = \text{ECLOSE}(\{q_2\} \cup \{q_4\}) = \text{ECLOSE}(q_2) \cup \text{ECLOSE}(q_4) = \{q_2,q_4\}$
4. $\hat{\delta}(q_0,3.1) = \text{ECLOSE}\,(\delta(q_2,1) \cup \delta(q_4,1)) = \text{ECLOSE}(\{q_4\}) = \{q_4\}$
5. $\hat{\delta}(q_0,3.14) = \text{ECLOSE}\,(\delta(q_4,4)) = \text{ECLOSE}(\{q_4\}) = \{q_4\}$

□

Definition. The *language accepted by an ϵ-NFA $A = (Q,\Sigma,\delta,q_0,F)$* is

$$L(A) \coloneqq \{w \in \Sigma^* \mid \hat{\delta}(q_0,w) \cap F \neq \emptyset\}.$$

2.3.4 Eliminating Epsilon-Transitions

Given an ϵ-NFA $E = (Q_E,\Sigma,\delta_E,q_0,F_E)$, we want to construct an equivalent DFA $D = (Q_D,\Sigma,\delta_D,q_D,F_D)$.

1. $Q_D \coloneqq \mathcal{P}(Q_E)$ is the set of subsets of Q_E. In fact, the reachable states in Q_D are those subsets of Q_E which are ϵ-closed, which is to say all subsets $S \subseteq Q_E$ such that $\text{ECLOSE}(S) = S$.
2. δ_D is the following computation:

   $$\delta_D(S,a) \coloneqq \text{ECLOSE}\left(\bigcup_{p\in S} \delta_E(p,a)\right).$$

   Said a different way, suppose $S \coloneqq \{p_1,\dots,p_m\}$. First, compute $\bigcup_{k=1}^{m} \delta_E(p_k,a)$ to some set $\{r_1,\dots,r_n\}$. Then $\delta_D(S,a)$ is set to $\text{ECLOSE}(\{r_1,\dots,r_n\})$.
3. $q_D \coloneqq \text{ECLOSE}(q_0)$ is the ϵ-closure of the only starting state of E.
4. $F_D \coloneqq \{S \in Q_D \mid S \cap F_E \neq \emptyset\}$ is the collection of states in D, which are subsets of states in Q_E, that contain one or more accepting state in F_E.

Example 2.11. Let us convert the floating point ϵ-NFA from figure 2.6 into an equivalent DFA. First note, that we will not include any dead states in figure 2.7 below in order to enhance clarity and readability. As the ϵ-closure of the start state of the ϵ-NFA is $\{q_0,q_1\}$, this is the start state of the equivalent DFA. From this state, we can either see a digit on the input and go to the state $\{q_1,q_3\}$; a decimal point and go to the state $\{q_2\}$; or a $+,-$ and go to the state $\{q_1\}$. From the state $\{q_1\}$, we can go to $\{q_1,q_3$ on a digit or to $\{q_2\}$ on a period. At $\{q_1,q_3\}$, we loop on a digit and go to $\{q_2,q_4\}$ on a dot. Both states $\{q_2\}$ and $\{q_2,q_4\}$ go to $\{q_4\}$ on a digit. Finally, we loop on $\{q_4\}$ on any more digits.

□

Theorem 2.3. *A language L is accepted by some ϵ-NFA if and only if L is accepted by some DFA.*

Proof. (If) Given a DFA $D = (Q,\Sigma,\delta_D,q_0,F)$ such that $L(D) = L$, define the equivalent ϵ-NFA $E = (Q,\Sigma,\delta_E,q_0,F)$ as follows:

- $\delta_E(q,a) = \{\delta_D(q,a)\}$ for any state q and input symbol a

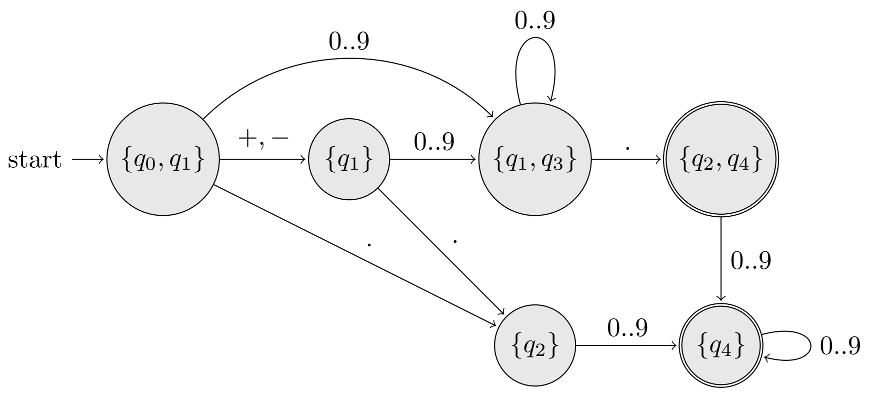


Figure 2.7: An equivalent DFA to accept a float

- $\delta_E(q, \epsilon) = \emptyset$ for any state q.

This way, we have explicitly stated that there are no ϵ-transitions in E and that all other transitions remain the same.

(Only if) Given an ϵ-NFA $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ such that $L(E) = L$, use the modified subset construction described at the start of this subsection to get the DFA $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$. We need to show that $L = L(E) = L(D)$ and will do so by showing that the two transition functions are the same, i.e. $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_D, w)$ for all strings w. Use induction on $|w|$.

Base: Suppose $|w| = 0$ and so $w = \epsilon$.

$$\hat{\delta}_E(q_0, \epsilon) = \text{ECLOSE}(q_0) = q_D = \hat{\delta}_D(q_D, \epsilon)$$

Ind($|w|$): Now, let $|w| = n + 1$ with $w = ua$ for some input symbol a and initial substring u of length n such that $\hat{\delta}_E(q_0, u) = \hat{\delta}_D(q_D, u)$.

$$\hat{\delta}_E(q_0, ua) = \text{ECLOSE}\left(\bigcup_{p \in \hat{\delta}_E(q_0, u)} \delta_E(p, a)\right) = \text{ECLOSE}\left(\bigcup_{p \in \hat{\delta}_D(q_D, u)} \delta_E(p, a)\right) = \hat{\delta}_D(q_D, ua)$$

More simply, suppose $\hat{\delta}_E(q_0, u) = \hat{\delta}_D(q_D, u) = \{p_1, \ldots, p_m\}$. By definition, $\hat{\delta}_E(q_0, w) = \hat{\delta}_E(q_0, ua)$ is the ϵ-closure of $\bigcup_{k=1}^{m} \delta_E(p_k, a)$. This is also the definition of $\hat{\delta}_D(q_D, w) = \hat{\delta}_D(q_D, ua)$.

Thus, we have proved by induction on $|w|$ that $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_D, w)$. □

Chapter 3

Regular Expressions

3.1 Definition

Definition. Given a language L, the *nth power of the language* L is recursively defined as $L^0 := \{\epsilon\}$ and $L^{n+1} := L^n \cdot L = \{uv \mid u \in L^n \wedge v \in L\}$. The *(Kleene) closure of* L is

$$L^* := \bigcup_{n=0}^{\infty} L^n = L^0 \cup L^1 \cup L^2 \cup \cdots .$$

In other words,

- $L^0 = \{\epsilon\}$
- $L^1 = L$
- $L^2 = L \cdot L = \{uv \mid u, v \in L\}$
- $L^3 = LLL = \{uvw \mid u, v, w \in L\}$
- L^n is the language of n words from L concatenated together.
- L^* is the language of zero or more words from L concatenated together.

We will go from a computational description of regular languages to an algebraic one. To do this, we recursively define regular expressions.

Definition. A regular expression E and the language it represents $L(E)$ is defined recursively as follows.

Base cases:

1. Constant regular expressions ϵ, with $L(\epsilon) = \{\epsilon\}$, and $\emptyset$, with $L(\emptyset) = \emptyset$.
2. If a is any symbol, then $\mathbf{a}$ is a regular expression with associated language $L(\mathbf{a}) = \{a\}$.

Recursive cases: (in order of lowest precedence to highest)

1. (Choice/union) If E and F are regular expressions, then $(E+F)$ is a regular expression whose language is $L(E+F) := L(E) \cup L(F)$.
2. (Concatenation) If E and F are regular expressions, then (EF) is a regular expression whose language is $L(EF) := L(E) \cdot L(F) = \{uv \mid u \in L(E), v \in L(F)\}$.
3. (Closure/star/Kleene closure) If E is a regular expression, then (E^*) is a regular expression whose language is $L\left(E^*\right) := L(E)^*$.

Remark. The Kleene closure of the empty set is the language with only the empty string by the following reasoning.

$$\emptyset^* = \emptyset^0 \cup \emptyset^1 \cup \emptyset^2 \cup \cdots = \{\epsilon\} \cup \emptyset \cup \emptyset \cup \cdots = \{\epsilon\}$$

Example 3.1. The following are the regular expressions for some of the finite automata that we looked at in the previous chapter.

1. Figure 2.1 p3, the language of binary strings that contained the substring 01: $\mathbf{1}^*\mathbf{0}\mathbf{0}^*\mathbf{1}(\mathbf{0}+\mathbf{1})^*$.
2. Figure 2.2 p3, the language over $\{a, b\}$ of strings that started and ended in the same letter:

$$\mathbf{a}(\mathbf{a}+\mathbf{b})^*\mathbf{a} + \mathbf{b}(\mathbf{a}+\mathbf{b})^*\mathbf{b}.$$

3. Figure 2.3 p6, the language of binary strings that end with 01: $(\mathbf{0}+\mathbf{1})^*\mathbf{01}$.

□

3.2 Finite Automata and Regular Expressions

We aim to show that every regular language—a language accepted by some DFA/NFA/ϵ-NFA—is the language of some regular expression. We have already shown the equivalences between DFAs, NFAs, and ϵ-NFAs. Now we will show that every DFA can be converted into a regular expression, and every regular expression can be converted into an ϵ-NFA. This will complete four equivalent notions of regular languages.

$$\begin{tikzcd}
\text{DFA} \arrow[r] \arrow[d, leftrightarrow] \arrow[dr, leftrightarrow] & \text{RegEx} \arrow[d] \\
\text{NFA} & \epsilon\text{-NFA}
\end{tikzcd}$$

Figure 3.1: Plan for showing the equivalence of our four notions for regular languages

3.2.1 DFA to Regular Expression

We will use an example to demonstrate the mechanics of the proof of the this theorem.

Example 3.2. Using the DFA in figure 2.1 p3, we will convert this to a regular expression. First, we will rename the states to $q_0 \mapsto 1$, $q_1 \mapsto 2$, and $q_2 \mapsto 3$. We will be building up the following regular expressions R_{ij}^k recursively in k, where R_{ij}^k is the regular expression whose language is the set of strings w such that w is the label for a path from i to j and that path has no interior nodes greater than k (this restriction does not apply to the end nodes i and j). We start this by constructing R_{ij}^0 which will be the regular expressions to get from state i to state j without going through any other states (as no other states are numbered 0 or lower).

Now we will use the recursive rule

$$R_{ij}^{k+1} = R_{ij}^k + R_{i,k+1}^k \left(R_{k+1,k+1}^k\right)^* R_{k+1,j}^k$$

which goes from i to j avoiding all states numbered larger than $k+1$ by either: (1) going from i to j avoiding states larger than k or (2) going from i to $k+1$, looping at $k+1$ zero or more times, and finally going from $k+1$ to j while staying at states number below $k+1$ in the interim. Also, we will avoid listing the regular expressions which are empty to conserve space in the following three tables.

R^0_{11}	$\epsilon + \mathbf{1}$
R^0_{12}	$\mathbf{0}$
R^0_{13}	$\emptyset$
R^0_{21}	$\emptyset$
R^0_{22}	$\epsilon + \mathbf{0}$
R^0_{23}	$\mathbf{1}$
R^0_{31}	$\emptyset$
R^0_{32}	$\emptyset$
R^0_{33}	$\epsilon + \mathbf{0} + \mathbf{1}$

Table 3.1: R^0_{ij} table for figure 2.1

R^1_{11}	$\epsilon + \mathbf{1} + (\epsilon + \mathbf{1})(\epsilon + \mathbf{1})^*(\epsilon + \mathbf{1}) = \mathbf{1}^*$
R^1_{12}	$\mathbf{0} + (\epsilon + \mathbf{1})(\epsilon + \mathbf{1})^*\mathbf{0} = \mathbf{1}^*\mathbf{0}$
R^1_{22}	$(\epsilon + \mathbf{0}) + \emptyset(\epsilon + \mathbf{1})^*\mathbf{0} = \epsilon + \mathbf{0}$
R^1_{23}	$\mathbf{1} + \emptyset(\mathbf{1}^*)^*\emptyset = \mathbf{1}$
R^1_{33}	$(\epsilon + \mathbf{0} + \mathbf{1}) + \emptyset(\epsilon + \mathbf{1})^*\emptyset = \epsilon + \mathbf{0} + \mathbf{1}$

Table 3.2: R^1_{ij} table for figure 2.1

Since 1 was the initial state and 3 is the only terminal state, the regular expression for the DFA is

$$R^3_{13} = R^2_{13} + R^2_{13}\left(R^2_{33}\right)^* R^2_{33} = \mathbf{1}^*\mathbf{0}\mathbf{0}^*\mathbf{1} + \mathbf{1}^*\mathbf{0}\mathbf{0}^*\mathbf{1}(\epsilon + \mathbf{0} + \mathbf{1})^*(\epsilon + \mathbf{0} + \mathbf{1}) = \mathbf{1}^*\mathbf{0}\mathbf{0}^*\mathbf{1}(\mathbf{0} + \mathbf{1})^*$$

. □

Theorem 3.1. *If L is the language accepted by some DFA A, then there is a regular expression E such that $L(R) = L$.*

Proof. We will take the states to be $\{1, \ldots, n\}$, and begin to build the R^k_{ij} regular expressions.

Base: Let $k = 0$. The R^0_{ij} are the regular expressions for paths that have no intermediate states. If $i \neq j$, then there are two possibilities: (1) there are no transition from i to j and so $R^0_{ij} = \emptyset$ or (2) there is one or more transitions with labels $a_1, \ldots, a_n$ and $R^0_{ij} = \mathbf{a}_1 + \cdots + \mathbf{a}_n$. Otherwise $i = j$, and the previous two possibilities hold: (1) there are no loops at i and $R^0_{ii} = \epsilon$ or (2) there are more than one loops with labels $a_1, \ldots, a_n$ and $R^0_{ii} = \epsilon + \mathbf{a}_1 + \cdots + \mathbf{a}_n$.

Ind(k): We need to build all the R^{k+1}_{ij}s from the R^k_{ij}s. And we can either go from i to j while staying strictly below $k+1$ or we can go from i to $k+1$, loop at $k+1$ zero or more times, and then go from $k+1$ to j without going above k. This is the regular expression

$$R^{k+1}_{ij} \coloneqq R^k_{ij} + R^k_{i,k+1}\left(R^k_{k+1,k+1}\right)^* R^k_{k+1,j}.$$

With R^n_{ij} built for all i and j, we assume without loss of generality that 1 is the initial state. The regular expression for the language of the DFA is then the sum of all the expressions R^n_{1j} for all j which are accepting. □

3.2.2 Regular Expression to Epsilon-NFA

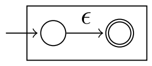
Theorem 3.2. *Ever language defined by a regular expression R is also defined by some finite automaton.*

R^2_{11}	$\mathbf{1}^*$
R^2_{12}	$\mathbf{1}^*\mathbf{0}$
R^2_{13}	$\mathbf{1}^*\mathbf{0}\mathbf{0}^*\mathbf{1}$
R^2_{22}	$\mathbf{0}^*$
R^2_{23}	$\mathbf{0}^*\mathbf{1}$
R^2_{33}	$\epsilon + \mathbf{0} + \mathbf{1}$

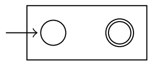
Table 3.3: R^2_{ij} table for figure 2.1

Proof. We will construct an ϵ-NFA E so that $L(E) = L(R)$ by structural induction on R. E will have exactly one accepting state, no arcs into the initial state, and no arcs out of the accepting state.

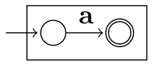
Base: We must have a base case for both constant regular expressions ϵ and $\emptyset$ as well as cases for the regular expressions for every symbol. These are shown in sub-figures 3.2a, 3.2b, and 3.2c respectively.



(a)



(b)



(c)

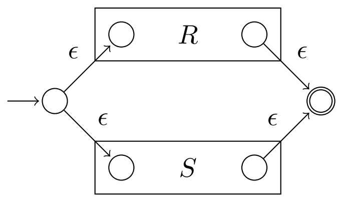
Figure 3.2: The 3 base cases of regular expressions to finite automata

Ind(R): We must have an inductive case for sums, concatenation, and for Kleene closure. For sums, we are given two regular expressions R and S and their equivalent ϵ-NFAs and construct the sum automaton as seen in figure 3.3a. Concatenation similarly has two regular expressions being given R and S which have equivalent automata, and the construction for RS is shown in figure 3.3b. To do Kleene closure, we assume that we have an arbitrary regular expression R and its equivalent automaton and construct the ϵ-NFA for R^* as shown in figure 3.3c.

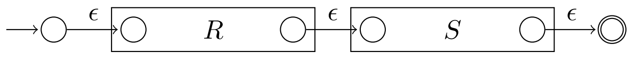
By structural induction on regular expressions, we have shown that there is an equivalent ϵ-NFA. □

3.3 Algebraic Laws of Regular Expressions

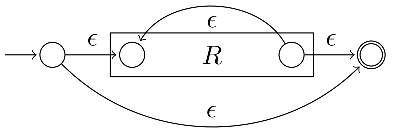
- Commutativity: $R + S = S + R$
- Associativity
  - $(R + S) + T = R + (S + T)$
  - $(RS)T = R(ST)$
- Identity
  - $\emptyset + R = R + \emptyset = R$
  - $\epsilon R = R\epsilon = R$
- Annihilator: $\emptyset R = R\emptyset = \emptyset$



(a)



(b)



(c)

Figure 3.3: The 3 inductive cases of regular expressions to finite automata

- Distributive
  - $R(S+T) = RS + RT$
  - $(R+S)T = RT + ST$
- Idempotent: $R + R = R$
- Closure laws
  - $(R^*)^* = R^*$
  - $\emptyset^* = \epsilon$
  - $\epsilon^* = \epsilon$

The following are syntactic sugar for our basic collection of regular expressions.

- One or more copies $R^+ \coloneqq RR^* = R^*R$
- Zero or one copies $R? \coloneqq R + \epsilon$
- Exactly n copies $R\{0\} \coloneqq \epsilon$ and $R\{n+1\} = RR\{n\}$.

3.3.1 Testing Identities

Example 3.3. We will show that $(R+S)^* = (R^*S^*)^*$. □

Proof. We will show that $L((R+S)^*) = L\left((R^*S^*)^*\right)$ by showing that a string in one is in the other as well. Given $w \in L((R+S)^*)$, then w is either empty or the concatenation of substrings $w = u_1u_2\cdots u_n$ with each $u_k \in L(R+S)$. If w is empty, then it is $w \in L\left((R^*S^*)^*\right)$. Now suppose $w = u_1 \cdots u_n$ where each $u_k \in L(R+S)$. Take u_1 for example, either $u_1 \in L(R)$ or $u_1 \in L(S)$. In either case, $u_1 \in L(R^*S^*)$. Similarly for each u_k, and w being is the concatenation of the u_k is in $L\left((R^*S^*)^*\right)$.

Given $w \in L\left((R^*S^*)^*\right)$, then w is either empty or $w = u_1 \cdots u_n$ where each $u_k \in L(R^*S^*)$. If w is empty, then $w \in L((R+S)^*)$. Otherwise, $w = u_1 \cdots u_n$ where each $u_k = r_{k1} \cdots r_{ki} s_{k1} \cdots s_{kj} \in L(R^*S^*)$. So each $u_k \in L((R+S)^*)$, and further $w \in L((R+S)^*)$. □

Chapter 4

Regular Languages

4.1 The Pumping Lemma

Example 4.1. We shall use the example of figure 2.1 on p3 to illustrate the mechanics of the proof for the pumping lemma. Recall that this is the language of strings that contains 01 as a substring. There are 3 states in the DFA, and so we will pick any string from this language with length 3 or more. Say $w \coloneqq 001$. We then define the following four states

$$\begin{aligned} p_0 &\coloneqq q_0 \\ p_1 &\coloneqq \hat{\delta}(q_0, 0) = q_1 \\ p_2 &\coloneqq \hat{\delta}(q_0, 00) = q_1 \\ p_3 &\coloneqq \hat{\delta}(q_0, 001) = q_2 \end{aligned}$$

By the pigeonhole principle, there has to be a duplication in this list of states. And, in fact, we can see that $p_1 = p_2 = q_1$. So we choose to split w into three parts: $x = 0$, $y = 0$, $z = 1$. Notice how $y \neq \epsilon$, $|xy| = 2 \leq 3$, and how every string of the form $xy^k z = 00^k 1$ is also in the language. □

Theorem 4.1 (Pumping lemma for regular languages)**.** *Let L be a regular language. There exists a constant n_L such that for every string $w \in L$ with $|w| \geq n_L$, we can break w into three parts $w = xyz$ such that (1) $y \neq \epsilon$, (2) $|xy| \leq n_L$, and (3) $xy^k z \in L$ for every $k \in \mathbf{N}$.*

Proof. Given a regular language L, there is a DFA A that accepts L. We shall use the pigeonhole principle on the states of A. Suppose A has n states. Given a string $w \in L$, suppose the length of w is at least as large as n, $|w| \geq n$. If there is no such w, then the theorem is vacuously true with $n_L \coloneqq n + 1$. Otherwise, there is a long enough string w in the language. Let $w = a_1 a_2 \cdots a_m$ where each a_k is an input symbol. For each $k = 0, 1, \ldots, n$, we will take the state $p_k \coloneqq \hat{\delta}(q_0, a_1 \cdots a_k)$ with $p_0 \coloneqq q_0$ where q_0 is the initial state of A.

We now have a list of $n + 1$ states $p_0, \ldots, p_n$ pulled from a collection of n states. By the pigeonhole principle, there exists a repeated state, i.e. there are constants i, j with $0 \leq i < j \leq n$ such that $p_i = p_j$. Now we can break apart w into three parts $x \coloneqq a_1 \cdots a_i$, $y \coloneqq a_{i+1} \cdots a_j$, and $z \coloneqq a_{j+1} \cdots a_{|w|}$. Note that z can be empty if $j = n = |w|$ or x can be empty if $i = 0$. But y can not be empty since $i < j$ and $|xy| = j \leq n$. And we are in the case show in the figure 4.1 below.

Consider the strings $xy^k z$ for any $k \in \mathbf{N}$. We have that $\hat{\delta}(p_0, x) = p_i$ and $\hat{\delta}(p_i, z)$ produces an accepting state. What we must show is that $\hat{\delta}(p_i, y^k) = p_i$ for all k. This is an easy argument by induction on k. Thus, the language L also accepts strings of the form $xy^k z$. □

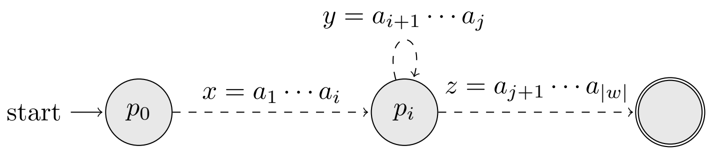


Figure 4.1: Every string longer than the number of states must contain a loop

4.1.1 Applying the Pumping Lemma

We will use the pumping lemma to show that languages are *not* regular. To do so, we will use the following two-player scheme. We pretend that we are playing against a cunning opponent.

1. We pick the language L which we believe is not regular.
2. The opponent chooses a value for n_L.
3. We choose the word w whose length is at least n_L.
4. The opponent gets to split w into three parts x, y, and z, but we know that $y \neq \epsilon$ and $|xy| \leq n_L$.
5. Finally, we must show that some string of the form $xy^kz \notin L$ for some $k \in \mathbf{N}$.

Example 4.2. We aim to show that the language $L \coloneqq \{0^n1^n \mid n \in \mathbf{N}\}$ is not regular. Given any value for n, we will choose $w = 0^n1^n$. For any split of $w = xyz$ with $y \neq \epsilon$ and $|xy| \leq n$, we know that y must be one or more zeros. Let $y = 0^k$ with $k \geq 1$. Then $xz = xy^0z = 0^{n-k}1^n \notin L$. Therefore, by the pumping lemma, L cannot be regular. □

Example 4.3. Here we will show that the language $L \coloneqq \{0^n \mid n \text{ is prime}\}$ is not regular. Given any value for n, we choose $w = 0^p$ where p is the smallest prime at least as large as $n+2$. For any splitting of $w = xyz$ with $y \neq \epsilon$ and $|xy| \leq p$. Define $m \coloneqq |y|$ and so $p - m = |xz|$, and note $m > 0$. The string $xy^{p-m}z$ is a string of $p - m + (p-m)m = (1+m)(p-m)$ zeros. However, the length of this string appears not to be prime. We still need to check that none of the factors are one though. Since $y \neq \epsilon$, $m > 0$ and so $1 + m > 1$. We chose $p \geq n+2$ and we have that $m = |y| \leq |xy| \leq p \leq n+2$. Hence, $p \geq n+2 \geq m+2$ and further $p - m \geq 2$. Thus, $xy^{p-m}z \notin L$, and L cannot be regular due to the pumping lemma. □

4.2 Closure Properties of Regular Languages

4.2.1 Closure under Union, Concatenation, and Kleene Star

Theorem 4.2. *If L and M are regular languages over an alphabet Σ, then so is $L \cup M$.*

Proof. Let R and S be regular expressions which accept L and M respectively. Then $L \cup M = L(R+S)$. □

As union is a primitive operation in our regular expressions, it is a simple matter to show that regular languages are closed under union, likewise for closure under concatenation and Kleene star.

Corollary 4.3. *If L and M are regular languages over the same alphabet, then so are $L \cdot M$ and L^*.*

4.2.2 Closure under Complement, Intersection

Theorem 4.4. *If L is a regular language, then so is its complement L^C.*

Proof. Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA that accepts the regular language L. Then $D' = (Q, \Sigma, \delta, q_0, Q \backslash F)$ is a similar DFA just with all accepting states made into non-accepting states and vice versa. Then $w \in L(D')$ if and only if $\hat{\delta}(q_0, w) \in Q \setminus F$, which occurs if and only if $w \notin L(D)$. Therefore, $L^C = L(D')$ is regular. □

Corollary 4.5. *If L and M are regular languages over the same alphabet, then so are $L \cap M$ and $L \setminus M$.*

Proof. By DeMorgan's laws, we can apply the previous two theorems repeatedly.

$$L \cap M = \left(L^C \cup M^C\right)^C$$

And note that $L \setminus M = L \cap M^C$. □

4.2.3 Closure under Reversal

Definition. The *reversal* of a string $a_1a_2\cdots a_n$ is the string $a_n \cdots a_2a_1$. We denote the reversal of string w as w^R. And the reversal of a language L is L^R which is the language whose strings are the reverse of strings in L.

Theorem 4.6. *If L is a regular language, then so is L^R.*

Proof. We will use structural induction on the corresponding regular expression E where $L = L(E)$. This to say that we will construct a reversed regular expression E^R such that $L\left(E^R\right) = L(E)^R$.

Base cases: If E is ϵ, $\emptyset$, or $\mathbf{a}$, then $E^R := E$.

Ind(E): There are three inductive cases:

1. If $E = E_1 + E_2$, then $E^R = E_1^R + E_2^R$. We have by the induction hypothesis that $L\left(E_k^R\right) = L(E_k)^R$ for $k = 1, 2$.
$$\begin{aligned} L\left(E^R\right) &= L\left(E_1^R + E_2^R\right) = L\left(E_1^R\right) \cup L\left(E_2^R\right) = L(E_1)^R \cup L(E_2)^R \\ &= (L(E_1) \cup L(E_2))^R = L(E_1 + E_2)^R = L(E)^R \end{aligned}$$

2. If $E = E_1E_2$, then $E^R = E_2^R E_1^R$.
$$L\left(E^R\right) = L\left(E_2^R E_1^R\right) = L\left(E_2^R\right) L\left(E_1^R\right) = L(E_2)^R L(E_1)^R = (L(E_1)L(E_2))^R = L(E_1E_2)^R = L(E)^R$$

3. If $E = E_1^*$, then $E^R = \left(E_1^R\right)^*$.
$$L\left(E^R\right) = \bigcup_{n=0}^{\infty} L\left(E_1^R\right)^n = \bigcup_{n=0}^{\infty} \left(L(E_1)^R\right)^n = \bigcup_{n=0}^{\infty} \left(L(E_1)^n\right)^R = \left(\bigcup_{n=0}^{\infty} L(E_1)^n\right)^R = L(E)^R$$

We have just shown how to construct E^R the reversal of a regular expression E such that $L\left(E^R\right) = L(E)^R$. □

4.2.4 Closure under Homomorphism

Definition. For alphabets Σ and T, a function $h : \Sigma \to T^*$ induces a *homomorphism on strings* $h : \Sigma^* \to T^*$ defined as $h(w) := h(a_1)h(a_2)\cdots h(a_n)$ for any string $w = a_1a_2\cdots a_n$ over Σ. Note that we reuse the name h for both functions.

Example 4.4. With $h(0) := ba$ and $h(1) = \epsilon$, we have that $h(0011) = h(0)h(0)h(1)h(1) = baba\epsilon\epsilon = baba$. □

Theorem 4.7. *If L is a regular language over Σ, and h is a string homomorphism from Σ, then $h(L) := \{h(w) \mid w \in L\}$ is also a regular language.*

Proof. Let $L = L(R)$ for some regular expression R. We will construct a homomorphism of regular expressions and show that $h(R)$ is a regular expression whose corresponding language is $h(L(R))$.

Base cases: If R is either ϵ or $\emptyset$, then the homomorphism h will not touch these regular expressions. In other words, $L(h(\epsilon)) = L(\epsilon) = \{\epsilon\}$ and $L(h(\emptyset)) = L(\emptyset) = \emptyset$. Otherwise, $R = \mathbf{a}$, and $h(R)$ is the regular expression whose language is single string $h(a)$. This is to say that $L(h(R)) = \{h(a)\} = h(L(R))$.

Ind(R):

1. If $R = R_1 + R_2$, then $h(R) = h(R_1) + h(R_2)$. The induction hypothesis is $L(h(R_k)) = h(L(R_k))$ for $k = 1, 2$.

$$\begin{aligned} L(h(R)) &= L(h(R_1) + h(R_2)) = L(h(R_1)) \cup L(h(R_2)) = h(L(R_1)) \cup h(L(R_2)) = h(L(R_1) \cup L(R_2)) \\ &= h(L(R_1 + R_2)) = h(L(R)) \end{aligned}$$

2. If $R = R_1R_2$, then $h(R) = h(R_1)h(R_2)$.

$$\begin{aligned} L(h(R)) &= L(h(R_1)h(R_2)) = L(h(R_1))L(h(R_2)) = h(L(R_1))h(L(R_2)) \\ &= h(L(R_1)L(R_2)) = h(L(R_1)L(R_2)) = h(L(R_1R_2)) = h(L(R_1R_2)) \end{aligned}$$

3. If $R = R_1^*$, then $h(R) = h(R_1)^*$.

$$L(h(R)) = L\left(h(R_1)^*\right) = L(h(R_1))^* = h(L(R_1))^* = h\left(L(R_1)^*\right) = h\left(L\left(R_1^*\right)\right) = h(L(R))$$

We have constructed a homomorphism of regular expressions from a homomorphism on strings h such that $L(h(R)) = h(L(R))$. □

Theorem 4.8. *Let h be a string homomorphism from alphabet Σ to T. If L is a regular language over T, then $h^{-1}(L)$ is also regular over Σ.*

Proof. Let $L = L(A)$ for some DFA $A = (Q, T, \delta, q_0, F)$. Define the DFA $B = (Q, \Sigma, \gamma, q_0, F)$ as $\gamma(q, a) = \hat{\delta}(q, h(a))$. In order to prove that $L(B) = h^{-1}(L)$, we will show that B accepts w if and only if A accepts $h(w)$. This will be done by showing that $\hat{\gamma}(q_0, w) = \hat{\delta}(q_0, h(w))$ by induction on $|w|$.

Base case: $|w| = 0$ and so $w = \epsilon$. $\hat{\gamma}(q_0, \epsilon) = q_0 = \hat{\delta}(q_0, \epsilon) = \hat{\delta}(q_0, h(\epsilon))$ because $h(\epsilon) = \epsilon$.

Ind($|w|$): Suppose $|w| = n + 1$, so $w = ua$ with $|u| = n$ and $\hat{\gamma}(q_0, u) = \hat{\delta}(q_0, h(u))$.

$$\hat{\gamma}(q_0, ua) = \gamma(\hat{\gamma}(q_0, u), a) = \gamma(\hat{\delta}(q_0, h(u)), a) = \hat{\delta}(\hat{\delta}(q_0, h(u)), h(a)) = \hat{\delta}(q_0, h(u)h(a)) = \hat{\delta}(q_0, h(ua))$$

□

Chapter 5

Context-Free Grammars

5.1 Definition

5.1.1 Informal Examples

Example 5.1. Below we give a context-free grammar for palindromes G_{pal}—strings w which satisfy the equation $w^R = w$.

$$
\begin{aligned}
&P \to \epsilon \\
&P \to 0 \\
&P \to 1 \\
&P \to 0P0 \\
&P \to 1P1
\end{aligned}
$$

The first three production rules are the base cases: ϵ, 0, and 1 are all palindromes. The last two rules are inductive cases: given a string w produced by these five rules, then $0w0$ and $1w1$ are palindromes as well. □

Example 5.2. The following context-free grammar, G_{exp}, is one for simple binary arithmetic expressions.

$$
\begin{aligned}
&E \to E + E \\
&E \to E * E \\
&E \to (E) \\
&E \to N \\
&N \to 0 \\
&N \to 1 \\
&N \to 0N \\
&N \to 1N
\end{aligned}
$$

The first three E-rules create additions, multiplications, and groupings. The fourth E-rule states that the base expressions is a binary number which can be produced by the N-rules.

□

5.1.2 Formal Definition

Definition. A *context-free grammar (CFG) G*, or just *grammar*, is a 4-tuple $G = (V, T, P, S)$ where

- V is the finite, nonempty set of *variables*, *nonterminals*, or *syntactic categories*.
- T is the finite set of *terminals* or *terminal symbols*.
- $P \subseteq V \times (V \cup T)^*$ is the finite set of *productions* or *rules*. Each production consists of a *head*, the first component, which is a variable being (partially) defined by the production, the production symbol $\rightarrow$, and finally a *body*, the second component, which is a string of zero or more terminals and nonterminals.
- $S \in V$ is the *start symbol*.

Definition. Let $G = (V, T, P, S)$ be a CFG with a production $A \rightarrow \gamma$. Then for any $\alpha, \beta \in (V \cup T)^*$, a relation called a *derivation* over strings of terminals and nonterminals, denoted by $\underset{G}{\Rightarrow}$ or just $\Rightarrow$ when G is understood, is defined as $\alpha A \beta \underset{G}{\Rightarrow} \alpha\gamma\beta$. We recursively define its transitive, reflexive closure $\underset{G}{\Rightarrow}^*$, or just $\Rightarrow^*$: the base case is that $\alpha \underset{G}{\Rightarrow}^* \alpha$ for any string α over terminals and nonterminals, and the recursive case is that $\alpha \underset{G}{\Rightarrow}^* \gamma$ whenever $\alpha \underset{G}{\Rightarrow}^* \beta$ and $\beta \underset{G}{\Rightarrow} \gamma$ for some string β.

In order to restrict the amount of choices there are in deriving strings, we define *leftmost derivation*, $\underset{lm}{\Rightarrow}$ and $\underset{lm}{\Rightarrow}^*$, which requires at each step the leftmost variable is replaced by one of its production bodies. Similarly, we define *rightmost derivation*, $\underset{rm}{\Rightarrow}$ and $\underset{rm}{\Rightarrow}^*$, which requires at each step the rightmost variable is replace by one of its production bodies.

5.1.3 Language of a Grammar

Definition. For any CFG $G = (V, T, P, S)$, the *language of* G, denoted $L(G)$, is the set of terminal strings which have derivations from the starting symbol.

$$L(G) \coloneqq \{w \in T^* \mid S \underset{G}{\Rightarrow}^* w\}$$

In fact, any string α over $(V \cup T)^*$ is called a *sentential form* if there is a derivation to it from the start symbol, i.e. $S \Rightarrow^* \alpha$. We will also say that α is a *left-sentential form* if $S \underset{lm}{\Rightarrow}^* \alpha$, and say that α is a *right-sentential form* if $S \underset{rm}{\Rightarrow}^* \alpha$.

Example 5.3. Consider the palindrome grammar G_{pal} on p21. We will show that $L(G_{pal})$ is the set of palindromes over $\{0, 1\}$.

Given a palindrome w, we will show that $w \in L(G_{pal})$ by induction on $|w|$.

Base: $|w|$ is 0 or 1, in which cases w is one of the three palindromes $\epsilon, 0, 1$. As we have a production for each of these strings, it follows that $P \Rightarrow^* w$.

Ind($|w|$): Suppose $|w| \geq 2$. Since w is a palindrome, it must begin and end with the same symbol. And so, $w = 0u0$ or $w = 1u1$ for some shorter palindrome u. By the inductive hypothesis, it follows that $P \Rightarrow^* u$. If $w = 0u0$, then

$$P \Rightarrow 0P0 \Rightarrow^* 0u0 = w.$$

Hence, $w \in L(G_{pal})$. A similar argument applies in the other case where $w = 1u1$.

Given a string $w \in L(G_{pal})$, we will show that w is a palindrome based upon induction on the number of steps in the derivation.

Base: If the derivation is one step, then either $P \Rightarrow \epsilon$, $P \Rightarrow 0$, or $P \Rightarrow 1$. And all three of these strings are palindromes.

Ind: Now suppose the derivation of w takes $n + 1$ steps. The induction hypothesis is that all strings which can be derived in at most n steps is a palindrome. The $(n + 1)$-step derivation of w must have one of the two forms $P \Rightarrow 0P0 \Rightarrow^* 0u0 = w$ or $P \Rightarrow 1P1 \Rightarrow^* 1u1 = w$, where $P \Rightarrow^* u$ is an n-step derivation. Therefore, u is a palindrome by the induction hypothesis. And so, w, being either $0u0$ or $1u1$, is another palindrome. □

$$\begin{tikzcd}
\text{Leftmost derivation} \arrow[d] & \text{Parse tree} \arrow[l] \arrow[d] \\
\text{Derivation} \arrow[ur] & \text{Rightmost derivation} \arrow[l]
\end{tikzcd}$$

Figure 5.1: Proving the equivalence of statements about grammars

5.2 Parse Trees

Definition. For any grammar $G = (V, T, P, S)$, the *parse trees* for G are trees defined as follows:

1. Each interior node is labeled by a variable in V.
2. Each leaf is labeled by either a variable, a terminal, or ϵ. If the leaf is labeled ϵ, then it can have no other siblings.
3. If an interior node is labeled A and its children are labeled $X_1, X_2, \ldots, X_n$ left to right, then $A \to X_1 X_2 \cdots X_n$ must be a production in P.

The *yield* of a parse tree is the concatenation of the leaves starting from the left.

Theorem 5.1. *Let $G = (V, T, P, S)$ be a CFG. If there is a parse tree with root label A and yield $w \in T^*$, then there is a leftmost derivation $A \underset{lm}{\Rightarrow}^* w$ in G. Further, there is a rightmost derivation $A \underset{rm}{\Rightarrow}^* w$ in G.*

Proof. We perform induction on the height of the parse tree.

Base: The height is one, which means the tree has a root labeled with some variable A and whose children are all leaves of terminals that yield w. Because this is a parse tree, $A \to w$ must be a production. Consequently, $A \underset{lm}{\Rightarrow} w$ is a one-step, leftmost derivation.

Ind: The height of the parse tree is $n + 1$. There is a root labeled by some variable A with children labeled $X_1, X_2, \ldots, X_n$ from left to right.

1. If X_k is a terminal, define $w_k := X_k$.
2. If X_k is a nonterminal, it must be the root of a subtree with a yield that we will call w_k. As the height of this tree is at most n, our inductive hypothesis applies, and so there is a leftmost derivation $X_k \underset{lm}{\Rightarrow}^* w_k$.

Note the yield of the parse tree at A is $w = w_1 w_2 \cdots w_n$.

We will construct a leftmost derivation of w from A as follows: start with the single step $A \underset{lm}{\Rightarrow} X_1 X_2 \cdots X_n$. For each, $k = 1, 2, \ldots, n$, we will show that

$$A \underset{lm}{\Rightarrow}^* w_1 w_2 \cdots w_k X_{k+1} X_{k+2} \cdots X_n.$$

This will actually be another induction, this time on k. The base case is that $k = 0$, but we already know that $A \underset{lm}{\Rightarrow} X_1 X_2 \cdots X_n$. For the inductive case, assume that we have shown that

$$A \underset{lm}{\Rightarrow}^* w_1 \cdots w_k X_{k+1} \cdots X_n.$$

If X_{k+1} is a terminal, then there is nothing to do except set $w_{k+1} := X_{k+1}$. Otherwise, X_{k+1} is a nonterminal, and we have assumed earlier that $X_{k+1} \underset{lm}{\Rightarrow^*} w_{k+1}$. Thus, we have the following.

$$A \underset{lm}{\Rightarrow^*} w_1 \cdots w_k X_{k+1} X_{k+2} \cdots X_n \underset{lm}{\Rightarrow^*} w_1 \cdots w_k w_{k+1} X_{k+2} \cdots X_n$$

Going up to $k = n$, we have constructed a leftmost derivation of w from A.

The proof of the rightmost derivation is symmetrical to the leftmost. □

Theorem 5.2. *Let $G = (V, T, P, S)$ be a context-free grammar. If there is a derivation $A \Rightarrow^* \gamma$ of G, then there is a parse tree for G whose root is A and whose yield is γ.*

Proof. The proof will be an induction based upon the length of the derivation $A \Rightarrow^* \gamma$.

Base: One step, $A \Rightarrow \gamma$. Then there is a production $A \to \gamma$, and so a parse tree of height 1 whose root is A and whose children are γ as read from the left.

Ind: The derivation takes $n+1$ steps, and so looks like $A \Rightarrow X_1 \cdots X_n \Rightarrow^* \gamma$ where each $X_k \in V \cup T$. We can split $\gamma = \gamma_1 \cdots \gamma_n$ so that $X_k \Rightarrow^* \gamma_k$ for each k. By the induction hypothesis, any X_k which is a nonterminal has a parse tree t_k whose root is X_k and whose yield is γ_k. There must be a production $A \to X_1 \cdots X_n$. So there is a parse tree whose root is A and yield is $X_1 \cdots X_n$. Extend this parse tree by replacing each X_k which is a nonterminal with the tree t_k. This forms a parse tree for the derivation $A \Rightarrow^* \gamma$ whose root is A and whose yield is $\gamma_1 \cdots \gamma_n = \gamma$. □

Corollary 5.3. *For a CFG $G = (V, T, P, S)$, let A be a nonterminal and w be a string of terminals. The following are equivalent:*

1. *$A \Rightarrow^* w$ is a derivation in G*
2. *$A \underset{lm}{\Rightarrow^*} w$*
3. *$A \underset{rm}{\Rightarrow^*} w$*
4. *There is a parse tree whose root is A and whose yield is w.*

5.3 Ambiguity in Grammars and Languages

Definition. Say a CFG $G = (V, T, P, S)$ is *ambiguous* if there is at least one string $w \in T^*$ for which there are two or more distinct parse trees whose roots are S and yields are w. A grammar is *unambiguous* if it is not ambiguous, which is to say that every string in the language of the grammar has exactly one parse tree with root S.

Example 5.4. The binary expression grammar on page 21 is actually ambiguous. Note that there is no notion of precedence in our context-free grammars. This is our source of ambiguity. The string $0 * 1 + 1$ has two parse trees:

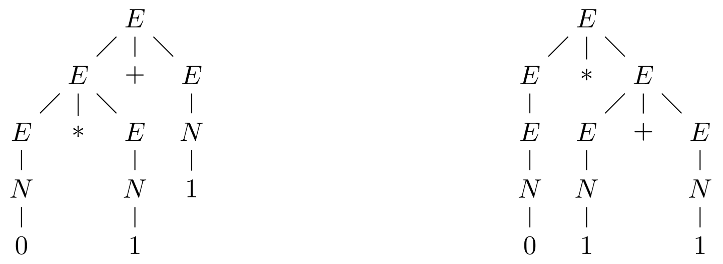


Figure 5.2: Demonstrating the expression grammar is ambiguous

□

Theorem 5.4. *For any grammar $G = (V, T, P, S)$ and string $w \in T^*$, w has two distinct parse trees if and only if it has two distinct leftmost derivations from S.*

Proof. (If) We will sketch an outline of how to construct a parse tree from a leftmost derivation. Start constructing the tree at the root, labeled S. Inspecting the derivation one step at a time, note that a variable will be replaced corresponding to the leftmost variable node in the tree which is currently a leaf. For the production used at this step, add symbols of the production body as children to this former leaf node. When there are two differing derivations, the nodes being constructed will get different children which will guarantee the parse trees produced will also be different.

(Only if) Notice in the proof of theorem 5.1 that wherever the parse trees have a node at which different productions are used, then the leftmost derivations constructed will also use different productions. Thus the derivations will differ as well. □

5.3.1 Inherent Ambiguity

Definition. A context-free language L is *inherently ambiguous* if all its grammars are ambiguous.

Example 5.5. Consider the language

$$L = \{a^n b^n c^m d^m \mid m, n \geq 1\} \cup \{a^n b^m c^m d^n \mid m, n \geq 1\}.$$

So L is a subset of the language $\mathbf{a}^+\mathbf{b}^+\mathbf{c}^+\mathbf{d}^+$ such that either:

1. There are as many *a*s as *b*s and as many *c*s as *d*s.
2. There are as many *a*s as *d*s and as many *b*s as *c*s.

This is a context-free language because the following grammar accepts it.

$$\begin{aligned}
S &\to HT \mid M \\
H &\to aHb \mid ab \\
T &\to cTd \mid cd \\
M &\to aMd \mid aBd \\
B &\to bBc \mid bc
\end{aligned}$$

And the string *aabbccdd* has two leftmost derivations. □

Example 5.6. Our binary expression language from p21 is not inherently ambiguous. The following is a non-ambiguous grammar.

$$\begin{aligned}
E &\to E + T \mid T \\
T &\to T * F \mid F \\
F &\to (E) \mid N \\
N &\to 0 \mid 1 \mid 0N \mid 1N
\end{aligned}$$

□

Chapter 6

Pushdown Automata

6.1 Definition

6.1.1 Informal Definition

Pushdown automata are going to be ϵ-NFAs augmented with a stack memory. In the diagram 6.1, each transition is labeled $a, A/\gamma$ where a is the current symbol on the input and A the current top stack symbol required to transition. The γ is the string of stack symbols that replace A on the top of the stack.

Example 6.1. The diagram is for a palindrome-like language L_{ww^R} is strings of the form ww^R. In other words, L_{ww^R} is the language of even length palindromes.

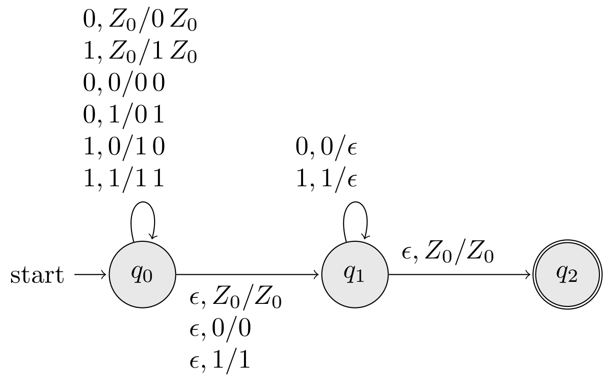


Figure 6.1: PDA for L_{ww^R} language

□

6.1.2 Formal Definition

Definition. A pushdown automata is a 7-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where

- Q is a finite, non-empty set of states.
- Σ is an alphabet of input symbols (which doesn't contain ϵ as an input symbol).
- Γ is an alphabet of stack symbols.

- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ is a transition function that takes a triple as input $\delta(q, a, X)$ where $q \in Q$ is the current state, a is either the current input symbol in Σ or the empty string ϵ, and $X \in \Gamma$ is the stack symbol at the top of the stack. The output of δ is a finite set of pairs (p, γ) where $p \in Q$ is the new state and $\gamma \in \Gamma^*$ is the string of stack symbols that replaces X on the top of the stack.

- $q_0 \in Q$ is the start state.

- $Z_0 \in \Gamma$ is the starting stack symbol. The stack starts with only one instance of this symbol.

- $F \subseteq Q$ is the set of accepting/final states.

An *instantaneous description (ID)* is a triple (q, w, γ) where $q \in Q$ is the current state, $w \in \Sigma^*$ is the remaining input, and $\gamma \in \Gamma^*$ is the stack contents. We define a transition relation $\underset{P}{\vdash}$, or just $\vdash$ when P is understood, as $(q, aw, X\beta) \vdash (p, w, \alpha\beta)$ if $\delta(q, a, X)$ contains (p, α) and $w \in \Sigma^*$ and $\beta \in \Gamma^*$. We extend this relation to its reflexive, transitive closure $\underset{P}{\vdash^*}$, or just $\vdash^*$ when P is understood: $I \vdash^* I$ for all IDs I is the base case, and the recursive case is if $I \vdash K$ and $K \vdash^* J$, then $I \vdash^* J$.

The following two theorems state that data which a PDA never looks at cannot affect its sequence of IDs (computation). The first theorem says that we can add data to the end of input and to the bottom of the stack in any valid computation.

Theorem 6.1. *If $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is a PDA and $(q, u, \alpha) \vdash^* (p, v, \beta)$ then for any strings $w \in \Sigma^*$ and $\gamma \in \Gamma^*$, we also have that*

$$(q, uw, \alpha\gamma) \vdash^* (p, vw, \beta\gamma).$$

Proof. The proof is done by induction on the number of steps it takes to get from (q, u, α) to (p, v, β).

Base: $(q, u, \alpha) \vdash^* (q, u, \alpha)$ takes 0 steps. And for any $w \in \Sigma^*$ and any $\gamma \in \Gamma^*$, $(q, uw, \alpha\gamma) \vdash^* (q, uw, \alpha\gamma)$.

Ind: Suppose the statement of the theorem holds for $(q, u, \alpha) \vdash^* (p, v, \beta)$ and that $(p, v, \beta) \vdash (p', v', \beta')$. For any $w \in \Sigma^*$ and any $\gamma \in \Gamma^*$, the induction hypothesis states that $(q, uw, \alpha\gamma) \vdash^* (p, vw, \beta\gamma)$. And by definition of $\vdash$, we have that $(p, vw, \beta\gamma) \vdash (p, v'w, \beta'\gamma)$. Thus, $(q, uw, \alpha\gamma) \vdash^* (p, v'w, \beta'\gamma)$.

Therefore, by structural induction on $\vdash^*$, we have that $(q, uw, \alpha\gamma) \vdash^* (p, vw, \beta\gamma)$ whenever $(q, u, \alpha) \vdash^* (p, v, \beta)$, $w \in \Sigma^*$, and $v \in \Gamma^*$. ☐

This second theorem says that if the some tail of the input is not consumed during computation, then it can be dropped. This is because a PDA can never consume some input and then later restore the same data back to the input. However, a PDA can pop a stack symbol and then later restore the stack to a previous configuration. So there is no analogous theorem for dropping stack symbols off the bottom of the input.

Theorem 6.2. *If $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is a PDA and $(q, uw, \alpha) \vdash^* (p, vw, \beta)$, then we also have that*

$$(q, u, \alpha) \vdash^* (p, v, \beta).$$

Proof. Again, we do an induction on the number of steps to go from $q, uw, \alpha)$ to $p, vw, \beta)$.

Base: Zero steps means $(q, uw, \alpha) \vdash^* (q, uw, \alpha)$, and so $(q, u, \alpha) \vdash^* (q, u, \alpha)$.

Ind: Suppose the statement of the theorem holds for $(q, uw, \alpha) \vdash^* (p, vw, \beta)$ and also suppose $(p, vw, \beta) \vdash (p', v'w, \beta')$. From the definition of $\vdash$, we have that $(p, v, \beta) \vdash (p', v', \beta')$ since the tail w is never seen/touched. And by the induction hypothesis, $(q, u, \alpha) \vdash^* (p, v, \beta)$. Hence, $(q, u, \alpha) \vdash^* (p', v', \beta)$.

By structural induction on $\vdash^*$, we have that $(q, u, \alpha) \vdash^* (p, v, \beta)$ whenever $(q, uw, \alpha) \vdash^* (p, vw, \beta)$. ☐

6.2 Languages of a PDA

6.2.1 Acceptance by Final State

Definition. Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. The *language accepted by P by final state* is

$$L(P) \coloneqq \{w \in \Sigma^* \mid \exists q \in Q, \alpha \in \Gamma^*.(q_0, Z_0, w) \vdash^* (q, \epsilon, \alpha) \wedge q \in F\}.$$

Example 6.2. We will show that the language accepted by diagram in figure 6.1 is indeed $L_{ww^R} = \{ww^R \mid w \in \{0,1\}^*\}$.

Given a binary string of the form ww^R, the PDA will consume the initial segment w symbol by symbol, placing them on the stack in reverse order. Then the PDA moves to the state q_1 on an ϵ-transition and leaves the stack alone. Next, the machine will consume the tail of the input w^R and at the same time pop the matching symbols on the stack, leaving us with an empty input and the starting symbol on the top of the stack. With a final ϵ-transition, the PDA will move into its accepting state q_2. This is formalized below.

$$(q_0, ww^R, Z_0) \vdash^* (q_0, w^R, w^R Z_0) \vdash (q_1, w^R, w^R Z_0) \vdash^* (q_1, \epsilon, Z_0) \vdash (q_2, \epsilon, Z_0)$$

Given a binary string u which is accepted by the PDA by final state, observe that the only way to enter the single accepting state q_2 is to be in q_1 with Z_0 on top of the stack. Thus, it is sufficient to show that $(q_0, u, Z_0) \vdash^* (q_1, \epsilon, Z_0)$ if u is of the form ww^R. We shall do this by induction on $|u|$ and prove the more general statement that

$$(q_0, u, \alpha) \vdash^* (q_1, \epsilon, \alpha)$$

for any stack string α.

Base: If $u = \epsilon$, then $u = \epsilon\epsilon^R$ and $(q_0, \epsilon, \alpha) \vdash^* (q_1, \epsilon, \alpha)$.

Ind: Suppose $u = a_1 \cdots a_n$ for some natural $n \geq 1$. There are two moves that the PDA can make from (q_0, u, α):

1. $(q_0, u, \alpha) \vdash (q_1, u, \alpha)$. From the state q_1, we can only pop the state with every input symbol consumed. Because $|u| > 0$, we must have that $(q_1, u, \alpha) \vdash^* (q_1, \epsilon, \beta)$ where β is some stack string which must be strictly shorter than α. Consequently, we cannot get $(q_0, u, \alpha) \vdash^* (q_1, \epsilon, \alpha)$ with this as our first move.

2. $(q_0, a_1 \cdots a_n, \alpha) \vdash^* (q_0, a_2 \cdots a_n, a_1\alpha)$. The only way a computation can end in (q_1, ϵ, α) is for the last move to consume a_n from the input and pop a_1 from the stack,

$$(q_0, a_2 \cdots a_n, a_1\alpha) \vdash^* (q_1, a_n, a_1\alpha) \vdash (q_1, \epsilon, \alpha)$$

. And this can only happen if $a_1 = a_n$. Using theorem 6.2, we can take a_n off the end of the input in the initial segment of the above sequence to get $(q_0, a_2 \cdots a_{n-1}, a_1\alpha) \vdash^* (q_1, \epsilon, a_1\alpha)$. By our induction hypothesis, it follows that $a_2 \cdots a_{n-1}$ is a string of the form ww^R for some w. Combining all of the information together, we see that

$$u = a_1 a_2 \cdots a_{n-1} a_n = a_1 ww^R a_1 = (a_1 w)(a_1 w)^R.$$

□

6.2.2 Acceptance by Empty Stack

Definition. For each PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, the *language accepted by P by empty stack* is

$$N(P) \coloneqq \{w \in \Sigma^* \mid \exists q \in Q. (q_0, w, Z_0) \vdash^* (q, \epsilon, \epsilon)\}.$$

Example 6.3. The binary language $\{0^n 1^n \mid n \in \mathbf{N}\}$, given by the diagram below accepts only by empty stack. □

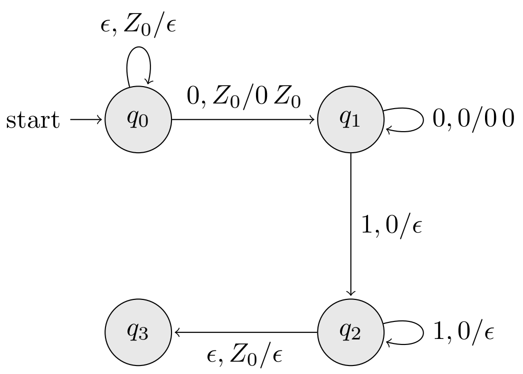


Figure 6.2: PDA for the $0^n 1^n$ language

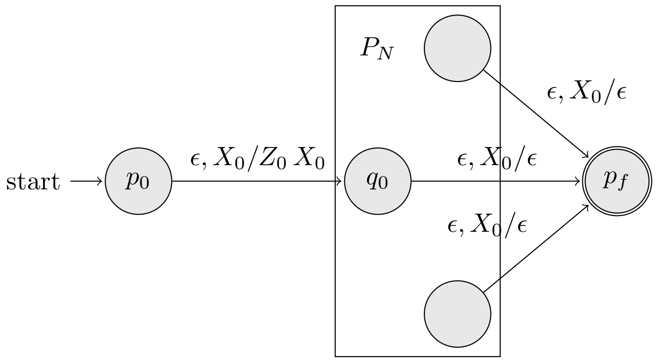


Figure 6.3: P_F simulating P_N and accepts if P_N empties its stack

6.2.3 From Empty Stack to Final State

Theorem 6.3. *For any PDA $P_N = (Q, \Sigma, \Gamma, q_0, Z_0, F)$, then there is a PDA P_F such that $N(P_N) = L(P_F)$.*

Proof. The definition of P_F is

$$P_F \coloneqq (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\}),$$

where p_0 and p_f are new states not in Q and X_0 is a new stack symbol not in Γ. The transition function δ_F is defined by the following conditions.

1. $\delta_F(p_0, \epsilon, X_0) \coloneqq \{(q_0, Z_0X_0)\}$. P_F immediately moves to start emulating P_N.
2. For all $q \in Q$, $a \in \Sigma \cup \{\epsilon\}$, and $Y \in \Gamma$, $\delta_F(q, a, Y) \coloneqq \delta_N(q, a, Y)$. This is where we state that P_F emulates P_N.
3. $\delta_F(q, \epsilon, X_0) \coloneqq \{(p_f, \epsilon)\}$ for all $q \in Q$. Here we terminate the emulation.
4. $\delta_F(p_f, \cdot, \cdot) \coloneqq \emptyset$. We cannot move once the emulation has stopped.

Given $w \in N(P_N)$, we have that $(q_0, w, Z_0) \underset{P_N}{\vdash^*} (q, \epsilon, \epsilon)$ for some $q \in Q$. By rule (2) above, we may say that $(q_0, w, Z_0X_0) \underset{P_F}{\vdash^*} (q, \epsilon, X_0)$. With theorem 6.1, we can append X_0 onto the bottom of the stack to get $(q_0, w, Z_0X_0) \underset{P_F}{\vdash^*} (q, \epsilon, X_0)$. Putting this together with the initial and final moves dictated by rules (1) and (3), we have that

$$(p_0, w, X_0) \underset{P_F}{\vdash} (q_0, w, Z_0X_0) \underset{P_F}{\vdash^*} (q, \epsilon, X_0) \underset{P_F}{\vdash} (p_f, \epsilon, \epsilon). \tag{6.1}$$

Therefore, P_F accepts w by final state.

Given $w \in L(P_F)$, we need to show that w is accepted by P_N by empty stack. The final step on input w in P_F has to have been rule (3), and we can only use that rule provided X_0 as the sole symbol on the stack. Also, X_0 cannot appear on the stack except as the bottommost symbol. Furthermore, rule (1) is the only transition which can be taken at the beginning, and so must be taken.

Hence, any computation of P_F on input w which accepts by final step must look like equation (6.1). The other, intermediate computations must be using rule (2), and so are computations of P_N with X_0 below the stack. Thus, it follows that $(q_0, w, Z_0) \underset{P_N}{\vdash^*} (q, \epsilon, \epsilon)$, which says that P_N accepts w by empty stack. $\square$

6.2.4 From Final State to Empty Stack

Theorem 6.4. *For any PDA $P_F = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, there is some PDA P_N such that $L(P_F) = N(P_N)$.*

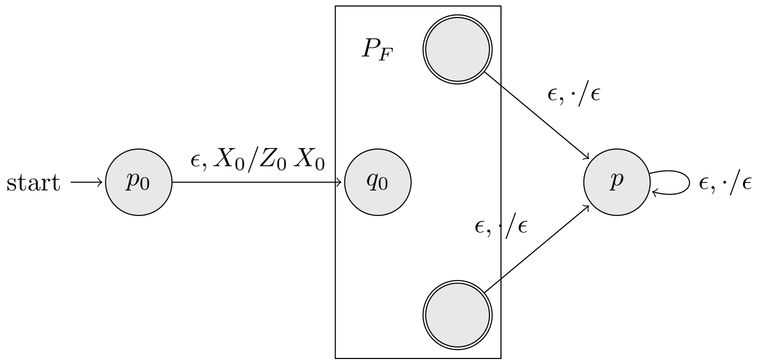

P_F
start
p_0
$\epsilon, X_0/Z_0\,X_0$
q_0
$\epsilon, \cdot/\epsilon$
p
$\epsilon, \cdot/\epsilon$
$\epsilon, \cdot/\epsilon$

Figure 6.4: P_N simulating P_F and empties its stack whenever P_F enters an accepting state

Proof. Construct P_N as suggested by figure 6.4, which is to say that

$$P_N \coloneqq (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0, \emptyset),$$

where p_0 and p are new states added to Q and X_0 is a new stack symbol added to Γ. The transition function δ_N is defined as follows.

1. $\delta_N(p_0, \epsilon, X_0) \coloneqq \{(q_0, Z_0X_0)\}$. P_N starts the emulation of P_F immediately.
2. For all states $q \in Q$, inputs $a \in \Sigma \cup \{\epsilon\}$, stack symbols $Y \in \Gamma$, $\delta_N(q, a, Y)$ contains all pairs in $\delta_F(q, a, Y)$. This ensures the emulation of P_F.
3. For all accepting states $q \in F$ and stack symbols $Y \in \Gamma \cup \{X_0\}$, $\delta_N(q, \epsilon, Y)$ contains (p, ϵ). Here the emulation of P_F ends.
4. For all stack symbols $Y \in \Gamma \cup \{X_0\}$, $\delta_N(p, \epsilon, Y) \coloneqq \{(p, \epsilon)\}$. We clean out the stack after emulation is over.

Given $w \in L(P_F)$, we must show that $w \in N(P_N)$. We know for some accepting state q and stack string α that $(q_0, w, Z_0) \underset{P_F}{\vdash^*} (q, \epsilon, \alpha)$. And since every transition of P_F is one in P_N, we can use theorem 6.1 to stick X_0 onto the bottom of the stack in order to get $(q_0, w, Z_0X_0) \underset{P_N}{\vdash^*} (q, \epsilon, \alpha X_0)$. The following moves can then be done entirely in P_N.

$$(p_0, w, X_0) \underset{P_N}{\vdash} (q_0, w, Z_0X_0) \underset{P_N}{\vdash^*} (q, \epsilon, \alpha X_0) \underset{P_N}{\vdash^*} (p, \epsilon, \epsilon)$$

The first move is thanks to rule (1), and the last set of transitions is due to rules (3) and (4). Therefore, w is accepted by P_N by empty stack.

Given $w \in N(P_N)$, we must show that $w \in L(P_F)$. The only way for P_N to have emptied its stack on w is to have entered state p, because X_0 is sitting at the bottom of any nonempty stack, and P_F cannot discharge X_0 from the stack. Further, the only way that P_N can get to state p is by entering an accepting state of P_F during its simulation. Lastly, the first move of P_N can only be given by rule (1). Therefore, every empty stack accepting computation of P_N looks like the following where q is an accepting state of P_F and α is any string over Γ.

$$(p_0, w, X_0) \underset{P_N}{\vdash} (q_0, w, Z_0X_0) \underset{P_N}{\vdash^*} (q, \epsilon, \alpha X_0) \underset{P_N}{\vdash^*} (p, \epsilon, \epsilon)$$

The IDs between (q_0, w, Z_0X_0) and $(q, \epsilon, \alpha X_0)$ are all transitions of P_F. And X_0 was never the top stack symbol between these two IDs, unless of course $\alpha = \epsilon$, in which case P_F emptied its stack exactly when it entered an accepting state. We may therefore conclude that the same computations can occur on P_F without X_0 on the stack, i.e. $(q_0, w, Z_0) \underset{P_F}{\vdash^*} (q, \epsilon, \alpha)$. Consequently, P_F accepts w by final state. $\square$

Chapter 7

Context-Free Languages

7.1 Equivalence of PDAs and CFGs

7.1.1 From Grammars to Pushdown Automata

Given a grammar $G = (V, T, P, S)$, we will construct the equivalent PDA P as follows

$$P \coloneqq (\{q\}, T, V \cup T, \delta, q, S, \emptyset)$$

where the transition function obeys the rules

1. For each nonterminal A,

$$\delta(q, \epsilon, A) \coloneqq \{(q, \beta) \mid A \to \beta \text{ is a production of } G\}$$

2. For each terminal a, $\delta(q, a, a) \coloneqq \{(q, \epsilon)\}$.

Example 7.1. We will convert the binary expression grammar in example 5.2 to a PDA. The input symbols are $0, 1, (,), +, *$. These 6 symbols along with E and N form the stack alphabet. The transition function is given by the following rules.

1. $\delta(q, \epsilon, N) = \{(q, 0), (q, 1), (q, 0N), (q, 1N)\}$
2. $\delta(q, \epsilon, E) = \{(q, N), (q, (E)), (q, E * E), (q, E + E)\}$
3. $\delta(q, 0, 0) = \delta(q, 1, 1) = \delta(q, (, () = \delta(q,),)) = \delta(q, +, +) = \delta(q, *, *) = \{(q, \epsilon)\}$

□

Theorem 7.1. *If PDA P is constructed from CFG G by the above construction, then $N(P) = L(G)$.*

Proof. Given $w \in L(G)$, then w has a leftmost derivation of the form

$$\gamma_1 \coloneqq S \underset{lm}{\Rightarrow} \gamma_2 \underset{lm}{\Rightarrow} \gamma_3 \underset{lm}{\Rightarrow} \cdots \underset{lm}{\Rightarrow} \gamma_n \coloneqq w.$$

In this proof, we will create a PDA that computes leftmost derivations in G. A leftmost sentential form can take the shape of $uA\alpha$ where $u \in T^*$ which is followed by a nonterminal A and a string of terminals and nonterminals α. We will call $A\alpha$ the *tail* of the leftmost sentential form. The only alternative is for a leftmost sentential form to consist of only terminals, in which case the tail is empty, i.e. ϵ.

We will show by induction on k that $(q, w, S) \underset{P}{\vdash^*} (q, v_k, \alpha_k)$ where α_k is the tail of γ_k, i.e. there is some string of terminals u_k such that $\gamma_k = u_k\alpha_k$, and v_k is such that $u_kv_k = w$.

Base: $n = 1$ and $\gamma_1 = S$. Then $v_1 = w$, $u_1 = \epsilon$, and $\alpha_1 = S$ since $(q, w, S) \vdash^* (q, w, S)$.

Ind: Assume for the sake of induction that $(q, w, S) \vdash^* (q, y_k, \alpha_k)$. Our goal is to show that $(q, w, S) \vdash^* (q, y_{k+1}, \alpha_{k+1})$ for some y_{k+1} and some α_{k+1}. As α_k is a tail, it must start with some nonterminal A. Otherwise, $\alpha_k = \epsilon$ and γ_k is just a string of terminals, which means we are done as $\gamma_k = w$. At the kth step of the derivation, $\gamma_k \Rightarrow \gamma_{k+1}$ involves replacing A with one of its production bodies, say β. Rule (1) allows us to replace A at the top of the stack with β, and rule (2) allows us to consume input that matches any and all terminals on top of the stack. Thus, we reach the ID $(q, v_{k+1}, \alpha_{k+1})$ which correctly represents the next sentential form γ_{k+1}.

At the end, γ_n will be w, leaving its tail α_n to be empty. Thus, $(q, w, S) \vdash^* (q, \epsilon, \epsilon)$, which means that P accepted w by an empty stack.

The other half of the proof will be more general. We shall show that if $(q, u, A) \underset{P}{\vdash^*} (q, \epsilon, \epsilon)$ then $A \underset{G}{\Rightarrow^*} u$. The proof is done by induction on the number of steps that P takes.

Base: One move. The only possibility is that $A \to \epsilon$ is a production of G. This means we are invoking transition from rule (1), and so $u = \epsilon$. And $A \Rightarrow \epsilon$ is true by definition.

Ind: Suppose P takes $n + 1$ moves. The first step has to be of type (1), where A is replaced by one of its production bodies, say $A \to Y_1 \cdots Y_k$ with each Y_j being either a terminal or a nonterminal. The next n moves must consume u from the input and pop all the Y_j off the stack. We can partition $u = u_1 \cdots u_k$ where each u_j is the portion of input consumed until Y_j is popped off the stack. In other words, $(q, u_j u_{j+1} \cdots u_k, Y_j) \vdash^* (q, u_{j+1} \cdots u_k, \epsilon)$ for each j, and they all happen in at most n steps. By theorem 6.2 and the induction hypothesis, we can conclude that $Y_j \Rightarrow^* u_j$ whenever Y_j is a nonterminal. Otherwise, Y_j is a terminal, u_j must be Y_j by rule (2), and we can conclude that $Y_j \Rightarrow^* u_j$ in one step. Finally, we have constructed the derivation

$$A \Rightarrow Y_1 Y_2 \cdots Y_k \Rightarrow^* u_1 Y_2 \cdots Y_k \Rightarrow^* \cdots \Rightarrow^* u_1 u_2 \cdots u_n = w.$$

To finish this part of the proof, we set $A := S$ and $u := w$. Given $w \in N(P)$, we have that $(q, w, S) \vdash^* (q, \epsilon, \epsilon)$. From what we have just shown, this means that $S \Rightarrow^* w$, which is the definition for $w \in L(G)$. □

7.1.2 From Pushdown Automata to Grammars

Theorem 7.2. *Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. There is a context-free grammar G such that $L(G) = N(P)$.*

Proof. We will construct the grammar $G = (V, \Sigma, R, S)$. The nonterminals V consist of:

1. A special starting symbol S
2. All multi-character symbols of the form $[pXq]$, where $p, q \in Q$ and $X \in \Gamma$.

Note these unusual symbols of the form $[pXq]$ will represent a (multi-step) transition from state p to state q in which the net effect is to pop X off the stack. Our intuition about the nonterminals $[qXp]$ is that $[qXp] \Rightarrow^* w$ if and only if $(q, w, X) \vdash^* (p, \epsilon, \epsilon)$. The productions of G are:

a. For all states $p \in Q$, G has the production $S \to [q_0 Z_0 p]$. The idea is to have the language of $[q_0 Z_0 p]$ be those strings w such that $(q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon)$.

b. Let $\delta(q, a, X)$ contain the pair $(r, Y_1 \cdots Y_k)$ where $a \in \Sigma \cup \{\epsilon\}$ and $k \in \mathbf{N}$ which includes the case $k = 0$. Then for all states $r_1, \ldots, r_k \in Q$, G has the production

$$[qXr_k] \to a[rY_1r_1][r_1Y_2r_2] \cdots [r_{k-1}Y_kr_k].$$

The intuition here is that one way to pop X going from state q to state r_k is to read a (which can be ϵ), then use some input to pop Y_1 going from r to r_1, then read more input to pop Y_2 going from r_1 to r_2, and so on. So for the case when $k = 0$, our rule becomes $[qXr] \to a$.

(If) Suppose $(q, w, X) \vdash^* (p, \epsilon, \epsilon)$. We show that $[qXp] \Rightarrow^* w$ by induction on the number of moves made by the PDA.

Base: One step. Then $(p, \epsilon) \in \delta(q, w, X)$ and w is a single symbol or ϵ. From the construction of G, $[qXp] \to w$ is a production, and so $[qXp] \Rightarrow w$.

Ind: Suppose the sequence $(q, w, X) \vdash^* (p, \epsilon, \epsilon)$ takes $n+1$ steps, and the first move looks like $(q, w, X) \vdash (r_0, u, Y_1 \cdots Y_k)$ where $w = au$ for some $a \in \Sigma \cup \{\epsilon\}$. It follows that $(r_0, Y_1 \cdots Y_k) \in \delta(q, a, X)$. By the construction of G, there is a production $[qXp] \to a[r_0Y_1r_1][r_1Y_2r_2] \cdots [r_{k-1}Y_kr_k]$, where $r_1, r_2, \ldots, r_{k-1} \in Q$ and $r_k \coloneqq p$. We can recursively choose the r_j and partition $u = w_1 \cdots w_k$ by computations of the form

$$(r_{j-1}, w_j \cdots w_k, Y_j) \vdash^* (r_j, w_{j+1} \cdots w_k, \epsilon). \tag{7.1}$$

Start by doing the computation $(r_0, u, Y_1) \vdash^* (r_1, \bar{w}_1, \epsilon)$ where $u = w_1\bar{w}_1$ for some string w_1. Then proceed recursively to compute $(r_{j-1}, \bar{w}_j, Y_j) \vdash^* (r_j, \bar{w}_{j+1}, \epsilon)$ where $\bar{w}_j = w_j\bar{w}_{j+1}$ for some string w_j, and moreover $u = w_1 \cdots w_j\bar{w}_{j+1}$. From theorem 6.2, we can remove the tails of the inputs in (7.1) to get $(r_{j-1}, w_j, Y_j) \vdash^* (r_j, \epsilon, \epsilon)$. Further, these sequence of moves can take as many as n moves as they correspond to a proper subsequence of $(q, w, X) \vdash^* (p, \epsilon, \epsilon)$, and so the inductive hypothesis applies. Consequently, $[r_{j-1}Y_jr_j] \Rightarrow^* w_j$. Putting these derivations together, we get

$$\begin{aligned}
[qXp] &\Rightarrow a[r_0Y_1r_1][r_1Y_2r_2] \cdots [r_{k-1}Y_kr_k] \\
&\Rightarrow^* aw_1[r_1Y_2r_2] \cdots [r_{k-1}Y_kr_k] \\
&\vdots \\
&\Rightarrow^* aw_1w_2 \cdots w_k = au = w
\end{aligned}$$

(Only if) The proof is done by induction on the number of steps in the derivation.

Base: One step $[qXp] \Rightarrow w$. Then $[qXp] \to w$ is a production. By rule (b), we have that (p, ϵ) must be inside $\delta(q, w, X)$ where w is a single symbol. So $(q, w, X) \vdash (p, \epsilon)$.

Ind: Suppose $[qXp] \Rightarrow^* w$ in $n+1$ steps. Then

$$[qXp] \Rightarrow a[r_0Y_1r_1][r_1Y_2r_2] \cdots [r_{k-1}Y_kr_k] \Rightarrow^* w$$

where $r_k \coloneqq p$ and the production comes from the fact that $(r_0, Y_1Y_2 \cdots Y_k) \in \delta(q, a, X)$. Break up $w = aw_1w_2 \cdots w_k$ such that each $[r_{j-1}Y_jr_j] \Rightarrow^* w_j$. By the induction hypothesis, it follows that $(r_{j-1}, w_j, Y_j) \vdash^* (r_j, \epsilon, \epsilon)$. Using theorem 6.1, we can pad the ID beyond w_j on the inputs and below Y_j on the stacks in order to get

$$(r_{j-1}, w_jw_{j+1} \cdots w_k, Y_jY_{j+1} \cdots Y_k) \vdash^* (r_j, w_{j+1} \cdots w_k, Y_{j+1} \cdots Y_k).$$

Stringing together all these sequences yields

$$\begin{aligned}
(q, aw_1w_2 \cdots w_k, X) &\vdash (r_0, w_1w_2 \cdots w_k, Y_1Y_2 \cdots Y_k) \\
&\vdash^* (r_1, w_2 \cdots w_k, Y_2 \cdots Y_k) \\
&\vdots \\
&\vdash^* (r_k, \epsilon, \epsilon)
\end{aligned}$$

Now, $S \Rightarrow^* w$ if and only if $[q_0Z_0p] \vdash^* w$ for some p by how the productions for the start symbol was constructed. We just showed that $[q_0Z_0p] \Rightarrow^* w$ if and only if $(q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon)$, which is exactly what is meant by saying that P accepts w on an empty stack. □

Example 7.2. Let us convert the 0^n1^n PDA from example 6.3 to a CFG. The nonterminals of the CFG are a fresh starting symbol S and the triples of the form $[q_jZ_0q_k]$ and $[q_j0q_k]$. But as the PDA can only go from state q_j to q_k when $j \leq k$, we can exclude the triples for which $j > k$. The productions of this grammar are as follows.

1. The only productions from S are of the form $S \to [q_0 Z_0 q_k]$, as we must start with triples that begin with the starting state and starting stack symbol. But notice that we never pop Z_0 from the stack when going from q_0 to q_1 or from q_0 to q_2. It is only by moving from q_0 to itself or from q_0 all the way to q_3 do we ever pop Z_0. Thus, our two rules are summarized below.
$$S \to [q_0 Z_0 q_0] \mid [q_0 Z_0 q_3]$$
2. $\delta(q_0, \epsilon, Z_0) = \{(q_0, \epsilon)\}$ means we have the production rule $[q_0 Z_0 q_0] \to \epsilon$.
3. As $\delta(q_0, 0, Z_0)$ contains $(q_1, 0Z_0)$, we have the production rules $[q_0 Z_0 q_k] \to 0[q_1 0 q_l][q_l Z_0 q_k]$. We already threw out $[q_0 Z_0 q_1]$ and $[q_0 Z_0 q_2]$ because Z_0 doesn't pop when moving from the first state to the second. And having the head be $[q_0 Z_0 q_0]$ would force us to use a triple $[q_j X q_k]$ where $j > k$. So the only head of this rule can be $[q_0 Z_0 q_3]$. And because 0 doesn't pop when looping at q_1, we are forced to have $q_l = q_2$. Our rule is now the following.
$$[q_0 Z_0 q_3] \to 0[q_1 0 q_2][q_2 Z_0 q_3]$$
4. Since $\delta(q_1, 0, 0)$ contains $(q_1, 00)$, we get the production rules $[q_1 0 q_k] \to 0[q_1 0 q_l][q_l 0 q_k]$. By familiar reasoning, we must have $q_l = q_k = q_2$.
$$[q_1 0 q_2] \to 0[q_1 0 q_2][q_2 0 q_2]$$
5. Because $\delta(q_1, 1, 0)$ contains (q_2, ϵ), the production rule generated is $[q_1 0 q_2] \to 1$.
6. $\delta(q_2, 1, 0) = \{(q_2, \epsilon)\}$ and so we get $[q_2 0 q_2] \to 1$.
7. Finally, $[q_2 Z_0 q_3] \to \epsilon$ as $\delta(q_2, \epsilon, Z_0) = \{(q_3, \epsilon)\}$.

We can compact these rules down to the following set.
$$\begin{aligned} S &\to \epsilon \mid 0A \\ A &\to 1 \mid 0A1 \end{aligned}$$

$\square$

Corollary 7.3. *For a context-free language L, the following are equivalent.*

1. *There is a context-free grammar that accepts L.*
2. *There is a pushdown automata that accepts L by final state.*
3. *There is a pushdown automata that accepts L by empty stack.*

7.2 Chomsky Normal Form for Context-Free Grammars

A CFG is said to be in Chomsky normal form if all productions are of the form $A \to BC$ or $A \to a$ where A, B, and C are nonterminals and a is a terminal.

7.2.1 Eliminating Useless Symbols

Definition. A symbol X is *useful* for a grammar $G = (V, T, P, S)$ if there is some derivation of the form $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$ where $w \in T^*$. Note that X can be either a terminal or a nonterminal. A symbol which is not useful will be called *useless*. Say that X is *generating* if there is some $w \in T^*$ such that $X \Rightarrow^* w$, and say that X is *reachable* if there is a derivation $S \Rightarrow^* \alpha X \beta$.

Example 7.3. Consider the following grammar.

$$\begin{aligned} &S \to AB \mid a \\ &A \to b \end{aligned}$$

To eliminate the useless symbols from the grammar, we will first eliminate all non-generating symbols. As S, A, a, and b are all generating (the terminals generate themselves), we are left to eliminate the symbol B and all associated productions, namely $S \to AB$. Then we will remove all unreachable symbols which amount to removing A and b. What we have left is the sole production $S \to a$ which only involves useful symbols.

Note the order in which we did this: (1) remove non-generating symbols and productions and (2) remove unreachable symbols and productions. If we did this in the reverse order, then we'll end up with a grammar that has useless symbols. Starting with reachability, we would find that everything is reachable. Then we would remove B for being non-generating. But that would leave us with a grammar with A and b being unreachable symbols. □

Theorem 7.4. *Let $G = (V, T, P, S)$ be a CFG such that $L(G) \neq \emptyset$. Construct the grammar $G' := (V', T', P', S)$ by*

1. *Eliminate all non-generating symbols and all productions involving at least one of these symbols. Let $G_I := (V_I, T_I, P_I, S)$ be this grammar. Note that S must be generating as we assume that $L(G)$ contains some string.*
2. *Eliminate all symbols that are not reachable from G_I.*

Then G' has no useless symbols and $L(G') = L(G)$.

Proof. Note $V' \subseteq V_I \subseteq V$ and likewise for the terminals and productions. Assume that X is a symbol that remains, i.e. $X \in V' \cup T'$. We want to show that X is useful. So X must generate some $w \in T^*$, $X \underset{G}{\Rightarrow}^* w$. Furthermore, every symbol in that derivation must be generating as well. Thus, $X \underset{G_I}{\Rightarrow}^* w$. Since X survived the second step, we know that X is reachable. Let α and β be strings such that $S \underset{G_I}{\Rightarrow}^* \alpha X \beta$. Further, every symbol used in this derivation is reachable too. Consequently, $S \underset{G'}{\Rightarrow}^* \alpha X \beta$.

Every symbol of $\alpha X \beta$ is reachable and contained inside $V_I \cup T_I$, which makes them generating. The derivation of some terminal string $\alpha X \beta \underset{G_I}{\Rightarrow}^* uwv$ involves only reachable symbols. Therefore, this derivation is also a part of G' as well.

$$S \underset{G'}{\Rightarrow}^* \alpha X \beta \underset{G'}{\Rightarrow}^* uwv$$

Hence, X is useful and in G'. As X was arbitrary, we have that all symbols of G' are useful.

Finally, we must show that $L(G') = L(G)$. As we have only eliminated symbols and productions, it follows that $L(G') \subseteq L(G)$. Given $w \in L(G)$, then we will show that $w \in L(G')$. It follows that $S \underset{G}{\Rightarrow}^* w$, and every symbol in that derivation is both reachable and generating. Thus, $S \underset{G'}{\Rightarrow}^* w$. This means that $w \in L(G')$. We have shown that $L(G') = L(G)$. □

7.2.2 Computing the Generating and Reachable Symbols

Theorem 7.5. *Let $G = (V, T, P, S)$ be a CFG. The following steps find all and only the generating symbols.*

1. *Every symbol of T is generating, as it generates itself.*
2. *For any production $A \to \alpha$ where every symbol of α is generating (including when $\alpha = \epsilon$), then we also know that A is generating. Repeat this step until no more generating symbols are found.*

Proof. We will show that every symbol identified as generating by the algorithm is indeed generating. The base case are the terminal symbols, which obviously generate themselves. Now assume we have a production rule $A \to \alpha$ where we have identified all the symbols in α as generating. By the induction hypothesis, every symbol in α is indeed generating, and so $\alpha \Rightarrow^* w$ for some $w \in T^*$. Then $A \Rightarrow \alpha \Rightarrow^* w$ is generating.

Now we will show that every generating symbol is identified by the algorithm as generating. Assume X is generating, then there is some terminal string w such that $X \Rightarrow^* w$. We will use induction on the length of this derivation.

Base: Zero steps in the derivation. Then X is a terminal symbol, and X was found in the first step.

Ind: The derivation takes $n+1$ steps, and so looks like $X \Rightarrow \alpha \Rightarrow^* w$. So the first step is actually a production rule $X \to \alpha$. Note that every symbol in α derives some substring in the terminal string w in at most n steps. By our induction hypothesis, each symbol of α has been found to be generating by our algorithm. Then the second step of our algorithm will find the production $X \to \alpha$ and conclude that X is also generating. □

Theorem 7.6. *Let $G = (V, T, P, S)$ be a grammar. The following algorithm will find all and only the reachable symbols of G.*

1. *S is clearly reachable.*
2. *If we find that a variable A is reachable, then in every production where A is the head, all the symbols in the body of said production are reachable as well. Repeat this step until no new reachable symbols are found.*

Proof. We start by showing that all the symbols identified by our algorithm are indeed reachable. The base case is the first step, and S is obviously reachable. For the inductive case, assume that A is reachable and there is a production rule $A \to \alpha$. Then $S \Rightarrow^* \beta A \gamma \Rightarrow \beta\alpha\gamma$ is a derivation which demonstrates that every symbol in α is also reachable.

Now we will show that every reachable symbol is indeed identified by our algorithm. Let A be a reachable symbol. We will do induction on the number of steps in the derivation $S \Rightarrow^* A$.

Base: Zero steps in the derivation. So $A = S$, and is obviously reachable.

Ind: Suppose $S \Rightarrow^* \beta X \gamma \Rightarrow \beta\beta' A \gamma' \gamma$ takes $n+1$ steps for some β, β', γ, γ', and nonterminal X. Then the derivation $S \Rightarrow^* \beta X \gamma$ takes n steps and every symbol in $\beta X \gamma$ has been identified as reachable by our algorithm. Further, the derivation $\beta X \gamma \Rightarrow \beta\beta' A \gamma' \gamma$ comes from some production rule $X \to \beta' A \gamma'$. Since X is reachable, our algorithm will identify every symbol in $\beta' A \gamma'$ as reachable too. □

7.2.3 Eliminating Epsilon-Productions

Definition. A nonterminal A in a CFG is *nullable* if $A \Rightarrow^* \epsilon$.

Theorem 7.7. *In any grammar G, the algorithm below will find exactly the nullable symbols.*

1. *If $A \to \epsilon$ is a production of G, then A is nullable.*
2. *If there is a production $B \to C_1 \cdots C_n$ where each C_k is nullable, then B is nullable as well. Note that we only have to consider all nonterminal production bodies. Repeat both steps until no more nullable symbols are found.*

Proof. We aim to show that if the algorithm identifies a nonterminal A as nullable, then it actually is nullable. The base case is the first rule, and any nonterminal with an ϵ-production is obviously nullable. The inductive case is the second rule. Suppose we have a production rule $B \to C_1 \cdots C_n$ where all the C_k are nullable. Then by definition we have derivations $C_k \Rightarrow^* \epsilon$ for each C_k. Combining these derivations, we get that $B \Rightarrow C_1 \cdots C_n \Rightarrow^* \epsilon$. Thus, B is also nullable.

Now we will show that if a symbol A is nullable, then it will be identified as such. We will do this based on induction on the length of the derivation $A \Rightarrow^* \epsilon$.

Base: Derivation is one step. Then the derivation $A \Rightarrow \epsilon$ corresponds to a production rule $A \to \epsilon$. Step 1 of the algorithm will identify A as nullable.

Ind: Derivation $A \Rightarrow^* \epsilon$ is $n+1$ steps. The first step of the derivation must look like $A \Rightarrow C_1 \cdots C_n$ where each C_k must derive ϵ is at most n steps. The inductive hypothesis says that the algorithm has identified each C_k as nullable. Then step 2 of the algorithm will notice that A is nullable as well, because $A \to C_1 \cdots C_n$ must be a production of G. □

Example 7.4. We will eliminate all ϵ-productions form the following grammar.

$$\begin{aligned} &S \to ASB \mid \epsilon \\ &A \to aAS \mid a \\ &B \to SbS \mid A \mid bb \end{aligned}$$

First we eliminate all ϵ-productions. Notice how S is the only nullable symbol. Starting with the first production, $S \to ASB$, we will add a production for each collection of nullable symbols removed. This means that we will add the production $S \to AB$. The same holds for the production $A \Rightarrow aAS$, which is spawn the added production $A \Rightarrow aA$. As there are two nullable symbols in $B \to SbS$, there will be three more productions added: (1) $B \to bS$, (2) $B \to Sb$, and $B \to b$. This gives us the following grammar.

$$\begin{aligned} &S \to ASB \mid AB \\ &A \to aAS \mid aA \mid a \\ &B \to SbS \mid bS \mid Sb \mid b \mid A \mid bb \end{aligned}$$

□

Theorem 7.8. *If the grammar G' is constructed from G by eliminating ϵ-productions, then $L(G') = L(G) \setminus \{\epsilon\}$.*

Proof. We shall prove the more general statement that $A \underset{G'}{\Rightarrow}^* w$ if and only if $A \underset{G}{\Rightarrow}^* w$ and $w \neq \epsilon$. In each direction, the proof will be an induction based on the length of the derivation.

(If) Suppose $A \underset{G}{\Rightarrow}^* w$ and that $w \neq \epsilon$.

Base: The derivation $A \underset{G}{\Rightarrow} w$ is one step. Then $A \to w$ is a production rule of G. And because $w \neq \epsilon$, this is also a production rule of G'.

Ind: The derivation is $n+1$ steps and so looks like $A \underset{G}{\Rightarrow} Y_1 \cdots Y_n \underset{G}{\Rightarrow}^* w$. Break up $w = w_1 \cdots w_n$ so that each $Y_k \underset{G}{\Rightarrow}^* w_k$. Let $X_1, \ldots, X_m$ be a subsequence of the Y_ks, in order, such that the corresponding $w_k \neq \epsilon$. It must be the case that $m \geq 1$ since $w \neq \epsilon$. Therefore, $A \to X_1 \cdots X_m$ must be a production rule of G'. We must have that $X_1 \cdots X_m \underset{G}{\Rightarrow}^* w$ because only those Y_k which are not present are those which were used to derive ϵ. Since each $Y_k \underset{G}{\Rightarrow}^* w_k$ must take at most n steps, we can apply the induction hypothesis to deduce that $Y_k \underset{G'}{\Rightarrow}^* w_k$ so long as $w_k \neq \epsilon$. Therefore, $A \underset{G'}{\Rightarrow} X_1 \cdots X_m \underset{G'}{\Rightarrow}^* w$.

(Only if) Suppose $A \underset{G'}{\Rightarrow}^* w$. Because G' has no ϵ-productions, it follows that $w \neq \epsilon$. We will show that $A \underset{G}{\Rightarrow}^* w$.

Base: The derivation $A \underset{G'}{\Rightarrow} w$ is one step. Then there is a production in G' of the form $A \to w$. By construction of G', there must be a production $A \to \alpha$ of G where α is just w with some nullable nonterminals interspersed. Then $A \underset{G}{\Rightarrow} \alpha \underset{G}{\Rightarrow}^* w$ where in the derivations after the first one, if there are any, derive ϵ for any nonterminals in α.

Ind: Suppose the derivation $A \underset{G'}{\Rightarrow} X_1 \cdots X_m \underset{G'}{\Rightarrow}^* w$ takes $n+1$ steps. The first step in the derivation must come from a G-production of the form $A \to Y_1 \cdots Y_n$ where the Y_ks are the X_js in order with zero or more nullable variables interspersed. Then break up $w = w_1 \cdots w_m$ where $X_k \underset{G'}{\Rightarrow} w_k$, which is a derivation of n or fewer steps. By the induction hypothesis, we have that each $X_k \underset{G}{\Rightarrow}^* w_k$. Finally, construct the derivation

$$A \underset{G}{\Rightarrow} Y_1 \cdots Y_n \underset{G}{\Rightarrow}^* X_1 \cdots X_m \underset{G}{\Rightarrow}^* w_1 \cdots w_m = w$$

as follows. The first step is a use of the G-production $A \to Y_1 \cdots Y_n$. The next set of steps derive ϵ for each of the nullable Y_ks. The final set of steps represent the derivations $X_j \underset{G}{\Rightarrow}^* w_j$, which come from the induction hypothesis.

Lastly, we will derive the statement of the theorem from our more general result.

$$\begin{aligned} w \in L(G') &\Leftrightarrow S \underset{G'}{\Rightarrow}^* w \\ &\Leftrightarrow S \underset{G}{\Rightarrow}^* w \text{ and } w \neq \epsilon \\ &\Leftrightarrow w \in L(G) \setminus \{\epsilon\}. \end{aligned}$$

□

7.2.4 Eliminating Unit Productions

Definition. In a grammar G, call a pair of nonterminals (A, B) a *unit pair* according to the following rules.

1. (A, A) is a unit pair for any nonterminal symbol A.
2. If (A, B) is a unit pair and there exists a production $B \to C$ where C is a nonterminal, then (A, C) is also a unit pair.

Example 7.5. Let us continue example 5.6 on p25 and eliminate unit productions from the binary expression grammar.

To eliminate unit productions from this grammar $G = (V, T, P, S)$ we will make use of the following rules in order to construct $G' = (V, T, P', S)$:

1. Find all unit pairs of G.
2. For each unit pair (A, B), add to P' all the productions of the form $A \to \alpha$ where $B \to \alpha$ is a non-unit production of P. As $A = B$ is possible, P' will contain all the non-unit productions of P.

Unit pairs	Non-unit productions
(E, E)	$E \to E + T$
(E, T)	$E \to T * F$
(E, F)	$E \to (E)$
(E, N)	$E \to 0 \mid 1 \mid 0N \mid 1N$
(T, T)	$T \to T * F$
(T, F)	$T \to (E)$
(T, N)	$T \to 0 \mid 1 \mid 0N \mid 1N$
(F, F)	$F \to (E)$
(F, N)	$F \to 0 \mid 1 \mid 0N \mid 1N$
(N, N)	$N \to 0 \mid 1 \mid 0N \mid 1N$

Table 7.1: Productions of P'

So the constructed grammar from table 7.1 is given below.

$$\begin{aligned}
&E \to E+T \mid T*F \mid (E) \mid 0 \mid 1 \mid 0N \mid 1N \\
&T \to T*F \mid (E) \mid 0 \mid 1 \mid 0N \mid 1N \\
&F \to (E) \mid 0 \mid 1 \mid 0N \mid 1N \\
&N \to 0 \mid 1 \mid 0N \mid 1N
\end{aligned}$$

□

Theorem 7.9. *If the grammar G' is constructed from a grammar G by eliminating unit productions, then $L(G') = L(G)$.*

Proof. Given $w \in L(G')$, we want to show that $w \in L(G)$. So $S \underset{G'}{\Rightarrow}^* w$. As every production in G' is a sequence of zero or more unit productions in G followed by a non-unit production, we have that $\alpha \underset{G'}{\Rightarrow} \beta$ is equivalent to $\alpha \underset{G}{\Rightarrow}^* \beta$. We can convert the derivation $S \underset{G'}{\Rightarrow}^* w$ into a possibly longer derivation $S \underset{G}{\Rightarrow}^* w$. Thus, $w \in L(G)$.

Given $w \in L(G)$, we want to show that $w \in L(G')$. We have that $S \underset{G}{\Rightarrow}^* w$. And by the equivalences in section 5.2, we have that there is a leftmost derivation $S \underset{lm}{\Rightarrow}^* w$. When a unit production, say $A \to B$ is used in the leftmost derivation, the leftmost nonterminal goes from A to B. The leftmost G-derivation can be partitioned into segments consisting of zero or more unit productions followed by a non-unit production. Each of these segments of the derivation can be represented by a single G'-production. Therefore, $S \underset{G'}{\Rightarrow}^* w$. Consequently, $w \in L(G')$. □

7.2.5 Chomsky Normal Form

We have to follow the order of elimination below otherwise we might reintroduce things that were previously removed.

1. Eliminate ϵ-productions
2. Eliminate unit productions
3. Eliminate useless symbols

Theorem 7.10. *If G is a CFG generating a language which contains at least one string other than ϵ, then there exists another CFG G' such that $L(G') = L(G) \setminus \{\epsilon\}$ and G' has no ϵ-productions, no unit productions, nor any useless symbols.*

Proof. Start by eliminating the ϵ-productions using theorem 7.8. Next, eliminate any unit productions like in theorem 7.9. This will not introduce any ϵ-productions, since the bodies of new productions are identical to old productions. Finally, eliminate useless symbols and their productions as in theorem 7.4. This final elimination only discards symbols and productions, and thus will introduce no new ϵ or unit productions. □

Definition. A grammar G is said to be in *Chomsky normal form (CNF)* if it has no useless symbols and every production has one of two forms.

1. $A \to BC$ where A, B, and C are all nonterminals.
2. $A \to a$ where A is a nonterminal and a is a terminal.

Theorem 7.11. *If G is a CFG whose language consists of at least one string other than ϵ, then there is a grammar G' in Chomsky normal form such that $L(G') = L(G) \setminus \{\epsilon\}$.*

Proof. By theorem 7.10, we can find a CFG G_I such that $L(G_I) = L(G) \setminus \{\epsilon\}$ and such that G_I has no useless symbols, no ϵ-productions, nor any unit productions. The construction we will describe converts G_I into a grammar G' by changing the productions so that one or more production from G' will simulate a single production of G_I. While, the new variables we introduce into G' will have only one production.

Given a terminal string $w \in L(G_I)$, we will replace each production used, say it has the form $A \to X_1 \cdots X_n$, by a sequence of productions in G'. If any X_k is a terminal, we introduce a possibly new nonterminal B_k and production $B_k \to X_k$. If $n > 2$, then we add the necessary variables and productions to G' so that G' has the productions $A \to B_1C_1$, $C_1 \to B_2C_2$, $C_2 \to B_3C_3$, and so on, where each $B_k = X_k$ if X_k is a nonterminal or B_k is a variable with a production $B_k \to X_k$ otherwise. These productions of G' emulate a single step derivation from G_I. Thus, there is a derivation of w in G', and so $w \in L(G')$.

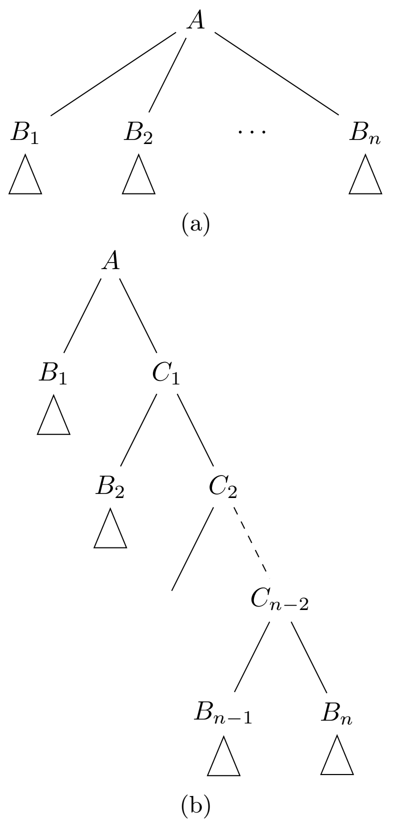


Figure 7.1: A parse tree in G' must use new nonterminals in a particular way

Given $w \in L(G')$, there is a parse tree in G' whose root is S and whose yield is w. We will first "undo" the Chomsky construction. This is to say that at any node labeled A with two children labeled B_1 and C_1 where C_1 is one of the new nonterminals we introduced in the construction of G'. Because these new nonterminals have only one production, there is only one way that they can appear, giving us back the production $A \to B_1 \cdots B_k$ as shown in figure 7.1.

We may not yet have a parse tree of G_I because we introduced extra productions that produce only single terminals. However, we can identify these new productions and associated new nonterminals in the parse tree. They would have some nonterminal labeled A with a single child a, and we can replace those subtrees by the single node a. Now every interior node of the parse tree forms a production in G_I. And the yield of the parse tree is w. Therefore, $w \in L(G_I)$. □

Example 7.6. We will continue example 7.4 on p38 by converting the grammar to CNF.

$$\begin{aligned}
&S \to ASB \mid \epsilon\\
&A \to aAS \mid a\\
&B \to SbS \mid A \mid bb
\end{aligned}$$

First we eliminated all ϵ-productions. The result has one unit production, $B \to A$.

$$\begin{aligned}
&S \to ASB \mid AB\\
&A \to aAS \mid aA \mid a\\
&B \to SbS \mid bS \mid Sb \mid b \mid A \mid bb
\end{aligned}$$

Removing the unit production, we get the grammar below.

$$\begin{aligned}
&S \to ASB \mid AB\\
&A \to aAS \mid aA \mid a\\
&B \to SbS \mid bS \mid Sb \mid b \mid bb \mid aAS \mid aA \mid a
\end{aligned}$$

We introduce new nonterminals so the production bodies can be of the correct form.

$$\begin{aligned}
S &\to AC_1 \mid AB\\
A &\to D_1C_2 \mid D_1A \mid a\\
B &\to SC_3 \mid D_2S \mid SD_2 \mid b \mid D_2D_2 \mid D_1C_2 \mid D_1A \mid a\\
C_1 &\to SB\\
C_2 &\to AS\\
C_3 &\to D_2S\\
D_1 &\to a\\
D_2 &\to b
\end{aligned}$$

□

7.3 Pumping Lemma for Context-Free Languages

Theorem 7.12. *Given a Chomsky normal form grammar $G = (V, T, P, S)$ and a parse tree whose yield is the terminal string w. If the length of the longest path in the tree is n, then $|w| \leq 2^{n-1}$.*

Proof. The proof is an induction on n.

Base: $n = 1$. Thus, w is a single terminal as we are in CNF. Hence, $|w| = 1 = 2^{1-1}$.

Ind: Suppose the longest path has length $n+1$. Then the root of the tree uses a production which must be of the form $A \to BC$, as the tree could not start with the production of a single terminal. Note no path rooted at the nodes B or C can have length greater than n. So by our induction hypothesis, we have that the yield of these two nodes is bounded above by 2^{n-1}. The yield of B and C concatenated together has to be at most $2^{n-1} + 2^{n-1} = 2^n$ in length. □

Theorem 7.13 (Pumping lemma for CFLs)**.** *Let L be a context-free language. Then there exists a constant $n_L \in \mathbf{N}$ such that if $z \in L$ and $|z| \geq n_L$, then we can write $z = uvwxy$ which must satisfy the following conditions.*

1. $|vwx| \leq n_L$

2. $vx \neq \epsilon$

3. For all $k \in \mathbf{N}$, $uv^kwx^ky \in L$.

Proof. First, we find a Chomsky normal form G for L. Note there is no such grammar if $L = \emptyset$ or $L = \{\epsilon\}$. If L is empty, then the theorem is vacuously true. Next, suppose $L = \{\epsilon\}$. Again, the theorem is vacuously true for $n_L := 1$.

With a CNF grammar $G = (V, T, P, S)$ such that $L(G) = L \setminus \{\epsilon\}$, suppose G has m nonterminals, i.e. $m := |V|$. Set $n_L := 2^m$. Let $z \in L$ be given with a length of at least n_L. Theorem 7.12 guarantees that any parse tree whose longest path is m or less, then its yield must be at most $2^{m-1} = n_L/2$ in length. Therefore, a parse tree for z must have a length of at least $m + 1$. Let the nonterminals $A_0, \ldots, A_k$ be the interior nodes on the longest path in zs parse tree. We have already seen that $k + 1 \geq m + 1$, or more simply $k \geq m$, meaning the longest path in the parse tree for z has at least $m + 1$ nonterminals on it. But there are only m nonterminals in the grammar. By the pigeonhole principle, there is a duplication in the tail of the longest path so that $A_i = A_j$ for some $k - m \leq i < j \leq k$.

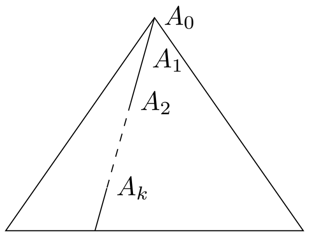


Figure 7.2: Every sufficiently long string in L must have a long path on its parse tree

Then we can divide $z = uvwxy$. As there are no unit productions, it must be the case that $vx \neq \epsilon$. Since we choose the tail of the longest path to apply the pigeonhole principle, theorem 7.12 states that the length of vwx must be at most $2^m = n_L$. Finally, we can construct parse trees of G for strings of the form uv^kwx^ky for every $k \in \mathbf{N}$ like in figure 7.4.

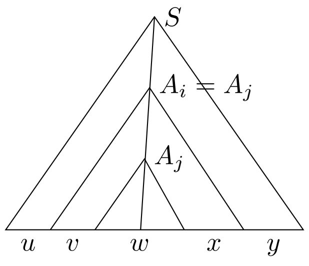
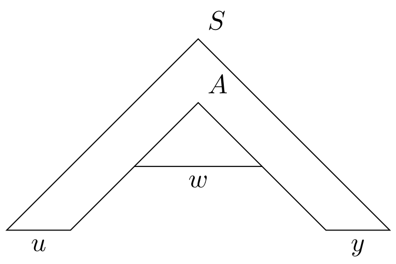


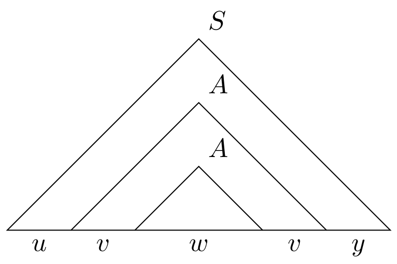
Figure 7.3: Partitioning string z in order to be pumped

□

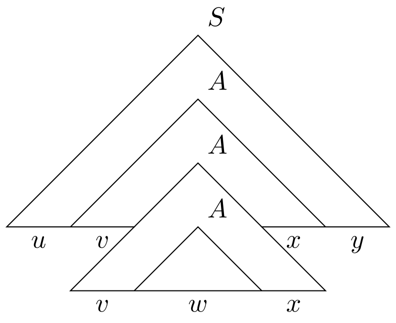
Example 7.7. The language $L = \{0^k1^k2^k \mid k \geq 1\}$ is not context-free. Assume otherwise. Given any $n \in \mathbf{N}$ for the pumping lemma, we choose the string $z = 0^n1^n2^n \in L$. For any splitting of $z = uvwxy$ with $|vwx| \leq n$ and $vx \neq \epsilon$, we aim to find some string of the form $uv^kwx^ky \notin L$ for some $k \in \mathbf{N}$. We know that vwx cannot contain both 0s and 2s because the rightmost 0 and the leftmost 2 are separated by $n + 1$ positions. This puts us in one of two cases: (1) vwx has no 2s or (2) vwx has no 0s. Without loss of generality, assume that vwx has not 2s. Thus vx consists of just 0s and 1s and must contain at least one of



(a) Pumping zero times



(b) Pumping once



(c) Pumping twice

Figure 7.4: Pumping strings v and x

these symbols. Then uwy has n 2s yet either fewer than n 0s or fewer than n 1s. Consequently, $uwy \notin L$. Our assumption that L is a CFL must be incorrect. □

Example 7.8. We will show the language $L = \{0^p \mid p \text{ is prime}\}$ is not context-free. Assume otherwise. Given an $n \in \mathbf{N}$, choose a string $w \in L$ whose length is some prime $p \geq n + 2$. For any splitting of w into $w = uvwxy$ with $vx \neq \epsilon$ and $|vwx| \leq n$, set $m := |vx|$. Note $0 < m \leq n$ and $|uwy| = p - m$. Call $z := uv^{p-m}wx^{p-m}y$ is a string of length $p - m + m(p - m) = (p - m)(1 + m)$. Because $1 + m > 1$ and $p - m \geq p - n \geq 2$, z is a string of 0s whose length is not prime. As this violates the pumping lemma for CFLs, our assumption that L was context-free must be incorrect. So L is not context-free. □

7.4 Closure Properties of Context-Free Languages

7.4.1 Substitutions

Definition. A *substitution* over an alphabet Σ is a function $s : \Sigma \to \mathcal{P}(T^*)$ that assigns to each symbol $a \in \Sigma$ a language $s(a)$. If $w = a_1 \cdots a_n$ is a string over Σ, then $s(w)$ is the concatenation of the languages $s(a_1) \cdots s(a_n)$. If L is a language over Σ, i.e. $L \subseteq \Sigma^*$, then $s(L)$ is the union of all languages of the form $s(w)$ where $w \in L$.

Theorem 7.14. *If L is a context-free language over an alphabet Σ, and s is a substitution over Σ such that $s(a)$ is a CFL for every $a \in \Sigma$, then $s(L)$ is a CFL too.*

Proof. The main idea is that we take a grammar for L, and replace each terminal a by the starting symbol for the grammar for $s(a)$. The result will be a single CFG which generates the language $s(L)$. We start with a grammar $G = (V, \Sigma, P, S)$ for L and for each $a \in \Sigma$, a grammar $G_a = (V_a, T_a, P_a, S_a)$ for $s(a)$. We will choose names for the nonterminals so that there is no overlap in V and all of the V_as. Construct a new grammar $G' := (V', T', P', S)$ as follows.

- $V' := V \cup \bigcup_{a \in \Sigma} V_a$ is the union of V and all of the V_as.

- $T' := \bigcup_{a \in \Sigma} T_a$ is the union of all the T_as.
- P' consists of two types of productions: (1) all productions in P_a for every $a \in \Sigma$ and (2) the productions of P but with each terminal a replaced by its corresponding start symbol, S_a.

We will show that $L(G') = s(L)$.

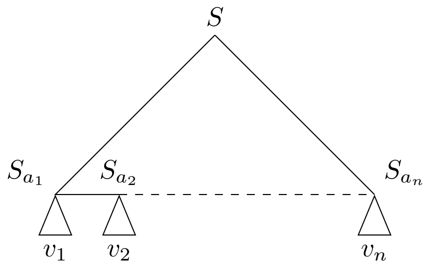


Figure 7.5: A parse tree for G' begins with a parse tree for G and ends with many parse trees from the G_as

Given $w \in s(L)$, there is some string $u = a_1 \cdots a_n \in L$ and some strings $v_k \in s(a_k)$ for $k = 1, \ldots, n$ such that $w = v_1 \cdots v_n$. Then the productions of G' that come from G can get us as far as a parse tree whose yield is the nonterminal string $S_{a_1} \cdots S_{a_n}$, see the top triangle in figure 7.5. To get the rest of the way, we will use the productions of G' that come from the G_as. The yield of this larger tree is $v_1 \cdots v_n = w$, and thus $w \in L(G')$.

Given $w \in L(G')$, we claim that the parse tree T for w must look like figure 7.5. This is because the nonterminals of G and the G_as are all distinct. The top of T, starting from S, must use only productions that came from G until some symbol S_a is reached. Downward from such a point, only productions from G_a can be used. Consequently, we can identify a string $a_1 \cdots a_n \in L(G)$ which is the top of the parse tree T, and strings $v_k \in s(a_k)$ such that (1) $S_{a_1} \cdots S_{a_n}$ is the yield of some subtree of T starting from S and (2) $w = v_1 \cdots v_n$. This, by definition, makes $w \in s(L)$ because it can be partitioned into substrings that come from the $s(a_k)$. $\square$

Theorem 7.15. *The context-free language are closed under the following operations.*

1. *Union*
2. *Concatenation*
3. *Kleene closure*
4. *Homomorphism*

Proof. We shall setup a substitution for each operation and use theorem 7.14.

1. Union: Let L_1 and L_2 be CFLs. Over the alphabet $\{1, 2\}$, define the substitution s as $s(1) := L_1$ and $s(2) := L_2$. Then as the finite language $\{1, 2\}$ is clearly context-free, $s(\{1, 2\}) = L_1 \cup L_2$ is as well.
2. Concatenation: Using the same construction as before, we can conclude that $s(\{12\}) = L_1 L_2$ is context-free because the language $\{12\}$ is also context-free.
3. Kleene closure: Using a similar construction, we can say that $s(\{1\}^*) = L_1^*$ is a CFL because the language $\{1\}^*$ is too.
4. Homomorphism: Let L be a CFL over the alphabet Σ and $h : \Sigma \to T^*$ be a homomorphism, then define the substitution s as $s(a) := \{h(a)\}$ for every $a \in \Sigma$. Then we will have that $h(L) = s(L)$ is context-free.

$\square$

7.4.2 Reversal

Theorem 7.16. *If L is a CFL, then so is L^R.*

Proof. Let $L = L(G)$ for some CFG $G = (V, T, P, S)$. Construct $G^R := \left(V, T, P^R, S\right)$, where P^R consists of productions of the form $A \to \alpha^R$ whenever $A \to \alpha$ is in P. It is an easy induction on the length of a derivation to show that all the sentential forms of G^R are the reverse of a corresponding sentential form of G and vice-versa, which is left as an exercise for the reader. Therefore, $L\left(G^R\right) = L(G)^R$. $\square$

7.4.3 Intersection with a Regular Language

The context-free languages are not closed under intersection, as can be seen by the counterexample below.

Example 7.9. From the example 7.7 p43, we know that the language

$$L = \{0^n 1^n 2^n \mid n \geq 1\}$$

is not context-free. However, we will show the following two languages are yet their intersection is equal to L.

$$\begin{aligned} L_1 &= \{0^n 1^n 2^k \mid n, k \geq 1\} \\ L_2 &= \{0^k 1^n 2^n \mid n, k \geq 1\} \end{aligned}$$

A grammar for L_1 is

$$\begin{aligned} &S \to AB \\ &A \to 0A1 \mid 01 \\ &B \to 2B \mid 2 \end{aligned}$$

where A generates all strings of the form 0^n1^n while B generates the 2s. Similarly, we have a grammar for L_2 which follows.

$$\begin{aligned} &S \to AB \\ &A \to 0A \mid 0 \\ &B \to 1B2 \mid 12 \end{aligned}$$

To see why $L = L_1 \cap L_2$, recognize that to be in L_1 means there are as many 0s as 1s, while membership in L_2 is having as many 1s as 2s. A string having equal numbers of all three symbols is exactly string in $L_1 \cap L_2$. Thus, CFLs cannot be closed under intersection. $\square$

Theorem 7.17. *If L is a CFL and R is a regular language, then $L \cap R$ is a CFL.*

Proof. We will run the corresponding PDA and DFA "in parallel" which will yield another PDA. Let $P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$ be a PDA that accepts L by final state, and let $A = (Q_A, \Sigma, \delta_A, q_A, F_A)$ be a DFA for R. Define the PDA P' as $P' := (Q_P \times Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A)$ where

$$\delta((q,p), a, X) := \{\left(\left(r, \hat{\delta}_A(p,a)\right), \gamma\right) \mid (r, \gamma) \in \delta_P(q, a, X)\}.$$

Note, we had to use $\hat{\delta}_A$ because a can either be a symbol or ϵ. It is an easy induction, left to the reader, on the number of moves by the two PDAs that $(q_P, w, Z_0) \underset{P}{\vdash^*} (q, \epsilon, \gamma)$ if and only if $((q_P, q_A), w, Z_0) \underset{P'}{\vdash^*} ((q, \hat{\delta}(q_A, w)), \epsilon, \gamma)$. This shows that P' accepts w if and only if both P and A do too. $\square$

7.4.4 Inverse Homomorphism

Theorem 7.18. *Let L be a CFL over the alphabet T, and let $h : \Sigma \to T^*$ be a homomorphism. Then $h^{-1}(L)$ is a CFL.*

Proof. We will start with a PDA $P = (Q, T, \Gamma, \delta, q_0, Z_0, F)$ which accepts L by final state. Construct another PDA $P' \coloneqq (Q', \Sigma, \Gamma, \delta', (q_0, \epsilon), F \times \{\epsilon\})$ where

1. $Q' \coloneqq \{(q, x) \mid q \in Q \land \exists a \in \Sigma.\, x \text{ is a suffix of } h(a)\}$. The first component of the state is a state from P, and the second component will be a buffer for the strings produced from h.
2. δ' is defined as follows:
   (a) $\delta'((q, \epsilon), a, X) \coloneqq \{((q, h(a)), X)\}$ for all $a \in \Sigma$, $X \in \Gamma$, and $q \in Q$. When the buffer is empty, P' will consume its next input a and place the string $h(a)$ onto the buffer.
   (b) $\delta'((q, bx), \epsilon, X) \coloneqq \{((p, x), \gamma) \mid (p, \gamma) \in \delta(q, b, X)\}$ for all $(q, bx) \in Q'$ with $b \in \Sigma \cup \{\epsilon\}$ and for any $X \in \Gamma$. P' has the option of simulating a move from P, using the front of the buffer.

Note that P' starts at the initial state of P and with an empty buffer. The accepting states of P' are the accepting states of P with an empty buffer.

The relationship between P and P' is given by the following statement.

$$(q_0, h(w), Z_0) \underset{P}{\vdash^*} (p, \epsilon, \gamma) \text{ if and only if } ((q_0, \epsilon), w, Z_0) \underset{P'}{\vdash^*} ((p, \epsilon), \epsilon, \gamma)$$

The proofs in both directions are inductions on the number of moves made by the two PDAs. For the "if" portion of the proof, we need to observe that once the buffer of P' is nonempty, it cannot read another symbol on the input until the buffer empties. The reader is left to complete this part of the proof. Once this is proven, we can see that P accepts $h(w)$ if and only if P' accepts w. Consequently, $L(P') = h^{-1}(L)$. □

7.5 Decision Problems for CFLs

7.5.1 Testing Emptiness of a CFL

We can test if the start symbol for a CFG is generating from section 7.2.2 in $\mathcal{O}(n)$ time where n is the length of entire representation of the CFG.

7.5.2 Testing Membership in a CFL

We can test whether a string w is a member of a CFL L in $\mathcal{O}(n^3)$ time.

7.5.3 Undecidable CFL Problems

1. Determine whether a given grammar is ambiguous.
2. Determine whether a given CFL is inherently ambiguous.
3. Determine whether the intersection of two CFLs is empty.
4. Determine whether a given CFL, over an alphabet Σ, is equal to Σ^*.

Chapter 8

Turing Machines

8.1 Definition

8.1.1 Informal Definition

A Turing machine is a finite state machine that runs on a tape which extends infinitely in two directions, left and right. The tape is partitioned into (countably) infinitely many squares or cells, each of which can hold exactly one symbol out of an alphabet of tape symbols. Initially, the tape consists of a finite number of cells which contain an input symbol, and the remaining cells are blank. There is a tape reader which is always positioned over a cell, and the system is said to be scanning that one cell. At the beginning, the reader is at the leftmost cell that holds an input symbol.

A move of the Turing machine is a function of the current state and symbol being scanned. In a move, the machine will

1. Change state, and note the new state can be the same as the current one,
2. Write a tape symbol to the cell being scanned, which replaces the cell's current tape symbol,
3. Move the tape reader to the next cell on the left or right of the current cell.

8.1.2 Formal Definition

Definition. A *Turing Machine (TM)*, M, shall be defined as a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where

1. Q is a finite, nonempty set of *states*
2. Σ is an alphabet of *input symbols* with $\Sigma \subseteq \Gamma$
3. Γ is an alphabet of *tape symbols*
4. δ is the *transition function* which is a partial function from $Q \times \Gamma$ to $Q \times \Gamma \times \{\text{Left}, \text{Right}\}$
   (a) The input to δ is a pair of the current state and current tape symbol being scanned.
   (b) The output to δ, if defined, is a triple (p, Y, D) consisting of the next state p, the new tape symbol Y being written to the cell being scanned, and a direction D—either left or right—in which the tape is to be shifted.
5. $q_0 \in Q$ is the *start state*
6. $B \in \Gamma \setminus \Sigma$ is the *blank tape symbol* that can be found on all but finitely many cells initially
7. $F \subseteq Q$ is the set of *final* or *accepting states*

8.1.3 Instantaneous Descriptions for Turing Machines

While the tape may extend infinitely in two directions, a Turing machine starts with finitely many non-blank symbols on the tape, and can only visit a finite number of cells after a finite number of moves. Therefore, at every step, there are always only a finite number of non-blank cells. So we need to describe a finite number of non-blank symbols, the current state of the machine, and the current position of the tape reader. To represent this as a single string, we will have the states and the tape symbols be distinct. We can rename the states if necessary.

Definition. The string $X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n$ to represent the *instantaneous description (ID)* such that

1. q is the state of the Turing machine
2. The tape head is scanning the ith symbol from the left
3. $X_1\cdots X_n$ is the segment of tape between the leftmost and rightmost non-blank symbol. Unless the head is to the left of the leftmost non-blank symbol or to the right of the rightmost, then we make an exception and allow some prefix or suffix of this segment to be blank, and i will be 1 or n, respectively.

We denote the *transition relation* for the Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ to be $\underset{M}{\vdash}$, or just $\vdash$ when M is clear, and $\underset{M}{\vdash^*}$ to be its transitive, reflexive closure. This transition relation is defined as follows.

1. If $\delta(q, X_i) = (p, Y, L)$, then in general
$$X_1\cdots X_{i-1}qX_i\cdots X_n \underset{M}{\vdash} X_1\cdots X_{i-2}pX_{i-1}YX_{i+1}\cdots X_n$$
the system writes Y to the ith cell and moves left. There are two exceptions to this rule.
   (a) If $i = 1$, then
$$qX_1\cdots X_n \underset{M}{\vdash} pBYX_2\cdots X_n$$
   as the tape scanner is now over a blank cell.
   (b) If $i = n$ and $Y = B$, then
$$X_1\cdots X_{n-1}qX_n \underset{M}{\vdash} X_1\cdots X_{n-2}pX_{n-1}$$
   as the reader has extended the trailing sequence of infinite blanks.
2. If $\delta(q, X_i) = (p, Y, R)$, then in general
$$X_1\cdots X_{i-1}qX_i\cdots X_n \underset{M}{\vdash} X_1\cdots X_{i-1}YpX_{i+1}\cdots X_n$$
the machine writes Y to the ith cell and moves right. Again, we have two exceptions.
   (a) If $i = 1$ and $Y = B$, then
$$qX_1\cdots X_n \underset{M}{\vdash} pX_2\cdots X_n$$
   as the tape reader has extended the leading sequence of infinite blanks.
   (b) If $i = n$, then
$$X_1\cdots X_{n-1}qX_n \underset{M}{\vdash} X_1\cdots X_{n-1}YpB$$
   as the scanner is now over a blank cell.

Example 8.1. We will design a Turing machine that accepts the language $\{0^n1^n \mid n \geq 1\}$. The input to this machine will be a binary string with infinitely many blanks to the left and to the right. This machine will (1) mark the cell with an X provided it contains a 0; (2) move right past any more 0s or Ys until it hits a 1, which it will then rewrite as a Y, or it hits a blank and at this point is done; (3) move left past any Ys or 0s until it hits an X at which point it moves right; (4) go back to step (1).

Formally,

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\})$$

with δ being given by the table below.

State\Symbol	0	1	X	Y	B
q_0	(q_1, X, R)	$-$	$-$	(q_3, Y, R)	$-$
q_1	$(q_1, 0, R)$	(q_2, Y, L)	$-$	(q_1, Y, R)	$-$
q_2	$(q_2, 0, L)$	$-$	(q_0, X, R)	(q_2, Y, L)	$-$
q_3	$-$	$-$	$-$	(q_3, Y, R)	(q_4, B, R)
q_4	$-$	$-$	$-$	$-$	$-$

Table 8.1: A Turing machine that accepts $\{0^n1^n \mid n \geq 1\}$

On the input 0011, this machine M will perform the following sequence of moves.

$$\begin{gathered} q_0 0011 \vdash X q_1 011 \vdash X 0 q_1 11 \vdash X q_2 0 Y 1 \vdash q_2 X 0 Y 1 \\ X q_0 0 Y 1 \vdash X X q_1 Y 1 \vdash X X Y q_1 1 \vdash X X q_2 Y Y \vdash X q_2 X Y Y \\ X X q_0 Y Y \vdash X X Y q_3 Y \vdash X X Y Y q_3 B \vdash X X Y Y B q_4 B \end{gathered}$$

On the input 0010, the machine will do the following and halts without accepting.

$$\begin{gathered} q_0 0010 \vdash X q_1 010 \vdash X 0 q_1 10 \vdash X q_2 0 Y 0 \vdash q_2 X 0 Y 0 \\ X q_0 0 Y 0 \vdash X X q_1 Y 0 \vdash X X Y q_1 0 \vdash X X Y 0 q_1 B \end{gathered}$$

□

8.1.4 Transition Diagram for Turing Machines

A transition diagram for Turing machines will be like all our transition diagrams. The transition $\delta(q, X) = (p, Y, D)$ will be represented by an edge from q to p labeled as X/YD.

Example 8.2. The transition diagram for the previous example is given below. □

8.1.5 Language of a Turing Machine

Definition. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a Turing machine. The *language accepted by M*, denoted $L(M)$, is

$$L(M) := \{w \in \Sigma^* \mid \exists \alpha, \beta \in \Gamma^*. \exists p \in Q.\, q_0 w \vdash^* \alpha p \beta \wedge p \in F\}.$$

The set of languages accepted by Turing machines is called the *computably enumerable languages* or CE languages. The other, more common, notion of acceptance for Turing machines is *acceptance by halting.* The Turing machine M *halts* when it enters a state q and scans a tape symbol X for which it has no moves, i.e. $\delta(q, X)$ is undefined. The set of languages for which there is a Turing machine that always halts even if it does not accept are the *computable languages.* A computable language is also called *decidable.* A language which is not computable is called *undecidable.*

We can assume that a TM halts when it enters some accepting state. However, we cannot assume that a TM will halt regardless of whether it accepts or not.

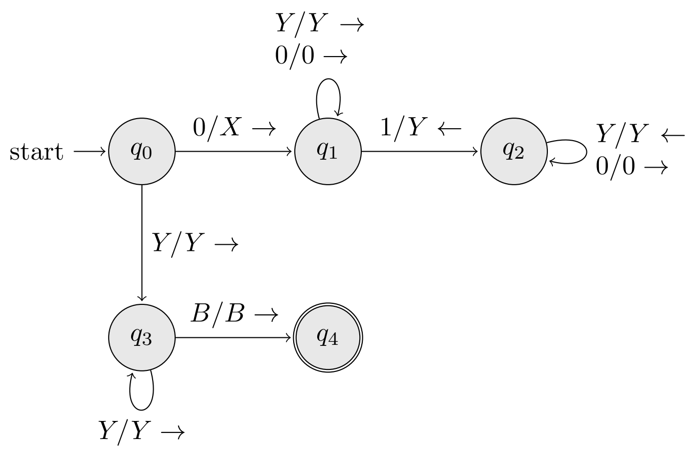


Figure 8.1: A transition diagram for a TM that accepts strings of the form 0^n1^n

8.1.6 Restricted Turing Machines

Theorem 8.1. *Every language accepted by a TM M is also accepted by a TM M' with the following restrictions:*

1. *M' never moves left past its initial position.*
2. *M' never writes a blank.*

Proof. Condition (2) is easy: create a new tape symbol B' that functions as a blank but is different from B.

1. If M has the rule $\delta(q, X) = (p, B, D)$, then change this rule to $\delta'(q, X) = (p, B', D)$.
2. Let $\delta'(q, B')$ be the same as $\delta(q, B)$ for every state q.

The first condition requires more effort. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be an equivalent Turing machine that never writes a blank. Construct $M' = (Q', \Sigma \times \{B\}, \Gamma', \delta', q_0', (B, B), F')$ where:

- $Q' \coloneqq \{q_0', q_1'\} \cup (Q \times \{U, L\})$. The states of M' are the initial state q_0', another state q_1', and all the states in M paired with an additional component U signalling the upper-half of the tape or L for lower-half of the tape.
- $\Gamma' \coloneqq \Gamma \times (\Gamma \cup \{*\})$ is all pairs of tape symbols from M plus a tape symbol paired with a marker $*$ for the left end of the tape for M'.
- δ' is defined by the following rules.
    1. $\delta'(q_0', (a, B)) \coloneqq (q_1', [a, *], R)$ for any $a \in \Sigma$. The first move of M' marks the left end of the tape and moves right, as it cannot remain stationary or move left.
    2. $\delta'(q_1', (X, B)) \coloneqq ((q_0, U), (X, B), L)$ for any $X \in \Gamma$. This moves M' back to its left end and changes the state to the initial state of M.
    3. If $\delta(q, X) = (p, Y, D)$, then for every $Z \in \Gamma$:
        (a) $\delta'((q, U), (X, Z)) \coloneqq ((p, U), (Y, Z), D)$
        (b) $\delta'((q, L), (Z, X)) \coloneqq \big((p, L), (Z, Y), \overline{D}\big)$

       where $\overline{D}$ is the opposite direction to D. If M' is not on the leftmost cell, then it emulates M on the appropriate track.

4. If $\delta(q, X) = (p, Y, R)$, then

$$\delta'((q, L), (X, *)) \coloneqq \delta'((q, U), (X, *)) \coloneqq ((p, U), (Y, *), R).$$

This covers how M moves right past its initial position.

5. If $\delta(q, X) = (p, Y, L)$, then

$$\delta'((q, L), (X, *)) \coloneqq \delta'((q, U), (X, *)) \coloneqq ((p, L), (Y, *), R).$$

This covers the case when M moves left past its initial position.

- $F' = F \times \{U, L\}$. So M' is accepting regardless of whether it is focused on its upper or lower track.

We can prove by induction on the number of moves made by M that M' will mimic those moves. Also, M' accepts exactly when M accepts. Therefore, $L(M') = L(M)$. □

8.1.7 Multi-Tape Turing Machines

Theorem 8.2. *Every language accepted by a multi-tape TM is computably enumerable.*

Proof. Given a language L that is accepted by a k-tape TM M, we will simulate M with a one-tape TM N that can be thought of has $2k$ tracks on the tape. Half of these tracks will contain the tapes of M, and the other half will contain only a single marker that indicates where the head for the corresponding tape of M is located. In order to emulate a move of M, N has to visit the k head markers. So N does not get lost, it must keep track of how many heads are to its left. This count will be stored as a component of N's state. After visiting each head marker and storing the scanned symbol in a component of its state, N knows what tape symbols are being scanned by each of M's heads. N also knows the state of M, which it stores in its state. So N knows what move M will make.

N must now revisit each of the head markers, change the symbol in the track representing the corresponding tape of M, and move the head marker left or right, if necessary. By now, N has simulated one move of M. As for accepting states, choose all the states of N that record M's state as one of the accepting states of M. Thus, N will accept exactly when M accepts. □

8.1.8 Nondeterministic Turing Machine

Definition. A *nondeterministic Turing machine* is like its deterministic variant except that $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\text{Left}, \text{Right}\})$ returns a set of triples rather than zero or one.

As it turns out, nondeterministic TMs are not more powerful than the deterministic kind. But simulating a nondeterministic machine on a deterministic one is computationally expensive.

Theorem 8.3. *If M_N is a nondeterministic Turing machine, then there is a deterministic Turing machine M_D such that $L(M_N) = L(M_D)$.*

Proof. M_D will be constructed as a multi-tape TM. The first tape of M_D will hold a sequence of IDs of M_N which include the state of M_N. There needs to be an inter-ID separator, such as '*', and a marker for the current ID being processed, such as 'x'. All IDs to the left of the current one have been explored and can therefore be ignored. To process the current ID, M_D does the following.

1. M_D examines the state and scanned symbol of the current ID. Built into the finite state of M_D is the knowledge of what choices of move M_N has for each state and symbol. If the state in the current ID is accepting, then M_D accepts and stops simulation of M_N.

2. If the state is not accepting, and the state-symbol combination has k moves, then M_D uses its second tape to copy the ID and then make k copies of the ID at the end of the sequence of IDs on tape 1.

3. M_D modifies each of the k IDs according to one of the k moves that M_N can make from the current ID.

4. M_D returns to the marked current ID, erases the mark and moves the mark to the next ID to the right. Repeat from step (1).

We need to confirm that if M_N enters an accepting ID after a sequence of n of its own moves, then M_D will eventually make that ID the current ID and accept. Suppose m is the maximum number of choices M_N has in any configuration. Then there is one initial ID of M_N, at most m IDs that M_N can reach after one move, at most m^2 IDs M_N can reach after two moves, and so on. After n moves, M_N can reach at most $1 + m + m^2 + \cdots + m^n \leq nm^n$ IDs. Thus, the accepting ID of M_N will be considered by M_D among the first nm^n IDs examined. We only care that M_D considers this ID in a finite amount of time. Therefore, if M_N accepts, then M_D will eventually accept. Hence, $L(M_N) = L(M_D)$. □

8.2 Undecidability

8.2.1 Codes for Turing Machines

We shall map all TMs with binary input to the natural numbers. For a TM $M = (Q, \{0,1\}, \Gamma, \delta, q_1, B, F)$, we first must assign numbers to the states, to the tape symbols, and to each direction.

- We may assume the states are $q_1, q_2, \ldots, q_r$ for some r. The starting state will always be q_1, and q_2 will be the only accepting state. As we may assume that the TM halts whenever it accepts, we will only need the one accepting state.
- We will assume that the tape symbols are $X_1, X_2, X_3, \ldots, X_s$ for some s. We will always have that $X_1 = 0$, $X_2 = 1$, and $X_3 = B$.
- We take D_1 to be the left direction and D_2 to be the right.

Now, we can encode the transition function δ. Given a transition rule $\delta(q_i, X_j) = (q_k, X_l, D_m)$, we encode this rule as the binary string $0^i10^j10^k10^l10^m$. Notice how our conventions demand that all i, j, k, l and m are all at least 1. So there are no two or more consecutive 1s in our code. The code for M will be

$$C_1 11 C_2 11 \cdots C_{n-1} 11 C_n$$

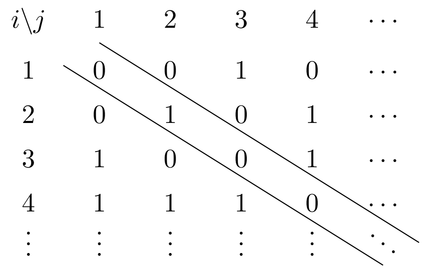
where the C_ks are the codes for all the transitions of M. Note that not all binary strings are codes for some TM. For example, 1101 cannot be a TM because it starts with a 1, and 01110 cannot be a TM because it contains three consecutive 1s.

8.2.2 Diagonal Language

Now with a concrete notion of M_i the "ith Turing machine", we can now define the *diagonal language*, L_d, as the set of binary strings w_i where w_i is the ith binary string such that it is *not accepted* by M_i. This is called the diagonal language because one can think of a table like in figure 8.2 where each entry represents whether machine M_i accepts the jth binary string, with 1 representing acceptance and 0 representing rejection.

Theorem 8.4. *L_d is not a computably enumerable language.*

Proof. Suppose L_d was accepted by some Turing machine M. As L_d is a language over $\{0,1\}$, there is some code for M, say i, i.e. $M = M_i$. Now, we inquire whether w_i is in L_d. If so, then M_i accepts w_i. But by the definition of L_d, it follows that w_i is not in L_d. This is a contradiction. If $w_i \notin L_d$, then M_i does not accept w_i. Again by the definition L_d, w_i is in L_d. Yet another contradiction. Our assumption that L_d is computably enumerable must be false. □



$i\backslash j$	1	2	3	4	$\cdots$
1	0	0	1	0	$\cdots$
2	0	1	0	1	$\cdots$
3	1	0	0	1	$\cdots$
4	1	1	1	0	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Figure 8.2: A crude sketch of the table that represents acceptance of strings by Turing machines

8.2.3 Complements of Computable Languages

Theorem 8.5. *If L is a computable language, then so is L^C.*

Proof. Let $L = L(M)$ for some Turing machine M that always halts. We will construct a TM M' such that $L(M') = L^C$ by turning acceptance into rejection and vice-versa. We modify M in order to create M' as follows:

1. The accepting states of M are made into non-accepting states for M' with no transitions out. So M' will halt without accepting.
2. M' has a new accepting state r, and there are no transitions out of r.
3. For each pair of a non-accepting state of M and a tape symbol such that M has no transition, which is to say M halts without accepting, add a transition to the accepting state r.

As M is guaranteed to halt, we have constructed M' so that it is also certain to halt. Further, M' accepts only those strings that M does not. Therefore, M' accepts L^C. $\square$

Theorem 8.6. *If both a language L and its complement L^C are computably enumerable, then they are both computable.*

Proof. Let $L = L(M_1)$ and $L^C = L(M_2)$. We will simulate M_1 and M_2 in parallel by a TM M, by making M a two-tape TM. One tape of M will simulate M_1, while the other tape simulates M_2. The states of M_1 and M_2 are components of the state of M. If the input w to M is in L, then M_1 will accept. In which case, M will also accept and halt. Otherwise, $w \notin L$ and M_2 will accept. In this case, M will halt without accepting. Since M always halts, and $L(M) = L$, we can say that L is computable. And by theorem 8.5, L^C is also computable. $\square$

8.2.4 Universal Language and a Universal Turing Machine

From section 8.2.1, we described how to convert a binary TM into a binary string, in which there were no three consecutive 1s. We can encode a pair $\langle M, w\rangle$ of a binary Turing machine and a binary string into a single binary string $C111w$ where C is the code for M. The *universal language* L_u is the language of binary strings which are codes for a Turing machine and input pair $\langle M, w\rangle$ such that $w \in L(M)$. We shall show that L_u is computably enumerable by demonstrating a *universal Turning machine*, U, that accepts L_u.

The TM U will be a multi-tape machine. On the first tape, U will store the input $\langle M, w\rangle$ in binary form. The second tape will be used to store the tape of M using the same format as section 8.2.1. Tape symbol X_k of M will be represented by 0^k, and tape symbols are separated by single 1s. The third tape of U will hold the state of M, with the state q_k represented by 0^k. The moves of U will be summarized below.

1. Examine the code for M and make sure it is a legitimate code for a TM. If not, U halts without accepting. Invalid codes are assumed to represent a TM without any moves, which will thus accept no input.

2. Initialize the second tape to contain the input w, in its encoded form. For each 0 of w, place 10 on the tape, and for each 1 of w, place 100 on the tape. Blanks on M are simulated by 1000, and these do not yet appear on the second tape of U. Instead, U knows that when it looks for a simulated symbol of M and finds its own blank instead, it must replace that blank with the sequence of symbols 1000 to simulate the blank of M.

3. Place 0, the representation of the start state of M, on the third tape. Move the head of U's second tape to the first simulated cell.

4. To emulate a move of M, U searches on its first tape for a transition $0^i10^j10^k10^l10^m$ such that 0^i is the state on tape 3, 0^j is the tape symbol of M that begins on tape 2 at the position being scanned by U. If such a transition is found, then this is the transition that M would make, and U should do the following.

   (a) Change the contents of tape 3 to 0^k. First U will change all the zeros in 0^i to blanks. Then it will copy 0^k from tape 1 to tape 3.

   (b) Replace 0^j on tape 2 by 0^l, which will change the tape symbol of M. If more or less space is needed, use the scratch tape, tape 4, in order to manage spacing.

   (c) Move the head of tape 2 to the position of next 1 on the left or right, respectively, depending on whether $m = 1$ (move left) or $m = 2$ (move right).

5. If no such transition is found that matches the simulated state and tape symbol, then in (4) no transition will be found. So M halts, and U must do likewise.

6. If M enters its accepting state, then U should accept as well.

8.2.5 Undecidability of the Universal Language

Theorem 8.7 (Halting problem)**.** *L_u is computably enumerable but not computable.*

Proof. In section 8.2.4, we showed that L_u is CE. Suppose L_u is computable. Then by theorem 8.5, the complement of L_u, L_u^C, would also be computable. We will show that if we have a TM M that accepts L_u^C, then we can construct a TM that accepts L_d, which would be a contradiction. Suppose for some TM M that $L_u^C = L(M)$. We will use M to construct a TM M' that accepts L_d as follows.

1. Given a string w on its input, M' changes the input to $w111w$.

2. M' emulates M on the new input. Suppose w is w_i in our enumeration of binary strings. Then M' accepts determines whether M_i accepts w_i. Since M accepts L_u^C, M' will accept if and only if M_i does not accept w_i, which is to say that $w_i \in L_d$.

Thus M' accepts w if and only if $w \in L_d$. As M' cannot exist by theorem 8.4, we conclude that L_u is not computable. $\square$

8.3 Undecidable Problems about Turing Machines

8.3.1 Reductions

Definition. A function $f : \Sigma^* \to \Sigma^*$ is a *computable function* if there is a Turing machine which when given input w halts with $f(w)$ on its tape. We say that a language L_1 *reduces* to language L_2, and write $L_1 \leq_m L_2$ if there is a computable function $f : \Sigma^* \to \Sigma^*$ such that

$$w \in L_1 \Leftrightarrow f(w) \in L_2.$$

Theorem 8.8. *If there is a reduction from L_1 to L_2, then*

1. *when L_1 is undecidable, L_2 is too, and*

2. *when L_1 is not computably enumerable, L_2 is too.*

Proof. First suppose L_1 is undecidable. If it is possible to decide L_2, then we can combine the reduction from L_1 to L_2 with the machine that decides L_2 in order to construct an algorithm that decides L_1. Given an instance u of L_1, apply the reduction that converts u into an instance v of L_2. Then use the machine that decides L_2 on v. If the output is "yes", then v is in L_2. Because we reduced L_1 to L_2, the answer to u for L_1 is "yes" as well, i.e. u is in L_1. Similarly, if v is not in L_2, then u is not in L_1. Whatever answer we give to the question "is v in L_2" is also the correct answer to "is u in L_1". This contradicts the assumption that P_1 is undecidable. We can therefore conclude that if L_1 is undecidable, then so is L_2.

Assume that L_1 is not CE, but L_2 is. We have an machine that reduces L_1 to L_2 and a machine that only recognizes instances of L_2. Starting with an instance u of L_1, reduce it to an instance v of L_2. Apply the TM for L_2 to v. If v is accepted by, then accept u. Otherwise, the machine for L_2 may not halt, and so our procedure will not halt as well. As we assumed that no TM exists for L_1, we have shown by contradiction that no TM exists for L_2 either. Thus, if L_1 is not CE, then neither is L_2. ☐

Corollary 8.9. *If $L_1 \leq_m L_2$, then*

1. *when L_2 is decidable, then L_1 is too, and*

2. *when L_2 is computably enumerable, then L_1 is too.*

Proof. This is the contrapositive of the previous theorem. ☐

8.3.2 Turing Machines That Accept the Empty Language

Let $L_{ne} \coloneqq \{\langle M \rangle \mid L(M) \neq \emptyset\}$ be the language of binary codes for TMs that accepts at least one string. Its complement L_{ne}^C is the language of binary codes for TMs that accept no strings.

Theorem 8.10. *L_{ne} is computably enumerable.*

Proof. We need to construct a TM that accepts L_{ne}. It is easiest to describe a nondeterministic TM M. And by theorem 8.3, M can be converted to a deterministic TM. The operation of M is as follows.

1. M takes as input a code for the TM M_i.

2. Using nondeterminism, M guesses an input w that M_i might accept.

3. M tests whether M_i accepts w. For this part, M emulates a universal TM U which accepts L_u.

4. If M_i accepts w, then M accepts its own input, which is M_i.

If M_i accepts one string, M will guess this string and accept M_i. Otherwise, $L(M_i) = \emptyset$, then no guess for w will be accepted by M_i, and so M will not accept M_i. Thus, $L(M) = L_{ne}$. ☐

Theorem 8.11. *L_{ne} is not computable.*

Proof. We will design an algorithm that converts an input that is a binary-coded pair $\langle M, w \rangle$ into a TM M' such that $L(M') \neq \emptyset$ if and only if M accepts w as input, that is if and only if $\langle M, w \rangle \in L_u$. In fact, if M accepts w, then M' will accept every input. Otherwise M does not accept w and M' will accept no input. M' is designed as follows.

1. M' ignores its own input u. Instead, it replaces its input by the string that represents the pair $\langle M, w \rangle$. As M' is designed for a specific pair $\langle M, w \rangle$, which has some length n, we can construct M' to have a sequence of states $q_0, q_1, \ldots, q_n$, where q_0 is the starting state.

   (a) In state q_i, for $i = 0, 1, \ldots, n-1$, M' writes the $(i+1)$st bit of the code $\langle M, w \rangle$, then goes to state q_{i+1}, and moves right.
   (b) In state q_n, M' moves right, if necessary, replacing any non-blanks in the tail of x by blanks.

2. When M' reaches a blank in state q_n, it uses another, similar collection of states to move its head to the left end of the tape.

3. Now, using additional states, M' simulates a universal TM U on its present tape.

4. If U accepts, then M' accepts. Otherwise, M' and U will never accept.

This machine M' is a reduction from L_u to L_{ne}. Given an encoded pair $\langle M, w \rangle$, if M accepts w, i.e. $\langle M, w \rangle \in L_u$, then M' will accept all of its inputs and so $\langle M' \rangle \in L_{ne}$. Otherwise, $\langle M, w \rangle \notin L_u$ and M' will accept none of its inputs. Consequently, $\langle M' \rangle \notin L_{ne}$. By theorems 8.8 and 8.7, we have that L_{ne} is not computable. □

Corollary 8.12. $L_{ne}^C = \{\langle M \rangle \mid L(M) = \emptyset\}$ *is not computably enumerable.*

Proof. The previous theorem and theorem 8.5. □

8.3.3 Rice's Theorem

Definition. A *property* of CE languages is a subset of all CE languages. A property is *trivial* if it is either empty or the entire set of CE languages. Otherwise, the property is *nontrivial.*

As most languages are infinite, they cannot be written down as a finite-length string on a TM. Instead, we will identify a language as the TM which accepts that language, as TMs can be encoded as finite strings. So a property P of CE languages will be identified as a language L_P which is the set of codes for TMs M_i such that $L(M_i)$ is a language in P. When talking about the decidability of a property P, we mean decidability of the language L_P.

Theorem 8.13 (Rice's Theorem)**.** *Every nontrivial property of the computably enumerable languages is undecidable.*

Proof. Let P be a nontrivial property of CE languages. Assume for the time being that $\emptyset \notin P$. Then there must be some nonempty language $L \in P$. Let M_L be a Turing machine that accepts L. We will reduce L_u to L_P which will prove that L_P is undecidable. The reduction will take an encoded pair $\langle M, w \rangle$ as input and produce a TM M'. M' will accept nothing if M does not accept w, and M' will accept $L = L(M_L)$ otherwise.

M' will be a two-tape TM. One tape is used to simulate M on w. The other tape of M' is used to simulate M_L on the input u to M', if necessary. M' is constructed to do the following.

1. Emulate M on input w. This can be done by writing $\langle M, w \rangle$ to one of its tapes, then running a universal Turing machine on that tape.

2. If M does not accept w, then M' does nothing else. M' never accepts its own input x, and so $L(M') = \emptyset$. As we assumed $\emptyset \notin P$, the code for M' is not in L_P.

3. If M accepts w, then M' begins to emulate M_L on its own input x. Thus M' will accept exactly L. Because $L \in P$, $\langle M' \rangle \in L_P$.

Finally, we need to consider the case where $\emptyset \in P$. If so, then the complement property P^C has just been proven to be undecidable. As L_P^C is the set of codes for Turing machines that do not accept a language in P, we have that $L_P^C = L_{P^C}$ is the set of codes for Turing machines that accept a language in P^C. Because the complement of a computable language is computable, see theorem 8.5, it follows that L_P, being the complement of L_{P^C} cannot be decidable. □

8.4 Post's Correspondence Problem

Definition. An instance of *Post's correspondence problem (PCP)* consists of two, equal length lists of strings over some alphabet Σ. We generally refer to the two lists as A and B writing $A = u_1, u_2, \ldots, u_n$ and $B = v_1, v_2, \ldots v_n$ for some integer n. For each k, the pair (u_k, v_k) are a *corresponding pair.* The instance of PCP *has a solution* if there is a sequence of at least one integer $k_1, k_2, \ldots k_m$ such that $u_{k_1} u_{k_2} \cdots u_{k_m} = v_{k_1} v_{k_2 \cdots v_{k_m}}$. The sequence $k_1, \ldots, k_m$ is a *solution* to the instance of PCP.

Example 8.3. Over the alphabet $\Sigma = \{0, 1\}$, let the lists be given in table 8.2. In this case, the PCP has a solution: $k_1 = 2$, $k_2 = 1$, $k_3 = 1$, $k_4 = 3$. We can verify this is a solution by concatenating the corresponding

	List A	List B
k	u_k	v_k
1	1	111
2	10111	10
3	10	0

Table 8.2: An instance of PCP

pairs.

$$u_2 u_1 u_1 u_3 = v_2 v_1 v_1 v_3 = 101111110$$

Note this solution is not unique because $2, 1, 1, 3, 2, 1, 1, 3$ is also a solution. □

8.4.1 Modified Problem

Definition. The *modified Post's correspondence problem (MPCP)* adds the additional requirement that the first corresponding pair in lists A and B must be the first pair in the solution. Given two lists $A = u_1, \ldots, u_n$ and $B = v_1, \ldots, v_n$, a solution is a sequence of zero or more integers $k_1, \ldots, k_m$ such that $u_1 u_{k_1} \cdots u_{k_m} = v_1 v_{k_1} \cdots v_{k_m}$.

We are going to reduce MPCP to PCP. Given an instance of MPCP with alphabet Σ and lists $A = u_1, \ldots, u_n$ and $B = v_1, \ldots, v_n$, we will construct an instance of PCP as follows. First, introduce two new symbols, say $*$ and \$, into the PCP instance. Construct lists $C = w_0, \ldots, w_{n+1}$ and $D = x_0, \ldots, x_{n+1}$ for PCP as follows.

1. For $k = 1, \ldots, n$, let w_k be u_k with a $*$ after each symbol, and let x_k be v_k with a $*$ before each symbol.
2. $w_0 \coloneqq *w_1$ and $x_0 \coloneqq x_1$. This along with the previous rule will force the solution to PCP to start with 0.
3. $w_{n+1} \coloneqq \$$ and $x_{n+1} = *\$$.

Example 8.4. Consider example 8.3 as an instance of MPCP. Then converting it to an instance of PCP by the above steps results in the following. □

	List C	List D
k	w_k	x_k
0	$*1*$	$*1*1*1$
1	$1*$	$*1*1*1$
2	$1*0*1*1*1*$	$*1*0$
3	$1*0*$	$*0$
4	\$	$*\$$

Table 8.3: Modifying an instance of MPCP to PCP

Theorem 8.14. *The above construction reduces MPCP to PCP.*

Proof. Given a solution $k_1, \ldots, k_m$ to an instance of MPCP with lists $A = u_1, \ldots, u_n$ and $B = v_1, \ldots, v_n$. We have that $u_{k_1} \cdots u_{k_m} = v_{k_1} \cdots v_{k_m}$. Replacing the us with ws and vs with xs, we would get two strings that are almost the same: $w_1 w_{k_1} \cdots w_{k_m}$ and $x_1 x_{k_1} \cdots x_{k_m}$. The difference would be that the first string would be missing a $*$ at the beginning, and the second would be missing a $*$ at the end.

$$*w_1 w_{k_1} \cdots w_{k_m} = x_1 x_{k_1} \cdots x_{k_m} *$$

As $w_0 = *w_1$ and $x_0 = x_1$, we can fix the initial missing $*$ by swapping the first corresponding pair for the zeroth.

$$w_0 w_{k_1} \cdots w_{k_m} = x_0 x_{k_1} \cdots x_{k_m} *$$

We can fix the trailing $*$ by appending the $(n+1)$st corresponding pair, since $w_{n+1} = \$$ and $x_{n+1} = *\$$.

$$w_0 w_{k_1} \cdots w_{k_m} w_{n+1} = x_0 x_{k_1} \cdots x_{k_m} x_{n+1}$$

Therefore, we have that $0, k_1, \ldots, k_m, n+1$ is a solution to the constructed PCP instance.

We must now show the converse, that if the constructed PCP instance has a solution, then the original MPCP instance has a solution as well. Note that any solution to the PCP instance must begin with the zeroth corresponding pair, as this is the only pair that start with the same character, and end with the $(n+1)$st corresponding pair, as this is the only pair that end with the same character. So the PCP solution must look like $0, k_1, \ldots, k_m, n+1$. If we remove the extra $*$s and \$ from $w_0 w_{k_1} \cdots w_{k_m} w_{n+1}$, then we will get the string $u_1 u_{k_1} \cdots u_{k_m}$. Similarly, we get $v_1 v_{k_1} \cdots v_{k_m}$ by removing the extra symbols from $x_0 x_{k_1} \cdots x_{k_m} x_{n+1}$. Because $w_0 w_{k_1} \cdots w_{k_m} w_{n+1} = x_0 x_{k_1} \cdots x_{k_m} x_{n+1}$, we have the following.

$$u_1 u_{k_1} \cdots u_{k_m} = v_1 v_{k_1} \cdots v_{k_m}$$

□

8.4.2 Undecidability

We will now reduce L_u to MPCP. The idea is that the constructed instance of MPCP (A, B) simulates the computation of TM M on input w, in its partial solutions. The partial solutions will consist of strings that are prefixes of IDs of M: $\#\alpha_1\#\alpha_2\#\alpha_3\#\cdots$, where $\alpha_1 = q_0 w$ is the initial ID of M with input w, and $\alpha_k \vdash \alpha_{k+1}$ for all k. The string from B will always be one ID ahead of A unless M enters an accepting state.

Using the fact that there is always an equivalent TM which never writes a blank and never goes left past its initial head position from theorem 8.1, let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be such a TM and let $w \in \Sigma^*$ be the input string. Construct the corresponding pairs as follows.

1. The first pair, which must be used by the rules of MPCP, starts A empty and B with the initial ID.

List A	List B
$\#$	$\# \, q_0 w \, \#$

2. Tape symbols and the separator # can be appended freely to both lists.

List A	List B	
X	X	for each X in Γ
$\#$	$\#$	

3. For all $q \in Q \setminus F$, $p \in Q$, and $X, Y, Z \in \Gamma$, we have:

List A	List B	
qX	Yp	if $\delta(q, X) = (p, Y, R)$
ZqX	pZY	if $\delta(q, X) = (p, Y, L)$
$q\#$	$Yp\#$	if $\delta(q, B) = (p, Y, R)$
$Zq\#$	$pZY\#$	if $\delta(q, B) = (p, Y, L)$

4. For all $q \in F$ and $X, Y \in \Gamma$, we add some pseudo-IDs representing what would happen if the accepting state were able to consume tape symbols on either side.

List A	List B
XqY	q
Xq	q
qY	q

5. After the accepting state has consumed all tape symbols, it stands alone as the last pseudo-ID on list B. So for all $q \in F$,

List A	List B
$q\#\#$	$\#$

Example 8.5. We will convert the TM $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, \{q_2\})$ where δ is given by the table 8.4 and the input is $w = 01$ to an instance of MPCP. M accepts w by the following sequence of moves.

q_k	$\delta(q_k, 0)$	$\delta(q_k, 1)$	$\delta(q_k, B)$
q_0	$(q_1, 1, R)$	$(q_1, 0, L)$	$(q_1, 1, L)$
q_1	$(q_2, 0, L)$	$(q_0, 0, R)$	$(q_1, 0, R)$
q_2	$-$	$-$	$-$

Table 8.4: Table giving δ for example 8.5

$$q_0 01 \vdash 1q_1 1 \vdash 10q_0 \vdash 1q_1 01 \vdash q_2 101$$

Starting with the first pair as required by MPCP, we have:

$$\begin{aligned} A :& \# \\ B :& \#q_0 01\# \end{aligned}$$

Next, we are forced to use the pair $(q_0 0, 1q_1)$ from rule (3) which is one of the move-simulating pairs.

$$\begin{aligned} A :& \#q_0 0 \\ B :& \#q_0 01\#1q_1 \end{aligned}$$

Using the copying pairs from rule (2) until we get to the next state, we arrive at

$$\begin{aligned} A :& \#q_0 01\#1 \\ B :& \#q_0 01\#1q_1 1\#1 \end{aligned}$$

Then, we must simulate another move with the rule (3) pair $(q_1 1, 0q_0)$.

$$\begin{aligned} A :& \#q_0 01\#1q_1 1 \\ B :& \#q_0 01\#1q_1 1\#10q_0 \end{aligned}$$

Again using rule (2) to copy symbols, gets us the following.

$$\begin{aligned} A :& \#q_0 01\#1q_1 1\#1 \\ B :& \#q_0 01\#1q_1 1\#10q_0\#1 \end{aligned}$$

Now we use the move-simulating pair $(0q_0\#, q_1 01\#)$ from rule (3).

$$\begin{aligned} A :& \#q_0 01\#1q_1 1\#10q_0\# \\ B :& \#q_0 01\#1q_1 1\#10q_0\#1q_1 01\# \end{aligned}$$

Another rule (3) pair, $(1q_10, q_210)$, gets us to acceptance.

$$\begin{aligned}A :&\#q_001\#1q_11\#10q_0\#1q_10\\ B :&\#q_001\#1q_11\#10q_0\#1q_101\#q_210\end{aligned}$$

We can now utilize the copying rule (2) and rule (4) to consume all the tape symbols, leaving q_2 by itself.

$$\begin{aligned}A :&\#q_001\#1q_11\#10q_0\#1q_101\#q_2101\#q_201\#q_21\#\\ B :&\#q_001\#1q_11\#10q_0\#1q_101\#q_2101\#q_201\#q_21\#q_2\#\end{aligned}$$

Finally, rule (5) will finish the job.

$$\begin{aligned}A :&\#q_001\#1q_11\#10q_0\#1q_101\#q_2101\#q_201\#q_21\#q_2\#\#\\ B :&\#q_001\#1q_11\#10q_0\#1q_101\#q_2101\#q_201\#q_21\#q_2\#\#\end{aligned}$$

□

Theorem 8.15. *Post's correspondence problem is undecidable.*

Proof. We have already reduced MPCP to PCP in theorem 8.14. And the above construction shows how we will reduce L_u to MPCP. Thus we will show that a TM M accepts its input w if and only if the constructed MPCP instance has a solution.

(Only if) Example 8.5 gives the basic idea. If $w \in L(M)$, then rule (1) will start off the MPCP. A pair from rule (3) will enable us to copy the state and simulate a move of M, while rule (2) will allow us to copy the rest of an ID as well as the separation symbol #. Once we reach an accepting state, the rules (4) and (5) allow the A string to catch up with the B string and form a solution.

(If) Suppose the constructed MPCP instance has a solution, we need to argue that M accepts w. Because we are dealing with MPCP, we have to start with the first pair, and so a partial solution begins as follows.

$$\begin{aligned}A :&\#\\ B :&\#q_0w\#\end{aligned}$$

Unless there is an accepting state in the partial solution, we are restricted by only using rules (2) and (3). And these rules always add as many symbols or more to B than they do to A. Since we assumed that this MPCP instance has a solution, it must follow that we encounter an accepting state. Consequently, M must accept w by our assumptions. □

Chapter 9

Intractable Problems

9.1 P and NP Classes

Definition. A Turing machine M is of *time complexity* $T(n)$ if whenever M is given an input of length n, then M halts after at most $T(n)$ moves, regardless of whether or not M accepts. We say that a language L is *in class P* (polynomial) if there is some polynomial $T(n)$ such that $L = L(M)$ for some deterministic TM M of time complexity $T(n)$. A language L is *in class NP* (nondeterministic polynomial) if there is some polynomial time complexity $T(n)$ for a nondeterministic TM M such that $L = L(M)$.

Definition. A reduction from L_1 to L_2 is *a polynomial-time reduction*, denoted as $L_1 \leq_p L_2$, if the running time of the reduction is polynomial in the size of the instance of L_1. A language L is said to be *NP-complete* if (1) L is in NP and (2) for every language L' that is in NP, there is a polynomial-time reduction of L' to L.

Theorem 9.1. *If L_1 is NP-complete, L_2 is in NP, and there is a polynomial-time reduction from L_1 to L_2, then L_2 is also NP-complete.*

Proof. We need to show that every language L in NP reduces in polynomial-time to L_2. Our assumption is that there is a reduction of L to L_1 whose time complexity is some polynomial $p(n)$ and that there is a reduction from L_1 to L_2 whose time complexity is some polynomial $q(m)$. A string $u \in L$ of length n can be converted into a string v of L_1 whose length is at most $p(n)$. Then the string v of L_1 can be transformed into a string w taking at most $q(p(n))$ time. The combined time to change u into w takes at most $p(n) + q(p(n))$ time, which is a polynomial. So L can be reduced to L_2 in polynomial-time, which means that L_2 is NP-complete. $\square$

Theorem 9.2. *If some NP-complete problem is in P, then P = NP.*

Proof. Suppose language L is both NP-complete and in P. Then every language L' in NP can be reduced in polynomial-time to L. Consequently, L' is in P. $\square$

9.2 The SAT Problem

Definition. A *truth assignment*, or *truth valuation, for a boolean expression E* is an mapping ϕ from the variables in E to true or false. Such a mapping can be extended to be over the entire expression E.

We can say that a truth assignment over E extends to another truth assignment over some expression E' with more variables, or we can say that a truth assignment restricts to another one over an expression with fewer variables.

Theorem 9.3 (Cook-Levin Theorem). *SAT is NP-complete.*

Proof. The proof that SAT is in NP is easy. Use the nondeterminism of an NTM to guess a truth assignment ϕ for the given expression E. Evaluate E with the assignment ϕ. If $\phi(E)$ is true, then accept. This can be done in polynomial time on an NTM.

The hard part is showing that if L is any NP language, then there is some polynomial time reduction of L to SAT. We will need the restrictions of theorem 8.1 to allow us to say that our NTM M for L never moves left past its initial head position and never writes a blank. As L is in NP, let the number of moves made by M be bounded above by the polynomial $p(n)$ where n is the length of the input. So when M accepts an input w with $|w| = n$, then there is a sequence of moves such that

1. α_0 is the initial ID of M with input w.
2. $\alpha_0 \vdash \alpha_1 \vdash \cdots \vdash \alpha_m$ where $m \leq p(n)$ and α_m is an ID with an accepting state.
3. Each α_k can be considered to consist of non-blanks only and extends from the initial head position to the right.

Our strategy will be to write each α_k as a sequence of symbols $X_{k0}, X_{k1}, \ldots, X_{k,p(n)}$, where each symbol is either a state or a tape symbol (but not both). Note that there is no need to represent symbols to the right of the first $p(n)$ symbols on the tape as M is guaranteed to halt after $p(n)$ steps. Next, to describe the sequence of IDs as boolean variables, we create a variable y_{jkA} to represent the proposition $X_{jk} = A$, where $0 \leq j, k \leq p(n)$ and A is either a tape symbol or a state. Finally, we will express the conditions that (1) there is a unique symbol in each cell, (2) the starting ID of M is correct, (3) the moves from one ID to another are correct, and (4) that M finishes in some accepting state. Formally, we give an algorithm to construct a boolean expression $E_{M,w}$ from M and w where $E_{M,w}$ has the form $U \wedge S \wedge N \wedge F$ where U expresses that all cells have a unique symbol in them, and S, N, and F express that M starts, moves, and finishes correctly.

Unique
U is the logical conjunction of all terms of the form $\neg(y_{ij\alpha} \wedge y_{ij\beta})$ where $\alpha \neq \beta$. There are $\mathcal{O}\left(p(n)^2\right)$ such terms.

Start
For S, note that X_{00} must be q_0, the starting state of M, X_{01} through X_{0n} must be the symbols of $w = a_1 a_2 \cdots a_n$, and the remaining X_{0k} must be blank. Thus, we have that

$$S = y_{00q_0} \wedge y_{01a_1} \wedge \cdots \wedge y_{0na_n} \wedge y_{0,n+1,B} \wedge \cdots \wedge y_{0,p(n),B}.$$

This can be done in $\mathcal{O}(p(n))$ time.

Finish
For F, we may assume that an accepting state repeats itself indefinitely. So acceptance by M is the same as finding acceptance in $\alpha_{p(n)}$. The statement that the kth cell in the $p(n)$th ID is accepting can be written as $F_k \coloneqq y_{p(n),k,a_1} \vee y_{p(n),k,a_2} \vee \cdots \vee y_{p(n),k,a_m}$ where the $a_1, a_2, \ldots, a_m$ are the accepting states of M. Then the statement that there is an accepting state amongst the symbols of the $p(n)$th ID is $F \coloneqq F_0 \vee F_1 \vee \cdots \vee F_{p(n)}$. This can be written out in $\mathcal{O}(p(n))$ time.

Next
The most complicated part is showing that M moves correctly between IDs. N will be the conjunction of expressions N_j for $j = 0, 1, \ldots, p(n) - 1$ which each state that the move $\alpha_j \vdash \alpha_{j+1}$ is valid. The N_j will be a conjunction of expressions $A_{jk} \vee B_{jk}$ for $k = 0, 1, \ldots, p(n)$. The A_{jk} say that the state of α_j is at position k, i.e. X_{jk} is a state, and that there is a move from M where X_{jk} is the state and $X_{j,k+1}$ is the symbol scanned which transforms the triple $X_{j,k-1}X_{jk}X_{j,k+1}$ into the triple $X_{j+1,k-1}X_{j+1,k}X_{j+1,k+1}$. The B_{jk} say that the state of α_j is not at position k, i.e. X_{jk} is not a state, and that if $X_{j,k-1}$ and $X_{j,k+1}$ are not states as well, then $X_{j+1,k} = X_{jk}$. (Note the case where the state is next to position k will be handled by $A_{j,k-1}$ or $A_{j,k+1}$.)

The B_{jk} are easiest to write. Letting $q_0, q_1, \dots, q_m$ being the states of M and $Z_0, Z_1, \dots, Z_r$ being the tape symbols of M, then

$$B_{jk} = \left(\bigvee_{l=1}^{m} (y_{j,k-1,q_s} \vee y_{j,k+1,q_s}) \right) \vee \left(\left(\bigvee_{s=1}^{r} y_{j,k,Z_s} \right) \wedge \left(\bigvee_{s=1}^{r} (y_{j,k,Z_s} \wedge y_{j+1,k,Z_s}) \right) \right).$$

The cases where $k = 0$ or $k = p(n)$ are special because either the atoms $y_{j,k-1,X}$ or $y_{j,k+1,X}$ do not exist, we need to delete some terms in each of these cases in order to make them valid expressions.

The expressions A_{jk} will represent all possible configurations of the six variables $X_{j,k-1}$, $X_{j,k}$, $X_{j,k+1}$, $X_{j+1,k-1}$, $X_{j+1,k}$, and $X_{j+1,k+1}$. Such a configuration is valid provided:

1. X_{jk} is a state, but $X_{j,k\pm1}$ are tape symbols.
2. There is a move from M that explains how $X_{j,k-1}X_{jk}X_{j,k+1}$ becomes $X_{j+1,k-1}X_{j+1,k}X_{j+1,k+1}$.

As there are a finite number of assignments to these six variables which are valid, we can take A_{jk} to be the disjunction of these assignments.

With the A_{jk} and B_{jk} defined, we construct the N_j as

$$N_j = \bigwedge_{k=0}^{p(n)} A_{jk} \vee B_{jk},$$

and

$$N = N_0 \wedge N_1 \wedge \cdots \wedge N_{p(n)-1}.$$

The length of each N_j is $\mathcal{O}(p(n))$, and so the length of N is $\mathcal{O}\left(p(n)^2\right)$.

Conclusion

For any NTM M with time complexity $p(n)$, we have an algorithm that takes an input w of length n and produces the boolean expression $E_{M,w}$ and the running time of this deterministic conversion is polynomial in n. □

9.3 Restricted SAT Problems

9.3.1 CSAT

Definition. A *literal* is either a variable or a negated variable. A *clause* is the disjunction of one or more literals. An expression is in *conjunctive normal form (CNF)* if it is the conjunction of clauses. *CSAT* is the problem where given an expression in CNF, is it satisfiable.

Theorem 9.4. *Every boolean expression E has an equivalent expression E' where negation only appears in literals. The length of E' is linear in the number of symbols in E and can be constructed in polynomial time.*

Proof. The proof will be done by induction the number of operators in E. Also, E' will have no more than $2n-1$ operators when E has $n \geq 1$ operators. Because E' does not need to have one pair of parentheses per operator, and the number of variables cannot go past one more than the number of operators, we can conclude that the length of E' is linear in the length of E.

Base: E has no operators, in which case $E = x$ is a single variable. In this case, $E' = E$.

Ind: E has $n+1$ operators. Consider the case when $E = E_1 \vee E_2$ where each E_k has n_k operators. Note $n = n_1 + n_2$. By the induction hypothesis, there are equivalent expressions E_1' and E_2' with at most $2n_1-1$ and $2n_2-1$ operators, respectfully. Then $E' = (E_1') \vee (E_2')$ has at most $2n_1-1+2n_2-1+1 = 2(n_1+n_2)-1 = 2n-1$ operators. The case where $E = E_1 \wedge E_2$ is similar.

Now suppose, $E = \neg E_0$ and we need to analyze the various cases for E_0. If $E_0 = x$ is a variable, then $E' = E = \neg x$. If $E_0 = E_1 \vee E_2$, then $E = \neg(E_1 \vee E_2)$ which is equivalent to $(\neg(E_1) \wedge \neg(E_2))$. As $\neg(E_1)$

and $\neg(E_2)$ have fewer operators than E, there are, by the induction hypothesis, equivalent E_1' and E_2' which push the negation onto the literals. Let $E' = E_1' \wedge E_2'$. Finally, the case where $E_0 = E_1 \wedge E_2$ is similar. We leave it to the reader to show that the number of operators in E' is at most twice those in E minus one. □

Theorem 9.5. *Given a boolean expression E, there is another expression E' in CNF such that E is satisfiable if and only if E' is.*

Proof. Given a boolean expression, first apply theorem 9.4 to it to obtain an expression E where negation is only appears in literals. We will prove this theorem by induction on the number of symbols in E.

Base: There are one or two symbols in E, in which case E is a literal and already in CNF form. Therefore, $E' = E$.

Ind: Assume that every expression shorted than E can be converted into a CNF form. Note there are only two cases here: $E = E_1 \wedge E_2$ and $E = E_1 \vee E_2$, because the negations have been pushed down to the literals.

Case 1: $E = E_1 \wedge E_2$. By the induction hypothesis, we have that there are expressions E_1' and E_2' in CNF. Further, exactly the satisfying assignments for E_k can be extended to a satisfying assignment for E_k'. Without loss of generality, we can assume that of the variables not in E, the other variables of E_1' and E_2' can be made to be distinct. Then $E' = E_1' \wedge E_2'$ is clearly in CNF. We must show that for any truth assignment ϕ for E can be extended into a satisfying assignment ϕ' for E' if and only if ϕ satisfies E.

(If) Given an assignment ϕ that satisfies E, let $\phi\restriction_{E_k}$ be the assignment restricted to E_k. By the induction hypothesis, we can extend $\phi\restriction_{E_k}$ to a satisfying assignment for E_k'. As the extra variables introduced in the E_k' are distinct, we can combine the extended assignments into one ϕ' which will satisfy $E' = E_1' \wedge E_2'$.

(Only if) Given an extension ϕ' to an assignment ϕ which satisfies E', let $\phi\restriction_{E_k}$ be the restriction of ϕ to E_k, and let $\phi'\restriction_{E_k'}$ be the restriction of ϕ' to E_k'. Then $\phi'\restriction_{E_k'}$ is an extension of $\phi\restriction_{E_k}$. Since E' is the conjunction of E_1' and E_2' it must follow that E_k' is satisfied by $\phi'\restriction_{E_k'}$. By the induction hypothesis, we have that $\phi\restriction_{E_k}$ must satisfy E_k. And by our definitions, we have that ϕ satisfies E.

Case 2: $E = E_1 \vee E_2$. The induction hypothesis has that E_k has a corresponding E_k' in CNF. And as in case 1, we may assume that the extra variables in E_1' are disjoint from the extra variables in E_2'. Suppose $E_1' = c_1 \wedge \cdots \wedge c_{n_1}$ and $E_2' = d_1 \wedge \cdots \wedge d_{n_2}$ where the c_ks and d_ks are clauses. Introduce a new variable x and let

$$E' = \left(\bigwedge_{k=1}^{n_1} (x \vee c_k) \right) \wedge \left(\bigwedge_{k=1}^{n_2} (\neg x \vee d_k) \right).$$

Now we must prove that for any truth assignment ϕ for E such that ϕ satisfies E then ϕ can be extended to a satisfying truth assignment for E'.

(If) Given a truth assignment ϕ that extends to an assignment ϕ' that satisfies E', we have to consider two cases depending on the truth value ϕ' assigns to x. Suppose, without loss of generality, that $\phi'(x)$ is false. Because ϕ' applied to E' yields true, we must have that ϕ' makes all the clauses true as well. Further, ϕ' must make each c_k true as $\phi'(x)$ is false. Thus, ϕ' has to satisfy E_1'. As $\phi'\restriction_{E_1'}$ is an extension of $\phi\restriction_{E_1}$, our induction hypothesis states that $\phi\restriction_{E_1}$ must satisfy E_1. Therefore, ϕ is required to satisfy E. The case where $\phi'(x)$ is true is analogous.

(Only if) Given an assignment ϕ which satisfies E, we have two cases to deal with as E is a disjunction. Without loss of generality, assume that ϕ satisfies E_1. Then $\phi\restriction_{E_1}$ satisfies E_1, and our induction hypothesis guarantees that some extension $\phi'\restriction_{E_1'}$ satisfies E_1'. Make ϕ' an extension to E' with $\phi'(x)$ being false. Then ϕ' satisfies E'. □

Theorem 9.6. *CSAT is NP-complete.*

Proof. We will show the above construction of theorem 9.5 can be done in quadratic time. Let $T(n)$ be the number of moves required to convert E into E' where n is the length of E. For the base cases of E being a literal, $T(1)$ and $T(2)$, nothing needs to be done other than read E. We will let a be the upper bound for these two cases. The inductive cases have us breaking up E into E_1 and E_2, converting each E_k to E_k', then piecing together E_1' and E_2'. The first and last steps can be done in time proportional to n, namely in bn

time for some constant b. For the recursive, middle step, assume E_1 has length j, and so E_2 will have length $n-1-j$. The time this takes will be equal to $T(j)+T(n-1-j)$. This gives us the bounded recursive relation below.

$$\begin{aligned} T(1), T(2) &\leq a \\ T(n) &\leq bn + \max_{0<i<n-1}(T(j)+T(n-1-j)) \text{ for } n \geq 3 \end{aligned}$$

We will determine a value of c in order to get $T(n) \leq cn^2$. For the base cases $n=1,2$ we just need to get c to be as large as a. Assume by induction that for every m up to n that $T(m) \leq cm^2$. For $0<j<n-1$,

$$\begin{aligned} T(j)+T(n-1-j) &= c\left(j^2+(n-1-j)^2\right) \\ &= c\left(j^2+(n-1)^2-2j(n-1)+j^2\right) \\ &= c\left(2j^2+n^2-2n+1-2jn+2j)\right) \\ &= c\left(n^2-2j(n-j)-2(n-j)+1\right) \\ &\leq c\left(n^2-n-2+1\right) \\ &\leq c\left(n^2-n\right). \end{aligned}$$

Plugging this into the bounded recursive relation, we get that $T(n) \leq bn + cn^2 - cn$. By ensuring c is at least b, we get that $T(n) \leq cn^2$. □

9.3.2 3SAT

Definition. An expression is in *k-conjunctive normal form (kCNF)* if it is the conjunction of clauses that are the disjunctions of up to k literals. *kSAT* is the problem where given an expression in kCNF, is it satisfiable.

Theorem 9.7. *3SAT is NP-complete.*

Proof. Clearly 3SAT is in NP because every instance of 3SAT is an instance of SAT which is in NP. To get NP-completeness, we will reduce CSAT to 3SAT. Given an expression in CNF $E=\bigwedge_{k=1}^{n} e_k$ where each e_k is a clause, we only need to modify those clauses that consist of four or more literals. Suppose some $e_k=\bigvee_{j=1}^{m} x_j$ for some literals x_j and $m \geq 4$. Replace this e_k with the following clauses:

$$(x_1 \vee x_2 \vee y_1) \wedge \bigwedge_{j=3}^{m-2} (x_j \vee \neg y_{j-2} \vee y_{j-1}) \wedge (x_{m-1} \vee x_m \vee \neg y_{m-3})$$

Given an assignment ϕ that satisfies E, ϕ must satisfy one of literals in e_k, say it makes x_j true. Extending ϕ to make $y_1, \ldots, y_{j-2}$ true and $y_{j-1}, \ldots, y_{m-3}$ false will satisfy the new clauses that have been inserted. Now conversely assume that ϕ makes all the x_j false. Then ϕ cannot be extended to make the new clauses true, because there are $m-2$ clauses and each of the $m-3$ ys can only make one clause true. □

9.4 Independent Sets

Definition. In an undirected graph G, a subset I of nodes from G is called an *independent set* if no two nodes in I are connected by an edge in G. An independent set is *maximal* if it is as large as any independent set from the same graph.

Problem Independent Set (IS)

Input A graph G and a lower limit l, which must be between 1 and the number of nodes in G.

Output "Yes"/"No" depending on whether G has an independent set with at least l nodes.

Reduction From 3SAT

Theorem 9.8. *Independent set is NP-complete*

Proof. IS is clearly in NP because a nondeterministic Turing machine would guess at n nodes and verify they form an independent set. Given a 3CNF expression $E = e_1 \wedge \cdots \wedge e_m$ where each e_k represents a clause, we construct from E a graph G with $3m$ nodes and have the bound be m. Name these nodes $[r, s]$ where $r = 1, \ldots, m$ and $s = 1, 2, 3$ and which represent the sth literal in the rth clause. As for the edges, we do the following.

1. For each j, add edges $([r, 1], [r, 2])$, $([r, 1], [r, 3])$, and $[s, 2], [s, 3]$ so that only one node per clause is chosen.

2. Also add an edge between nodes $[r_1, s_1]$ and $[r_2, s_2]$ whenever one represents a variable x and the other represents its negation $\neg x$. This makes it impossible to choose two nodes that represent conflicting literals.

Note if a clause contains just 1 or 2 literals, for example is just the literal α, then we can repeat literals as in $\alpha \vee \alpha \vee \alpha$. This will be fine for our purposes.

Given an independent set I for G with m nodes, it is clear from our construction that no two distinct nodes can come from the same clause. This is because $[r, s_1]$ and $[r, s_2]$, with $s_1 \neq s_2$, have an edge between them due to (1). Consequently, we have chosen exactly one node from every clause. From (2), we cannot have two nodes in I which represent some variable x and its negation $\neg x$. Construct a truth assignment ϕ from I as follows: if a node in I corresponds to x, take $\phi(x)$ to be true; and if a node in I corresponds to $\neg x$, take $\phi(x)$ to be false. Otherwise, there is no node in I associated with the variable in question, we can just make ϕ arbitrary for that variable. This choice of ϕ is a satisfying truth assignment for E.

Given a satisfying truth assignment ϕ for E, ϕ must make all the clauses of E true as well. Choose one literal per clause that make ϕ true, then take the corresponding nodes into a set I which we will prove is an independent set. The edges that fully connect a clause will not be connecting any nodes of I, as we choose exactly one node from each clause. Because ϕ cannot make both x and $\neg x$ true, the edges that connect negated literals will not cause us trouble either. As these are the only edges, we conclude that I is an independent set.

The construction of G from E should not take more than quadratic time in the length of E. This makes IS an NP-complete problem. □

9.5 Directed Hamiltonian-Circuit

Definition. In a graph, a *Hamiltonian circuit* or a *Hamiltonian cycle* is a loop that goes through all the nodes exactly once.

Problem Directed Hamiltonian-Circuit problem (DHC)

Input A directed graph G

Output "Yes"/"No" depending on whether there is a directed Hamiltonian circuit

Claim 9.9. *Directed Hamiltonian-circuit problem is NP-complete.*

9.6 Undirected Hamiltonian-Circuit

Problem Undirected Hamiltonian-Circuit problem (HC)

Input An undirected graph G

Output "Yes"/"No" depending on whether there is a Hamiltonian circuit

Reduction from Directed Hamiltonian circuit

Theorem 9.10. *Undirected Hamiltonian-Circuit problem is NP-complete.*

Proof. Given a directed graph G_d, we will construct an undirected graph G_u with three times as many nodes. For every node $x \in G_d$, there are three nodes $x^{(0)}, x^{(1)}, x^{(2)} \in G_u$ and edges $\left(x^{(0)}, x^{(1)}\right)$ and $\left(x^{(1)}, x^{(2)}\right)$ in G_u. For every edge $x \to y$ in G_d, there is an edge $\left(x^{(2)}, y^{(0)}\right)$ in G_u. The construction of G_u from G_d can be done in polynomial time.

Given a directed Hamiltonian cycle $x_1 \to x_2 \to \cdots \to x_n \to x_1$, there is an undirected Hamiltonian cycle

$$x_1^{(0)}, x_1^{(1)}, x_1^{(2)}, x_2^{(0)}, x_2^{(1)}, x_2^{(2)}, x_3^{(0)}, \ldots, x_n^{(2)}, x_1^{(0)}.$$

Given an undirected Hamiltonian cycle, note that the $x^{(1)}$ nodes have exactly two edges and must appear in the cycle surrounded by $x^{(0)}$ and $x^{(2)}$. So the superscripts in the cycle must either go $0, 1, 2, 0, 1, 2, \ldots$ or $2, 1, 0, 2, 1, 0, \ldots$. As these two options represent going around the graph in opposite directions, we may take either. We shall pick the $0, 1, 2, \ldots$ direction, which corresponds to a directed Hamiltonian cycle in G_d. ☐

9.7 Traveling Salesman Problem

Problem Traveling Salesman Problem (TSP)

Input An undirected, graph G with integer weights on its edges and an upper limit L

Output "Yes"/"No" depending on whether there is a Hamiltonian cycle in G whose total weight is at most L.

Reduction from Hamiltonian-circuit problem

Theorem 9.11. *Traveling Salesman Problem is NP-complete.*

Proof. Given a graph G, construct G' which is the same as G just with edges given an equal weight of 1. With the limit being the n number of nodes of G'. Then a Hamiltonian circuit of G exists if and only if there is one in G' with weight n. ☐