Theory of Computation

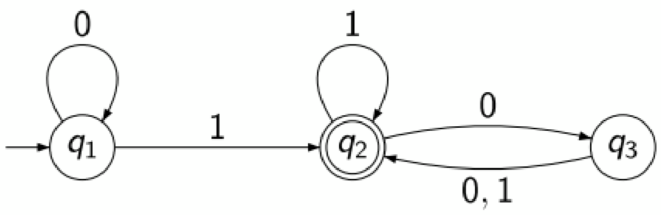
Regular Languages

Deterministic Finite Automata

A finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $q_0 \in Q$ is the start state, and
- $F \subseteq Q$ are the accept states.

Here δ is a total function (defined on all of the domain).



The automaton M_1 (above) can be described precisely as

$M_1 = (\{q_1, q_2, q_3\}, \{0, 1\}, \delta, q_1, \{q_2\})$ with

δ	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

$$L(M_1) = \left\{ w \;\middle|\; \begin{array}{l} w \text{ contains at least one 1, and an} \\ \text{even number of 0s follow the last 1} \end{array} \right\}$$

Let $M = (Q, \Sigma, \delta, q_0, F)$ and let $w = v_1 v_2 \cdots v_n$ be a string from Σ^*.

M accepts w iff there is a sequence of states $r_0, r_1, \ldots, r_n$, with each $r_i \in Q$, such that

1. $r_0 = q_0$
2. $\delta(r_i, v_{i+1}) = r_{i+1}$ for $i = 0, \ldots, n-1$
3. $r_n \in F$

'M recognises language A' is equivalent to $A = \{w \mid M \text{ accepts } w\}$.

Let A and B be languages. The regular operations are:

- **Union:** $A \cup B$
- **Concatenation:** $A \circ B = \{xy \mid x \in A, y \in B\}$
- **Kleene star:** $A^* = \{x_1 x_2 \cdots x_k \mid k \geq 0, \text{ each } x_i \in A\}$

Note that the empty string, ϵ, is always in A^*.

Let $A = \{aa, abba\}$ and $B = \{a, ba, bba, bbba, \ldots\}$.

$A \cup B = \{a, aa, abba, ba, bba, bbba, \ldots\}$.

$A \circ B = \{aaa, abbaa, aaba, abbaba, aabba, abbabba, \ldots\}$.

$$A^* = \left\{ \begin{array}{l} \epsilon, aa, abba, aaaa, aaabba, abbaaa, abbaabba, \\ aaaaaa, aaaaabba, aaabbaaa, aaabbaabba, \ldots \end{array} \right\}.$$

Non-Deterministic Finite Automata

For an alphabet Σ, let Σ_ϵ denote $\Sigma \cup \{\epsilon\}$.

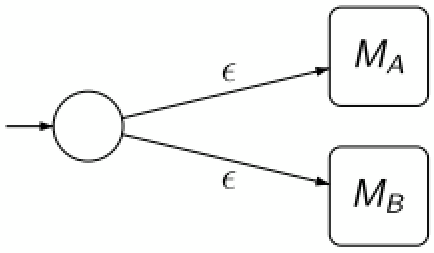
An NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is the transition function,
- $q_0 \in Q$ is the start state, and
- $F \subseteq Q$ are the accept states.

NFAs may also be allowed to move from one state to another without consuming input.

Such a transition is an ϵ transition (JFLAP calls it a λ transition).

Amongst other things, this is useful for constructing a machine to recognise the union $A \cup B$ of two languages:



where M_A and M_B are recognisers for A and B, respectively.

Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and let $w = v_1 v_2 \cdots v_n$ where each v_i is a member of Σ_ϵ.

N accepts w iff there is a sequence of states $r_0, r_1, \ldots, r_n$, with each $r_i \in Q$, such that

1. $r_0 = q_0$
2. $r_{i+1} \in \delta(r_i, v_{i+1})$ for $i = 0, \ldots, n-1$
3. $r_n \in F$

'N recognises language A' is equivalent to $A = \{w \mid N \text{ accepts } w\}$.

Constructing DFAs from NFAs

The class of languages recognised by NFAs is exactly the class of regular languages. Every NFA has an equivalent DFA.

1. Create a new state for each possible combination of initial states, so 2^n new states in total
2. The new start state will be the state corresponding to the old start state, plus any states it could epsilon-transition to
3. The new accept states will be any of the new states which include the original accept state
4. The transition functions are obtained just by looking at what the original machine did at each state and for each possible symbol. In this example, state {3} goes to state {1,3} on input a, since it can also epsilon-transitions back to 3.

$$\{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}.$$

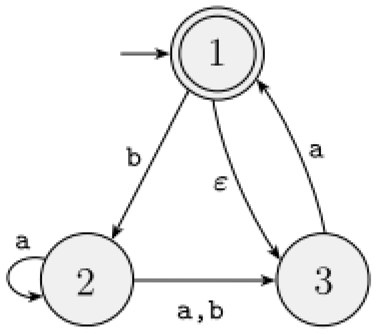


FIGURE 1.42
The NFA N_4

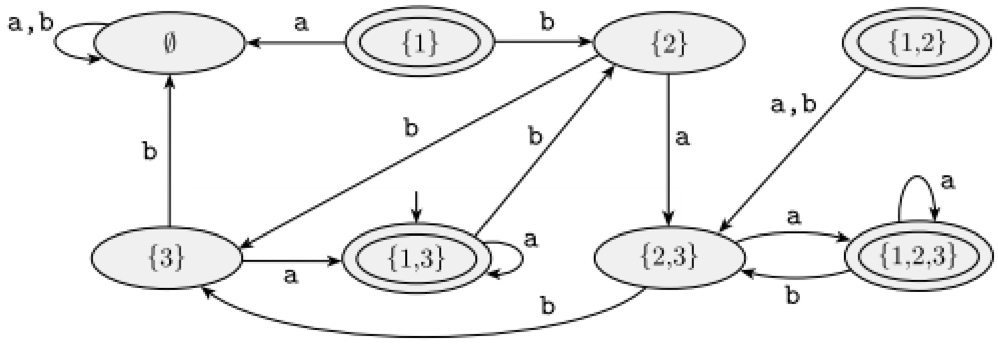


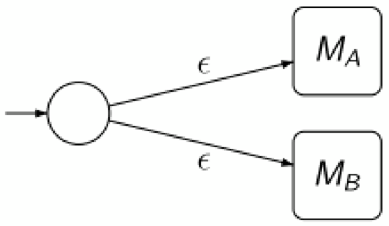
FIGURE 1.43
A DFA D that is equivalent to the NFA N_4

Note that epsilon transitions by themselves do not count for anything in the DFA. They only allow you to move extra after having made some initial transition.

Closure Properties

Theorem: The class of regular languages is closed under union.

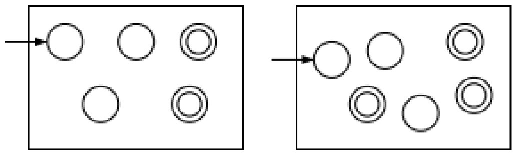
Proof: Let A and B be regular languages, with recognisers M_A and M_B. An NFA that recognises $A \cup B$ is easily constructed:



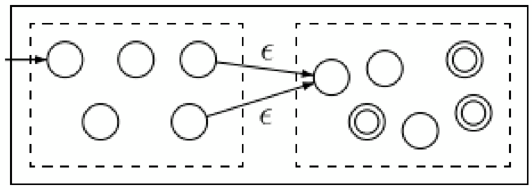
The ϵ-transitions go to the start states of M_A and M_B.

Theorem: The class of regular languages is closed under $\circ$.

Proof: Let A and B be regular languages with these recognisers:

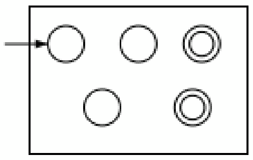


From these we can easily construct an NFA that recognises $A \circ B$:

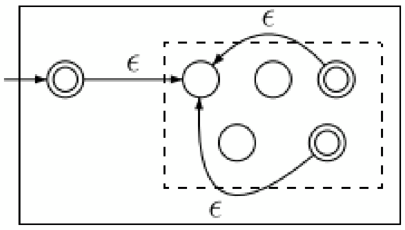


Theorem: The class of regular languages is closed under Kleene star.

Proof: Let A be a regular language with the recogniser shown on the right.



Here is how we construct an NFA to recognise A^*:



Regular Expressions

Regular expressions constitute a notation for languages. The semantics are fairly simple:

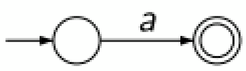
$$
\begin{array}{lcl}
L(a) & = & \{a\} \\
L(\epsilon) & = & \{\epsilon\} \\
L(\emptyset) & = & \emptyset \\
L(R_1 \cup R_2) & = & L(R_1) \cup L(R_2) \\
L(R_1\ R_2) & = & L(R_1) \circ L(R_2) \\
L(R^*) & = & L(R)^*
\end{array}
$$

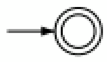
Some examples of regular expressions:

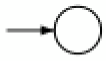
110	:	$\{110\}$
$(\Sigma\Sigma)^*$	:	all strings of even length
$(0 \cup \epsilon)(\epsilon \cup 1)$	:	$\{\epsilon, 0, 1, 01\}$
1^*	:	all sequences of 1s
$\epsilon \cup 1 \cup (\epsilon \cup 1)^*(\epsilon \cup 1)$	:	all sequences of 1s

Constructing Regex to NFA

Theorem: L is regular iff L can be described by a regular expression. The construction algorithm is fairly simple.

Case $R = a$: Construct 

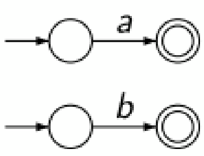
Case $R = \epsilon$: Construct 

Case $R = \emptyset$: Construct 

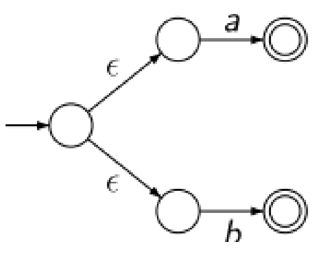
Case $R = R_1 \cup R_2$, $R = R_1\ R_2$, or $R = R_1^*$:
We already gave the constructions when we showed that regular languages were closed under the regular operations.

$(a \cup b)^* bc$.

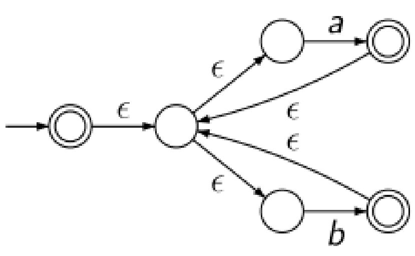
Start from innermost expressions and work out:



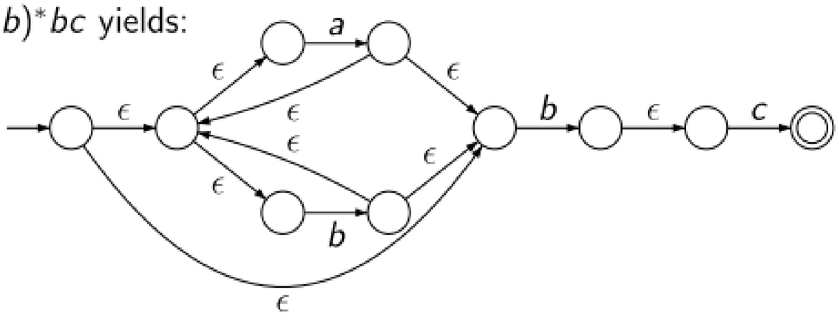
So $a \cup b$ yields:



Then $(a \cup b)^*$ yields:



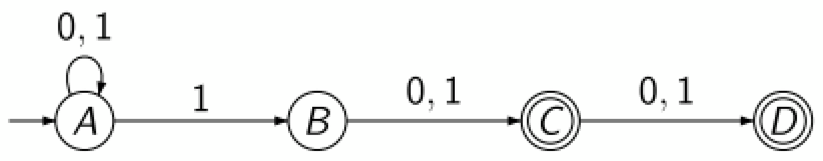
Finally $(a \cup b)^* bc$ yields:



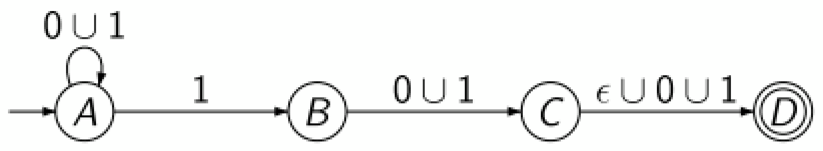
Converting NFA to Regex

An NFA can be turned into a regular expression in a systematic process of “state elimination”.

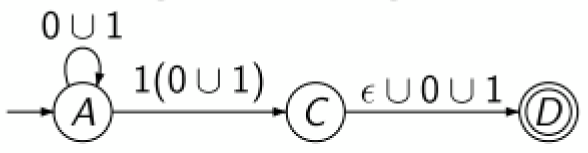
- If there are several accept states, we can reduce that to just one, by introducing epsilon-transitions from the “no-longer” accept states to one remaining accept state
- Then proceed by eliminating states one by one, bypassing then with an appropriate regex



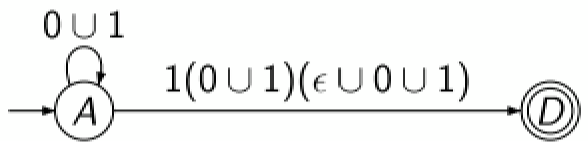
Keep just one accept state (and use regular expressions with all arcs):



Now eliminate B:



and then C:



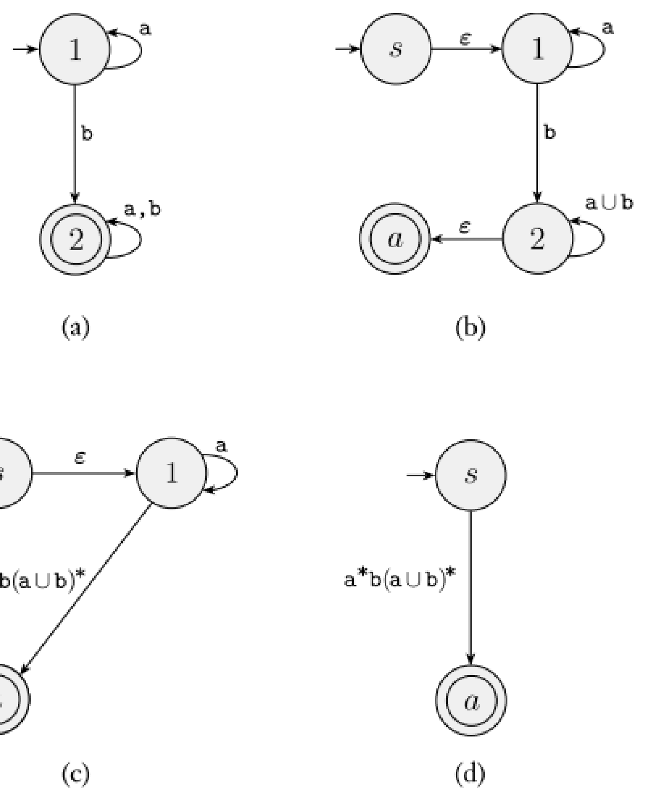
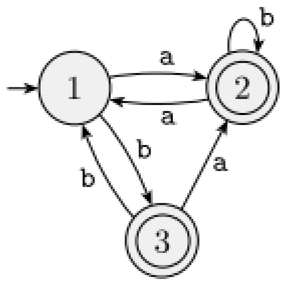
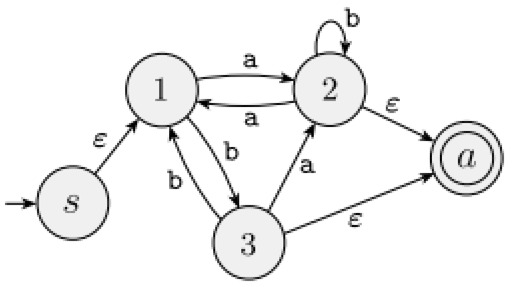


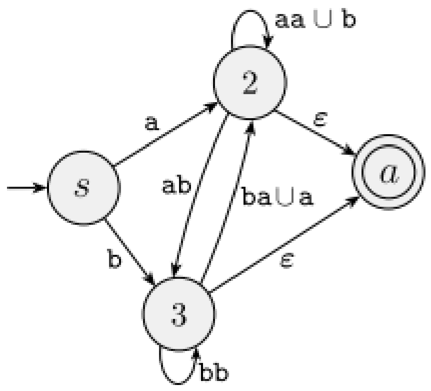
FIGURE 1.67
Converting a two-state DFA to an equivalent regular expression



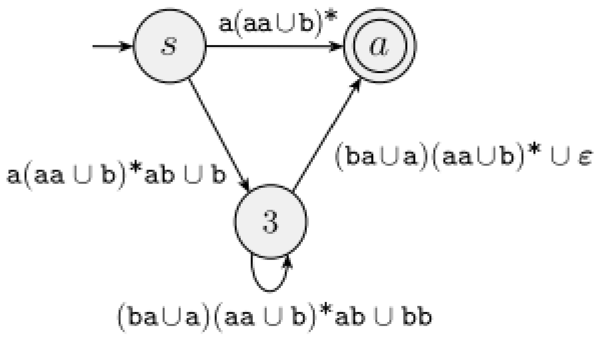
(a)



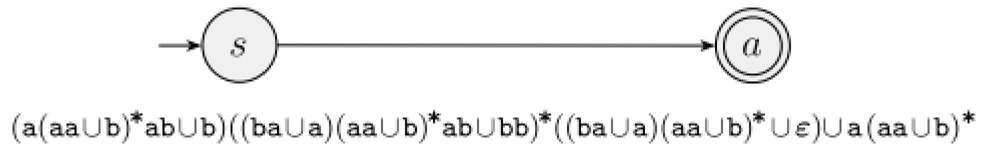
(b)



(c)



(d)


s
a
(a(aa∪b)*ab∪b)((ba∪a)(aa∪b)*ab∪bb)*((ba∪a)(aa∪b)* ∪ε)∪a(aa∪b)*

Pumping Lemma for Regular Languages

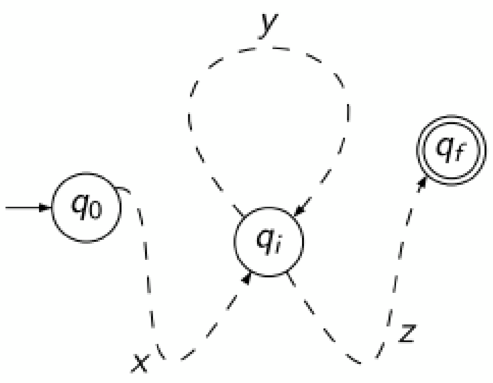
This is our main tool for proving languages non-regular. Loosely, it says that if we have a regular language A and consider a sufficiently long string s, then a recogniser for A must traverse some loop to accept s. So A must contain infinitely many strings exhibiting repetition of some substring in s.

Pumping Lemma: If A is regular then there is a number p such that for every string $s \in A$ with $|s| \geq p$, s can be written as $s = xyz$, satisfying

1. $xy^iz \in A$ for all $i \geq 0$
2. $y \neq \epsilon$
3. $|xy| \leq p$

Let DFA $M = \{Q, \Sigma, \delta, q_0, F)$ recognise A. Let the number of states of M be p, let $|s| \geq p$.

Some state q_i must be visited more than once. Consider the first two times q_i (on the right) is visited. This suggests a way of splitting s into x, y and z such that $xz, xyz, xyyz, \ldots$ are all in A.



Pumping Example 1

We show that $B = \{0^n 1^n \mid n \geq 0\}$ is not regular.

Assume it is, and let p be the pumping length.

Consider $0^p 1^p \in B$ with length greater than p.

By the pumping lemma, $0^p 1^p = xyz$, with $xy^i z$ in B for all $i \geq 0$.

But y cannot consist of all 0s, since $xyyz$ then has more 0s than 1s.

Similarly y cannot consist of all 1s. And if y has at least one 0 and one 1, then some 1 comes before some 0 in $xyyz$.

Pumping Example 3

We show that $D = \{ww \mid w \in \{0,1\}^*\}$ is not regular.

Assume it is, and let p be the pumping length.

Consider $0^p10^p1 \in D$ with length greater than p.

By the pumping lemma, $0^p10^p1 = xyz$, with xy^iz in D for all $i \geq 0$, $y \neq \epsilon$, and $|xy| \leq p$.

Since $|xy| \leq p$, y consists entirely of 0s.

But then $xyyz \notin D$, a contradiction.

Example 4 – Pumping Down

We show that $E = \{0^i1^j \mid i > j\}$ is not regular.

Assume it is, and let p be the pumping length.

Consider $0^{p+1}1^p \in E$.

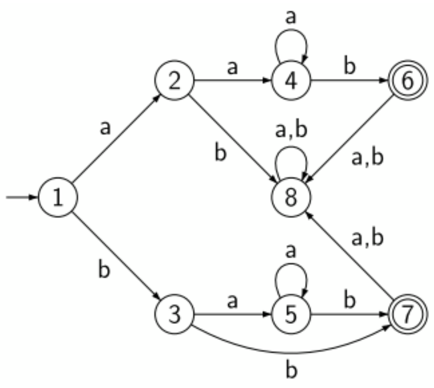
By the pumping lemma, $0^{p+1}1^p = xyz$, with xy^iz in E for all $i \geq 0$, $y \neq \epsilon$, and $|xy| \leq p$.

Since $|xy| \leq p$, y consists entirely of 0s.

But then $xz \notin E$, a contradiction.

Minimizing DFAs

It is always possible to find a minimal DFA (with the smallest number of states) for a given regular language. This also gives us a way of testing the equivalence of two DFAs: just test whether their minimal versions are identical (apart from the names chosen for states). This takes quadratic time.



Initial Partition $\pi_0 = \{A, B\}$ where $A = \{1,2,3,4,5,8\}, B = \{6,7\}$.

Choose A with symbol 'b'. We have

From state	1	2	3	4	5	8
To state	3	8	7	6	7	8
Now in set	A	A	B	B	B	A

so our new partition is $\pi_1 = \{A_1, A_2, B\}$ where $A_1 = \{1, 2, 8\}$, $A_2 = \{3, 4, 5\}$, $B = \{6, 7\}$.

Now choose A_1 with symbol 'b'.

From state	1	2	8
To state	3	8	8
Now in set	A_2	A_1	A_1

so our new partition is $\pi_2 = \{A_{11}, A_{12}, A_2, B\}$ where $A_{11} = \{1\}$, $A_{12} = \{2, 8\}$, A_2, B as before.

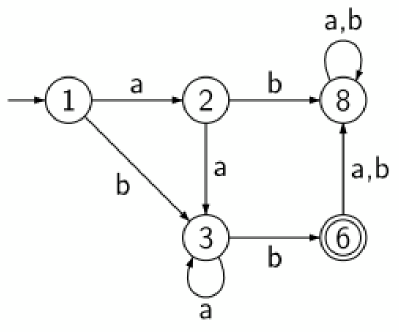
Now choose A_{12} with 'a'.

From state	2	8
To state	4	8
Now in set	A_2	A_{12}

giving us a final partition $\pi' = \{\{1\}, \{2\}, \{8\}, \{3, 4, 5\}, \{6, 7\}\}$.

It is final because every other choice of partition element with a symbol does not further partition the states.

The minimized DFA looks like this:



Note how the original states 3, 4, and 5 have collapsed into one (the state now labeled 3).

Context Free Languages

Context Free Grammars

We have already used the formalism of context-free grammars. To specify the syntax of regular expressions we gave a grammar, much like

$$\begin{array}{lcl} R & \to & 0 \\ R & \to & 1 \\ R & \to & \epsilon \\ R & \to & \emptyset \\ R & \to & R \cup R \\ R & \to & R\ R \\ R & \to & R^* \end{array}$$

Hence a grammar is a set of substitution rules, or productions. We have the shorthand notation

$$R \to 0 \mid 1 \mid \epsilon \mid \emptyset \mid R \cup R \mid R\ R \mid R^*$$

A is called a variable. Other symbols (here 0 and 1) are terminals. We refer to a valid string of terminals (such as 000111111) as a sentence. The intermediate strings that mix variables and terminals are sentential forms.

A context-free grammar (CFG) G is a 4-tuple (V, Σ, R, S), where

1. V is a finite set of variables,
2. Σ is a finite set of terminals,
3. R is a finite set of rules, each consisting of a variable (the left-hand side) and a sentential form (the right-hand side),
4. S is the start variable.

Closure Properties

The class of CFLs is closed under

- union,
- concatenation,
- Kleene star,
- reversal.

They are not closed under intersection!

Consider these two CFLs:

$$\begin{aligned} E &= \{a^m b^n c^n \mid m, n \in \mathbb{N}\} \\ F &= \{a^n b^n c^m \mid m, n \in \mathbb{N}\} \end{aligned}$$

Exercise: Prove that they are context-free!

But $E \cap F$ is the language $B = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ which we just proved not to be context-free.

Parse Trees and Ambiguity

Consider the grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid 0 \mid 1 \mid \ldots \mid 9$$

This grammar allows not only different derivations, but different parse trees for 3 + 7 ∗ 2:

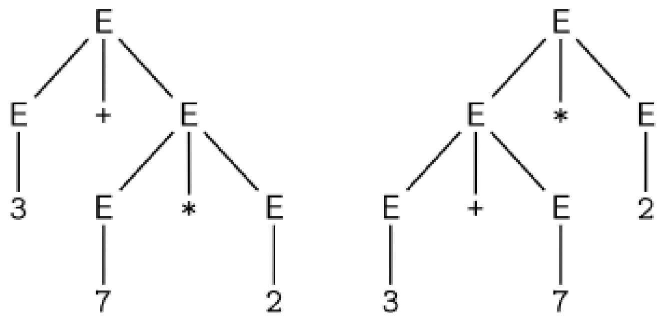
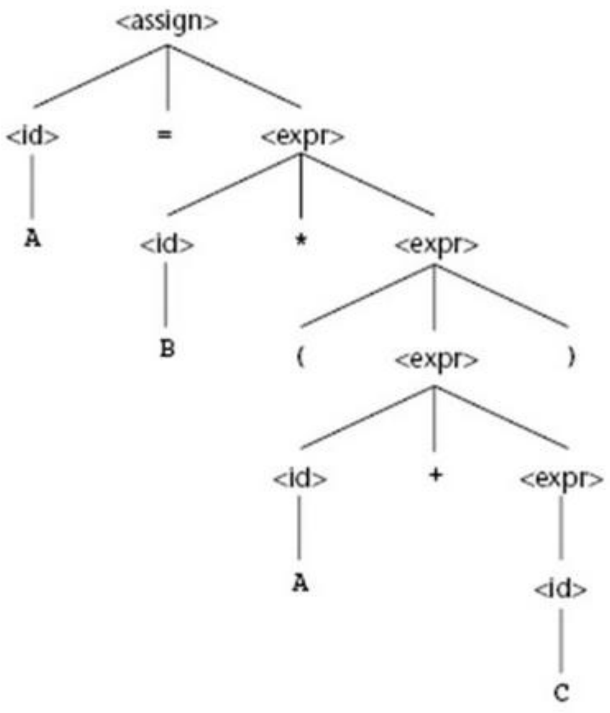


Figure 3.1

A parse tree for the simple statement `A = B * (A + C)`



A grammar that has different parse trees for some sentence is ambiguous.

Sometimes we can find a better grammar (as in our example) which is not ambiguous, and which generates the same language.

However, this is not always possible: There are CFLs that are inherently ambiguous.

Chomsky Normal Form

A simple normal form is Chomsky normal form where every rule is of one of these forms:

$$\begin{aligned} A &\rightarrow B\ C \\ A &\rightarrow a \\ S &\rightarrow \epsilon \end{aligned}$$

where S is the start variable, A may be the start variable, B and C are (non-start) variables, and a is a terminal.

Theorem: Every context-free language has a context-free grammar in Chomsky normal form.

The method for converting a grammar to Chomsky normal form is this:

1. Add a new start variable S_0 and rule $S_0 \to S$.
2. Eliminate epsilon rules $A \to \epsilon$.
3. Eliminate unit rules $A \to B$.
4. Eliminate useless symbols.
5. Ensure that right-hand sides with length greater than 1 consist of variables only.
6. Break right-hand sides of length 3 or more into several rules by introducing fresh variables.

1. The original CFG G_6 is shown on the left. The result of applying the first step to make a new start variable appears on the right.

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
 & & & \boldsymbol{S_0} & \boldsymbol{\rightarrow} & \boldsymbol{S} \\
S & \rightarrow & ASA \mid \texttt{a}B & S & \rightarrow & ASA \mid \texttt{a}B \\
A & \rightarrow & B \mid S & A & \rightarrow & B \mid S \\
B & \rightarrow & \texttt{b} \mid \varepsilon & B & \rightarrow & \texttt{b} \mid \varepsilon
\end{array}
$$

2. Remove ε-rules $B \rightarrow \varepsilon$, shown on the left, and $A \rightarrow \varepsilon$, shown on the right.

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
S_0 & \rightarrow & S & S_0 & \rightarrow & S \\
S & \rightarrow & ASA \mid \texttt{a}B \mid \mathbf{a} & S & \rightarrow & ASA \mid \texttt{a}B \mid \texttt{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \mid \boldsymbol{S} \\
A & \rightarrow & B \mid S \mid \boldsymbol{\varepsilon} & A & \rightarrow & B \mid S \mid \varepsilon \\
B & \rightarrow & \texttt{b} \mid \varepsilon & B & \rightarrow & \texttt{b}
\end{array}
$$

3a. Remove unit rules $S \rightarrow S$, shown on the left, and $S_0 \rightarrow S$, shown on the right.

$$
\begin{array}{rcl@{\qquad}rcl}
S_0 & \rightarrow & S & S_0 & \rightarrow & S \mid \boldsymbol{ASA} \mid \mathbf{a}\boldsymbol{B} \mid \mathbf{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \\
S & \rightarrow & ASA \mid \texttt{a}B \mid \texttt{a} \mid SA \mid AS \mid S & S & \rightarrow & ASA \mid \texttt{a}B \mid \texttt{a} \mid SA \mid AS \\
A & \rightarrow & B \mid S & A & \rightarrow & B \mid S \\
B & \rightarrow & \texttt{b} & B & \rightarrow & \texttt{b}
\end{array}
$$

3b. Remove unit rules $A \rightarrow B$ and $A \rightarrow S$.

$$
\begin{array}{rcl@{\qquad}rcl}
S_0 & \rightarrow & ASA \mid \texttt{a}B \mid \texttt{a} \mid SA \mid AS & S_0 & \rightarrow & ASA \mid \texttt{a}B \mid \texttt{a} \mid SA \mid AS \\
S & \rightarrow & ASA \mid \texttt{a}B \mid \texttt{a} \mid SA \mid AS & S & \rightarrow & ASA \mid \texttt{a}B \mid \texttt{a} \mid SA \mid AS \\
A & \rightarrow & B \mid S \mid \mathbf{b} & A & \rightarrow & S \mid \texttt{b} \mid \boldsymbol{ASA} \mid \mathbf{a}\boldsymbol{B} \mid \mathbf{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \\
B & \rightarrow & \texttt{b} & B & \rightarrow & \texttt{b}
\end{array}
$$

4. Convert the remaining rules into the proper form by adding additional variables and rules. The final grammar in Chomsky normal form is equivalent to G_6. (Actually the procedure given in Theorem 2.9 produces several variables U_i and several rules $U_i \rightarrow \texttt{a}$. We simplified the resulting grammar by using a single variable U and rule $U \rightarrow \texttt{a}$.)

$$
\begin{array}{rcl}
S_0 & \rightarrow & AA_1 \mid UB \mid \texttt{a} \mid SA \mid AS \\
S & \rightarrow & AA_1 \mid UB \mid \texttt{a} \mid SA \mid AS \\
A & \rightarrow & \texttt{b} \mid AA_1 \mid UB \mid \texttt{a} \mid SA \mid AS \\
A_1 & \rightarrow & SA \\
U & \rightarrow & \texttt{a} \\
B & \rightarrow & \texttt{b}
\end{array}
$$

■

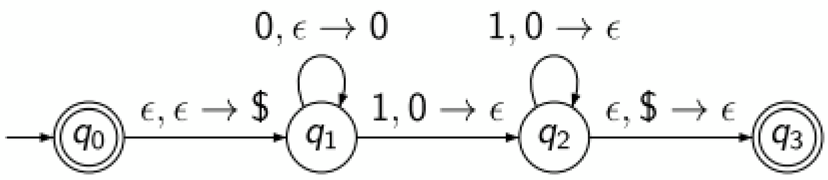
Pushdown Automata

A pushdown automaton (PDA) is a finite-state automaton, equipped with a stack which allows it to store items in memory. We shall consider the non-deterministic version of a PDA. It may, in one step: read a symbol from input and the top stack symbol (which is popped); based on these, and the current state, it transitions to the next state, and pushes a symbol onto the stack. The PDA may choose to leave out the popping, or the pushing, or both. It may also ignore the input.

A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where

- Q is a finite set of states,
- Σ is the finite input alphabet,
- Γ is the finite stack alphabet,
- $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function,
- $q_0 \in Q$ is the start state, and
- $F \subseteq Q$ are the accept states.

This PDA recognises $\{0^n 1^n \mid n \geq 0\}$:



The PDA $(Q, \Sigma, \Gamma, \delta, q_0, F)$ accepts input w iff $w = v_1 v_2 \cdots v_n$ with each $v_i \in \Sigma_\epsilon$, and there are states $r_0, r_1, \ldots, r_n \in Q$ and strings $s_0, s_1, \ldots, s_n \in \Gamma^*$ such that

1. $r_0 = q_0$ and $s_0 = \epsilon$.
2. $(r_{i+1}, b) \in \delta(r_i, v_{i+1}, a)$, $s_i = at$, $s_{i+1} = bt$ with $a, b \in \Gamma_\epsilon$ and $t \in \Gamma_\epsilon^*$.
3. $r_n \in F$.

Note 1: There is no requirement that $s_n = \epsilon$, so the stack may be non-empty when the machine stops (even when it accepts).

Note 2: Trying to pop an empty stack leads to rejection of input, rather than "runtime error".

Note that the accept state takes effect only upon reaching the end of the input.

Equivalence of CFGs and PDA

Theorem: L is context-free iff some PDA recognises L.

Since an NFA is also a PDA (which ignores its stack), we have:

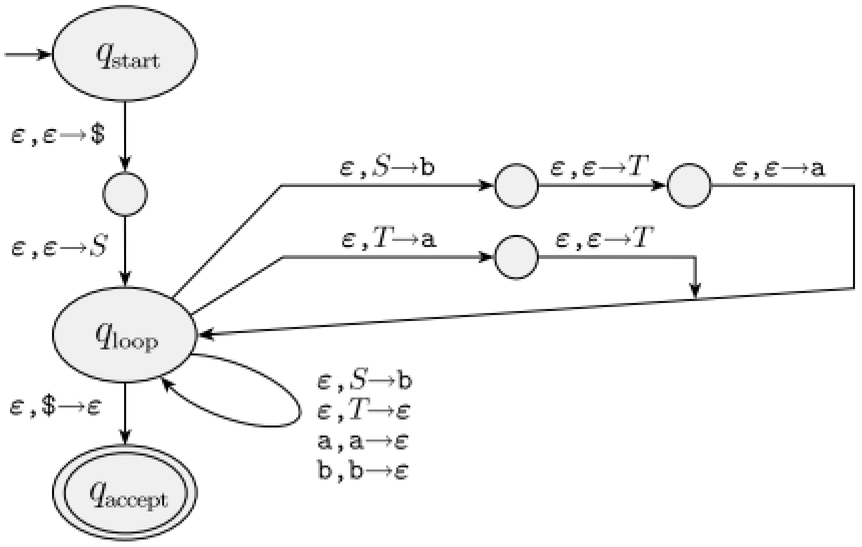
Corollary: Every regular language is context-free.

EXAMPLE 2.25

We use the procedure developed in Lemma 2.21 to construct a PDA P_1 from the following CFG G.

$$S \to \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \to T\mathtt{a} \mid \varepsilon$$

The transition function is shown in the following diagram.



q_{start}
$\varepsilon,\varepsilon\to\$$
$\varepsilon,\varepsilon\to S$
$\varepsilon,S\to\mathtt{b}$
$\varepsilon,\varepsilon\to T$
$\varepsilon,\varepsilon\to\mathtt{a}$
$\varepsilon,T\to\mathtt{a}$
$\varepsilon,\varepsilon\to T$
q_{loop}
$\varepsilon,S\to\mathtt{b}$
$\varepsilon,T\to\varepsilon$
$\mathtt{a},\mathtt{a}\to\varepsilon$
$\mathtt{b},\mathtt{b}\to\varepsilon$
$\varepsilon,\$\to\varepsilon$
q_{accept}

Pumping Lemma for Chomsky Normal Form

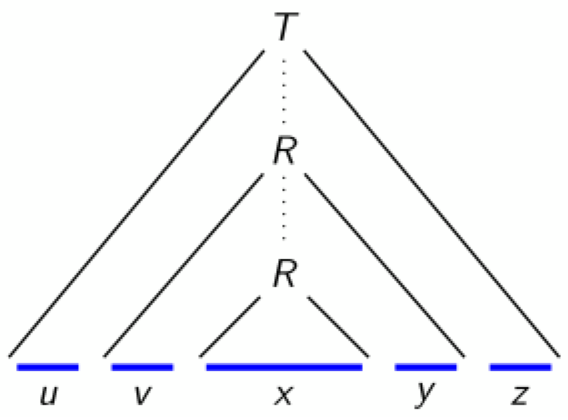
If L is context free then there is a number p such that for every string $s \in L$ with $|s| \geq p$, s can be written as $s = uvxyz$, satisfying

1. $uv^ixy^iz \in L$ for all $i \geq 0$
2. $|vy| > 0$
3. $|vxy| \leq p$

Proof: L has a context-free grammar G in Chomsky Normal Form. The pumping length p is just 2^{N+1}, where N is the number of variables in G.

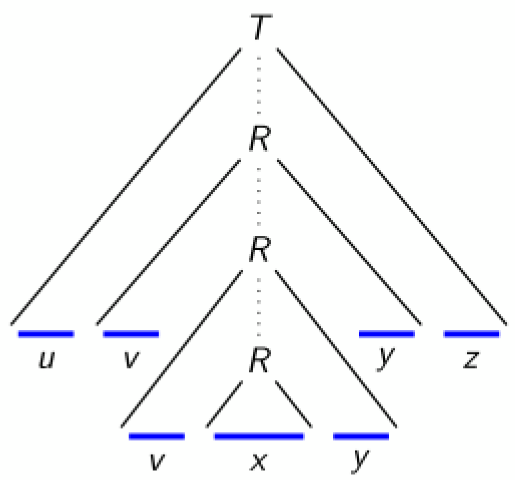
A derivation tree for s must have height at least $|V| + 2$.

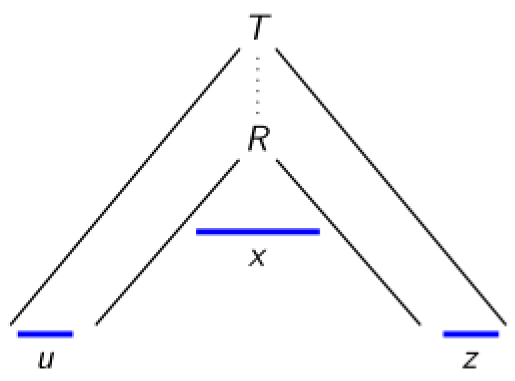
Consider a maximum-depth path in the parse tree for s.



The longest path has $|V| + 1$ variables or more, so some variable must occur repeatedly in that path.

Clearly these are also valid parse trees:





The idea is that we pick a string long enough such that it is high enough so that it must repeat a variable by the pidgeonhole principle. Then we can always attain a new valid parse tree by replacing the subtree from under the first appearance of the repeated variable under the second instance of the repeated variable (as shown above).

We need the longest path to have $V + 1$ variables to ensure the pidgeonhole principle applies. To get this size we need a height of $V + 2$, since the last node is a terminal (not a variable). We set the pumping length to be $p = b^{V+2}$, since this is the maximum length of a string that can be generated by a tree of this height.

Pumping Example 1

$B = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not context free. Assume it is, and let p be the pumping length.

Consider $a^p b^p c^p \in B$ with length greater than p. By the pumping lemma, $a^p b^p c^p = uvxyz$, with $uv^i xy^i z$ in B for all $i \geq 0$.

At least one of v or y is non-empty.

If one of them contains two different symbols from $\{a, b, c\}$ then uv^2xy^2z has symbols in the wrong order, and so cannot be in B.

So both v and y must contain only one kind of symbol. But then uv^2xy^2z can't have the same number of as, bs, and cs.

In all cases, there is a contradiction.

Pumping Example 2

$D = \{ww \mid w \in \{0, 1\}^*\}$ is not context-free. Assume it is, and let p be the pumping length.

Consider $0^p1^p0^p1^p \in D$. By the pumping lemma, $0^p1^p0^p1^p = uvxyz$, with uv^ixy^iz in D for all $i \geq 0$, and $|vxy| \leq p$.

There are three ways that vxy can be part of

$$00...0011...1100...0011...11$$

If it straddles the midpoint, it has form 1^n0^m, so pumping down, we are left with $0^p1^i0^j1^p$, with $i < p$, or $j < p$, or both.

If it is in the first half, uv^2xy^2z will have inserted at least one 1 into the first sequence of 1s. If it is in the second half, the same happens with the 0s.

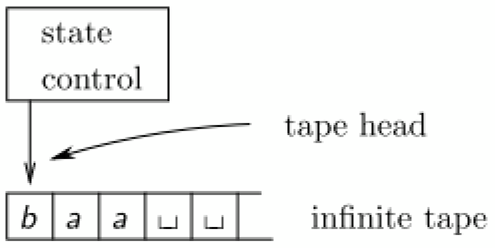
Turing-Recognisable Languages

Turing Machines

A Turing machine has an unbounded tape through which it takes its input and performs its computations.

Unlike a finite automaton, it can

- both read from and write to the tape, and
- move either left or right over the tape.



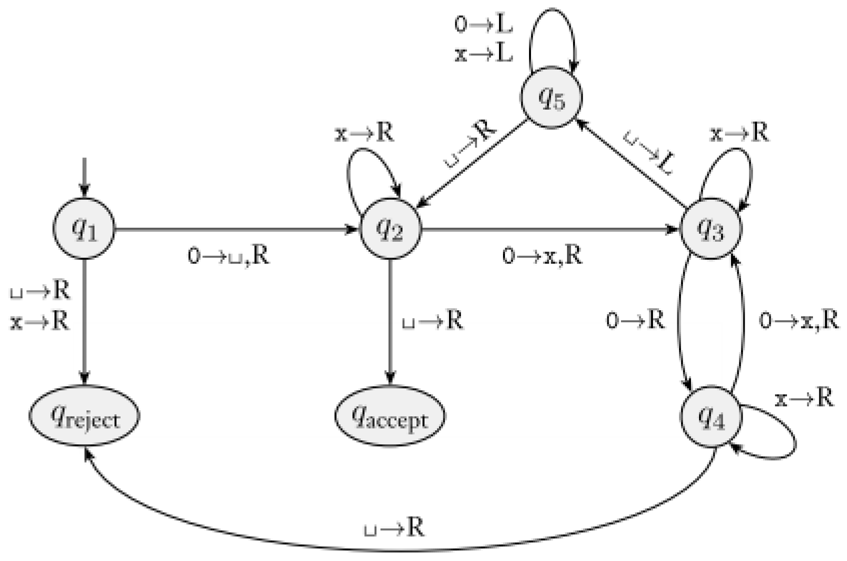
The machine has both accept and reject states, in which it accepts/rejects regardless of tape head location.

A Turing machine is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ where

- Q is a finite set of states,
- Γ is a finite tape alphabet, which includes the blank character, $\sqcup$,
- $\Sigma \subseteq \Gamma \setminus \{\sqcup\}$ is the input alphabet,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
- q_0 is the initial state,
- q_a is the accept state, and
- q_r $(\neq q_a)$ is the reject state.

Now we give the formal description of $M_2 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$:

- $Q = \{q_1, q_2, q_3, q_4, q_5, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0\}$, and
- $\Gamma = \{0, \text{x}, \sqcup\}$.
- We describe δ with a state diagram (see Figure 3.8).
- The start, accept, and reject states are q_1, q_{accept}, and q_{reject}, respectively.



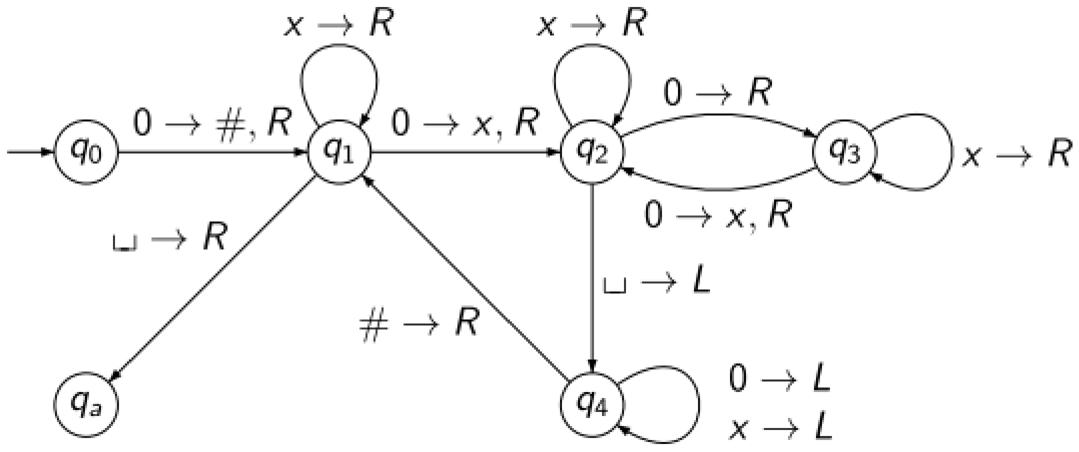
$q_1 0000$
$\sqcup q_2 000$
$\sqcup \mathrm{x} q_3 00$
$\sqcup \mathrm{x}0 q_4 0$
$\sqcup \mathrm{x}0\mathrm{x} q_3 \sqcup$
$\sqcup \mathrm{x}0 q_5 \mathrm{x}\sqcup$
$\sqcup \mathrm{x} q_5 0\mathrm{x}\sqcup$

$\sqcup q_5 \mathrm{x}0\mathrm{x}\sqcup$
$q_5 \sqcup \mathrm{x}0\mathrm{x}\sqcup$
$\sqcup q_2 \mathrm{x}0\mathrm{x}\sqcup$
$\sqcup \mathrm{x} q_2 0\mathrm{x}\sqcup$
$\sqcup \mathrm{xx} q_3 \mathrm{x}\sqcup$
$\sqcup \mathrm{xxx} q_3 \sqcup$
$\sqcup \mathrm{xx} q_5 \mathrm{x}\sqcup$

$\sqcup \mathrm{x} q_5 \mathrm{xx}\sqcup$
$\sqcup q_5 \mathrm{xxx}\sqcup$
$q_5 \sqcup \mathrm{xxx}\sqcup$
$\sqcup q_2 \mathrm{xxx}\sqcup$
$\sqcup \mathrm{x} q_2 \mathrm{xx}\sqcup$
$\sqcup \mathrm{xx} q_2 \mathrm{x}\sqcup$
$\sqcup \mathrm{xxx} q_2 \sqcup$
$\sqcup \mathrm{xxx}\sqcup q_{\mathrm{accept}}$ ■

This machine decides the language $\{0^{2^n} \mid n \geq 0\}$.

It has input alphabet $\{0\}$ and tape alphabet $\{\sqcup, 0, x, \#\}$.



Running the machine on input 000:

$$q_0 000 \Rightarrow \# q_1 00 \Rightarrow \# x q_2 0 \Rightarrow \# x 0 q_3 \sqcup \Rightarrow \# x 0 \sqcup q_r$$

Multi-Tape Turing Machines

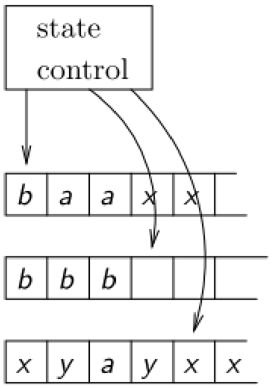
A multitape Turing machine has k tapes. It takes its input on tape 1, other tapes are blank.

The transition function now has type

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$$

and specifies how the k tape heads behave when the machine is in state q_i, reading $a_1, \dots a_k$:

$$\delta(q_i, a_1, \dots, a_k) = (q_j, (b_1, \dots, b_k), (d_1, \dots, d_k))$$



PROOF We show how to convert a multitape TM M to an equivalent single-tape TM S. The key idea is to show how to simulate M with S.

Say that M has k tapes. Then S simulates the effect of k tapes by storing their information on its single tape. It uses the new symbol # as a delimiter to separate the contents of the different tapes. In addition to the contents of these tapes, S must keep track of the locations of the heads. It does so by writing a tape symbol with a dot above it to mark the place where the head on that tape would be. Think of these as "virtual" tapes and heads. As before, the "dotted" tape symbols are simply new symbols that have been added to the tape alphabet. The following figure illustrates how one tape can be used to represent three tapes.

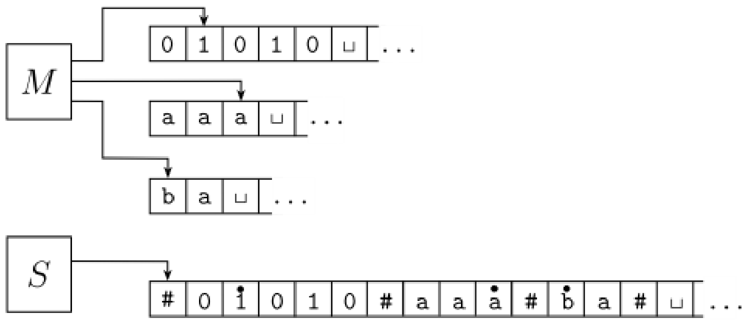


FIGURE 3.14
Representing three tapes with one

S = "On input $w = w_1 \cdots w_n$:

1. First S puts its tape into the format that represents all k tapes of M. The formatted tape contains
$$\#\dot{w}_1 w_2 \cdots w_n \#\dot{␣}\#\dot{␣}\# \cdots \#.$$
2. To simulate a single move, S scans its tape from the first #, which marks the left-hand end, to the $(k+1)$st #, which marks the right-hand end, in order to determine the symbols under the virtual heads. Then S makes a second pass to update the tapes according to the way that M's transition function dictates.
3. If at any point S moves one of the virtual heads to the right onto a #, this action signifies that M has moved the corresponding head onto the previously unread blank portion of that tape. So S writes a blank symbol on this tape cell and shifts the tape contents, from this cell until the rightmost #, one unit to the right. Then it continues the simulation as before."

Non-Deterministic Turing Machines

A nondeterministic Turing machine has a transition function of type

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

If some computation branch leads to *accept* then the machine accepts its input. This is essentially the same type of nondeterminism that NFAs possess.

Theorem: A language is Turing recognisable iff some nondeterministic Turing machine recognises it.

Proof sketch: We need to show that every nondeterministic Turing machine N can be simulated by a deterministic Turing machine D.

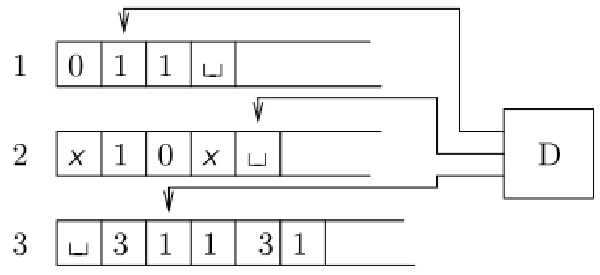
We show how it can be simulated by a 3-tape machine.

Let k be the largest number of choices, according to N's transition function, of all state-symbol combinations.

Tape 1 (always) contains the input.

Tape 3 holds longer and longer sequences from $\{1, \ldots, k\}^*$.

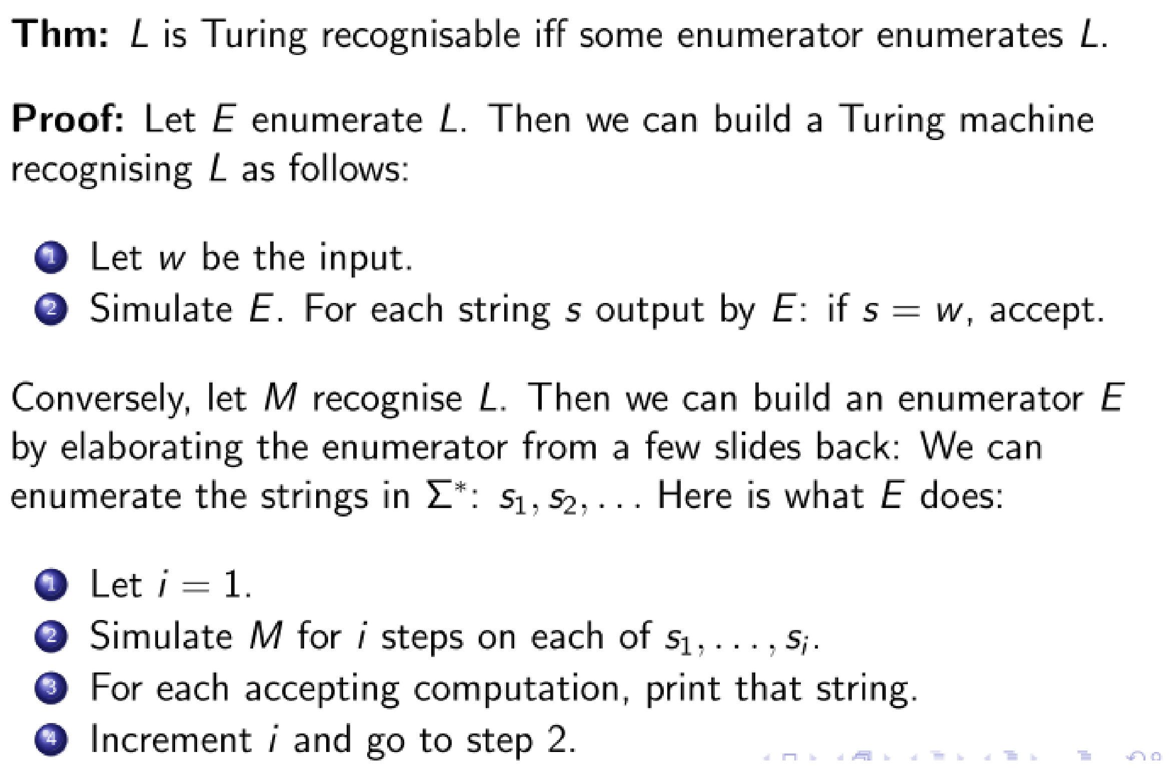
Tape 2 is used to simulate N's behaviour for each fixed sequence of choices given by tape 3.



1. Initially, tape 1 contains input w, and the other two tapes are empty.
2. Overwrite tape 2 by w.
3. Use tape 2 to simulate N. Tape 3 dictates how N should make its choices. If tape 3 gets exhausted, go to step 4. If N says accept, accept.
4. Generate the next "choice" sequence on tape 3. Go to step 2.

Enumerators

For an enumerator to enumerate a language L, for each string in L, it must eventually print the string. It is allowed to print w as often as it wants, and it can print the strings in L in an arbitrary order.



Thm: L is Turing recognisable iff some enumerator enumerates L.

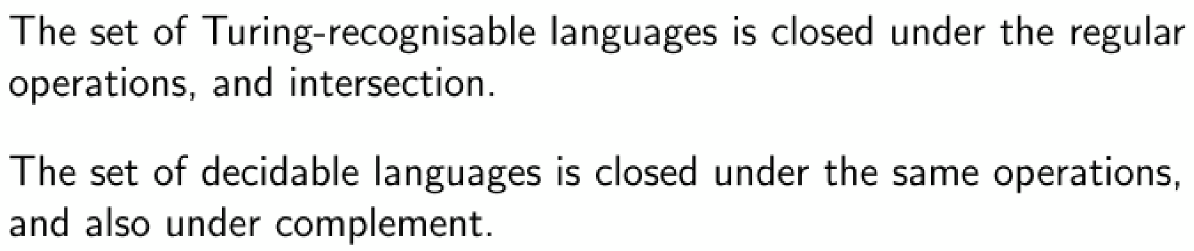
Proof: Let E enumerate L. Then we can build a Turing machine recognising L as follows:

1. Let w be the input.
2. Simulate E. For each string s output by E: if $s = w$, accept.

Conversely, let M recognise L. Then we can build an enumerator E by elaborating the enumerator from a few slides back: We can enumerate the strings in Σ^*: $s_1, s_2, \ldots$ Here is what E does:

1. Let $i = 1$.
2. Simulate M for i steps on each of $s_1, \ldots, s_i$.
3. For each accepting computation, print that string.
4. Increment i and go to step 2.

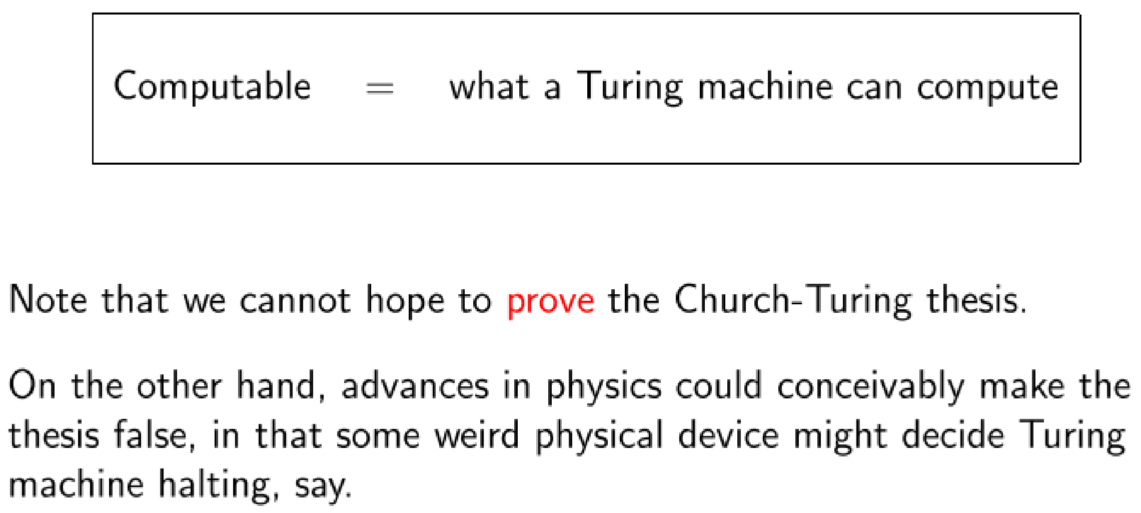
Closure Properties



The set of Turing-recognisable languages is closed under the regular operations, and intersection.

The set of decidable languages is closed under the same operations, and also under complement.

Church-Turing Thesis

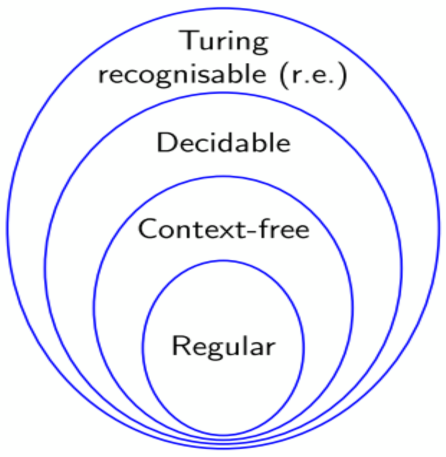


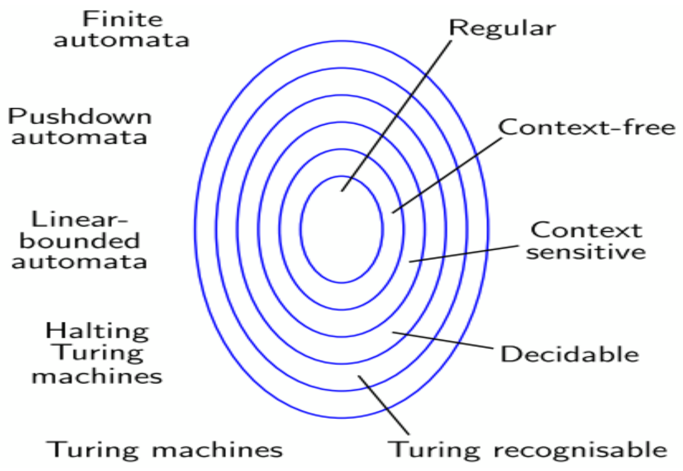
$$\boxed{\text{Computable} \quad = \quad \text{what a Turing machine can compute}}$$

Note that we cannot hope to prove the Church-Turing thesis.

On the other hand, advances in physics could conceivably make the thesis false, in that some weird physical device might decide Turing machine halting, say.

Summary of Languages and Machines





Computability Theory

Decidable Problems

Theorem: A language L is decidable iff both L and $\overline{L}$ are Turing recognisable.

Proof: If L is decidable, clearly L and also $\overline{L}$ are recognisable.

Assume both L and $\overline{L}$ are recognisable. That is, there are recognisers M_1 and M_2 for L and $\overline{L}$, respectively.

A Turing machine M can then take input w and run M_1 and M_2 on w in parallel. If M_1 accepts, so does M. If M_2 accepts, M rejects.

At least one of M_1 and M_2 is guaranteed to accept.

Hence M decides L.

Theorem: A_{DFA} is a decidable language.

Proof sketch: The crucial point is that it is possible for a Turing machine M to simulate a DFA D.

M finds on its tape, say

$$\underbrace{1\ldots n}_{Q}\#\#\underbrace{ab\ldots z}_{\Sigma}\#\#\underbrace{1a2\#\ldots\#nbn}_{\delta}\#\#\underbrace{\ 1\ }_{q_0}\#\#\underbrace{3\ 7}_{F}\#\#\underbrace{baa\ldots}_{w}\$$$

First M checks that the first five components represent a valid DFA, and if not, rejects.

Then M simulates the moves of D, keeping track of D's state and the current position in w, by writing these details on its tape, after \$.

When the last symbol in w has been processed, M accepts if D is in a state in F, and rejects otherwise.

Theorem:

$$E_{DFA} = \{\langle D\rangle \mid D \text{ is a DFA and } L(D) = \emptyset\}$$

is decidable.

Proof sketch: We can design a Turing machine which takes $\langle D\rangle = (Q, \Sigma, \delta, q_0, F)$ as input and performs a reachability analysis:

1. Set $\mathit{reachable} = \{q_0\}$, D's start state.
2. Set $\mathit{new} = \{q \mid \delta(m, x) = q,\ \text{with } m \in \mathit{reachable}\} \setminus \mathit{reachable}$.
3. If $\mathit{new} \neq \emptyset$, set $\mathit{reachable} = \mathit{reachable} \cup \mathit{new}$ and go to step 2.
4. If $\mathit{reachable} \cap F = \emptyset$, accept, otherwise reject.

Theorem:

$$EQ_{DFA} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

is decidable.

Proof sketch: We previously saw how it is possible to construct, from DFAs A and B, DFAs for $A \cap B$, $A \cup B$, and $\overline{A}$.

These procedures are mechanistic and finite—a halting Turing machine M can perform them.

Hence from A and B, M can produce a DFA C to recognise

$$L(C) = \big(L(A) \cap \overline{L(B)}\big) \cup \big(\overline{L(A)} \cap L(B)\big)$$

Note that $L(C) = \emptyset$ iff $L(A) = L(B)$.

Theorem:

$$A_{CFG} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates } w\}$$

is decidable.

Proof: We can put a bound on the number of derivation steps needed to derive each $w \in L(G)$.

First, convert G to an equivalent G' in Chomsky Normal Form. So every rule is of form $A \rightarrow BC$ or $A \rightarrow a$ (disregarding $S' \rightarrow \epsilon$).

Let $n = |w|$. To derive w, we would need to apply a rule of the first form $n - 1$ times, and a rule of the second form n times. So each derivation of w has exactly $2n - 1$ steps.

A Turing machine can thus take G and w, generate G', and list all derivations of length $2n - 1$. It accepts iff one of these generates w.

THEOREM 4.8

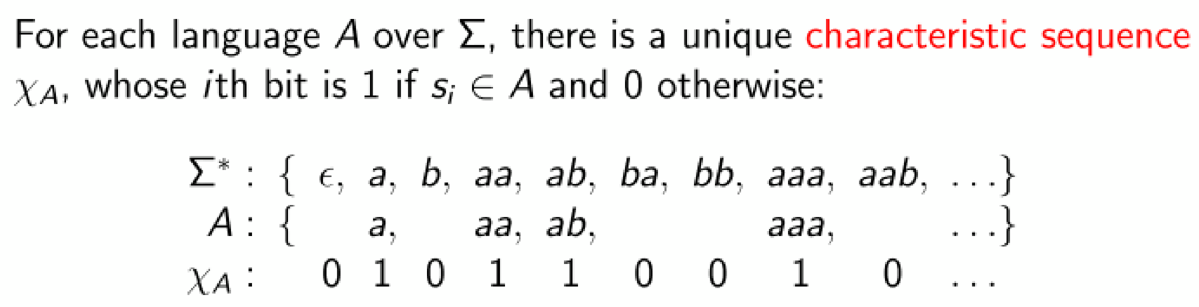
E_{CFG} is a decidable language.

PROOF

```
R = "On input $\langle G\rangle$, where $G$ is a CFG:
    1. Mark all terminal symbols in $G$.
    2. Repeat until no new variables get marked:
    3.     Mark any variable $A$ where $G$ has a rule $A \rightarrow U_1U_2\cdots U_k$ and
           each symbol $U_1, \ldots, U_k$ has already been marked.
    4. If the start variable is not marked, *accept*; otherwise, *reject*."
```

Unrecognisable Languages

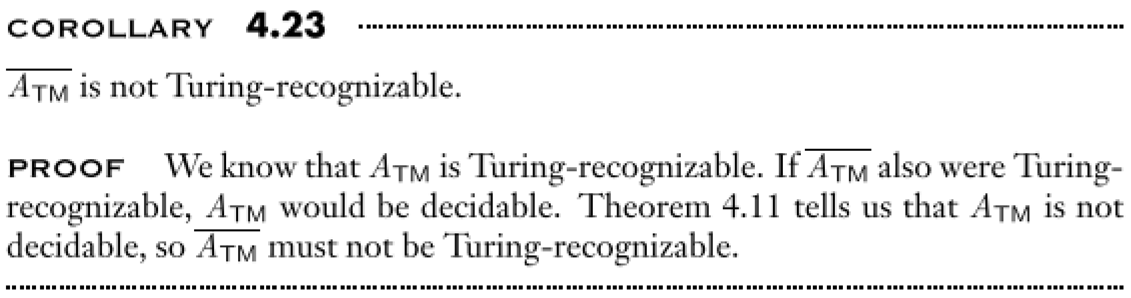
The set of all Turing machines is countable because each Turing machine M has an encoding into a string. If we simply omit those strings that are not legal encodings of Turing machines, we can obtain a list of all Turing machines. To show that the set of all languages is uncountable, we observe that the set of all infinite binary sequences is uncountable:



For each language A over Σ, there is a unique characteristic sequence χ_A, whose ith bit is 1 if $s_i \in A$ and 0 otherwise:

$$\begin{array}{rlcccccccccl}
\Sigma^* : & \{ & \epsilon, & a, & b, & aa, & ab, & ba, & bb, & aaa, & aab, & \ldots\} \\
A : & \{ & & a, & & aa, & ab, & & & aaa, & & \ldots\} \\
\chi_A : & & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & \ldots
\end{array}$$

Thus we have shown that the set of all languages cannot be put into a correspondence with the set of all Turing machines. We conclude that some languages are not recognized by any Turing machine.

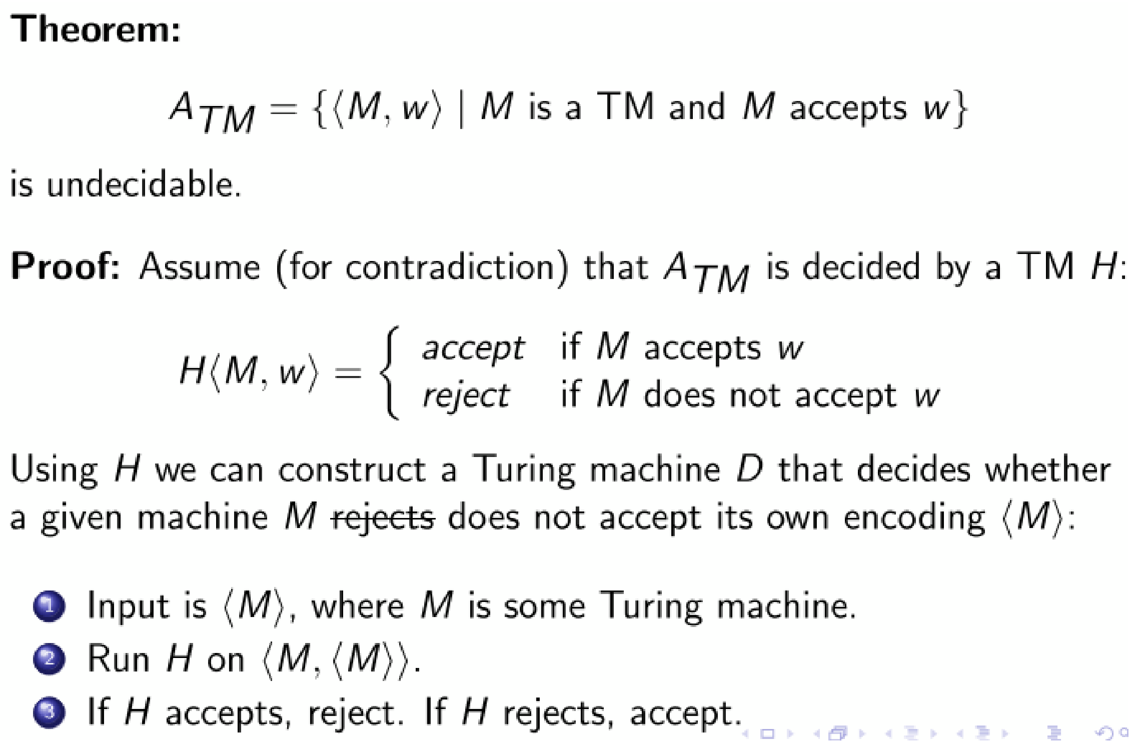


COROLLARY 4.23

$\overline{A_{\mathrm{TM}}}$ is not Turing-recognizable.

PROOF We know that A_{TM} is Turing-recognizable. If $\overline{A_{\mathrm{TM}}}$ also were Turing-recognizable, A_{TM} would be decidable. Theorem 4.11 tells us that A_{TM} is not decidable, so $\overline{A_{\mathrm{TM}}}$ must not be Turing-recognizable.

TM Acceptance is Undecidable



Theorem:

$$A_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

is undecidable.

Proof: Assume (for contradiction) that A_{TM} is decided by a TM H:

$$H\langle M, w\rangle = \begin{cases} \mathit{accept} & \text{if } M \text{ accepts } w \\ \mathit{reject} & \text{if } M \text{ does not accept } w \end{cases}$$

Using H we can construct a Turing machine D that decides whether a given machine M ~~rejects~~ does not accept its own encoding $\langle M\rangle$:

1. Input is $\langle M\rangle$, where M is some Turing machine.
2. Run H on $\langle M, \langle M\rangle\rangle$.
3. If H accepts, reject. If H rejects, accept.

TM Halting is Undecidable

THEOREM 5.1

$HALT_{\mathsf{TM}}$ is undecidable.

PROOF Let's assume for the purpose of obtaining a contradiction that TM R decides $HALT_{\mathsf{TM}}$. We construct TM S to decide A_{TM}, with S operating as follows.

```
S = "On input ⟨M, w⟩, an encoding of a TM M and a string w:
    1. Run TM R on input ⟨M, w⟩.
    2. If R rejects, reject.
    3. If R accepts, simulate M on w until it halts.
    4. If M has accepted, accept; if M has rejected, reject."
```

Clearly, if R decides $HALT_{\mathsf{TM}}$, then S decides A_{TM}. Because A_{TM} is undecidable, $HALT_{\mathsf{TM}}$ also must be undecidable.

TM Emptiness is Undecidable

THEOREM 5.2

E_{TM} is undecidable.

PROOF Let's write the modified machine described in the proof idea using our standard notation. We call it M_1.

```
M_1 = "On input x:
    1. If $x \neq w$, reject.
    2. If $x = w$, run M on input w and accept if M does."
```

This machine has the string w as part of its description. It conducts the test of whether $x = w$ in the obvious way, by scanning the input and comparing it character by character with w to determine whether they are the same.

Putting all this together, we assume that TM R decides E_{TM} and construct TM S that decides A_{TM} as follows.

```
S = "On input ⟨M, w⟩, an encoding of a TM M and a string w:
    1. Use the description of M and w to construct the TM M_1 just
       described.
    2. Run R on input ⟨M_1⟩.
    3. If R accepts, reject; if R rejects, accept."
```

Note that S must actually be able to compute a description of M_1 from a description of M and w. It is able to do so because it only needs to add extra states to M that perform the $x = w$ test.

If R were a decider for E_{TM}, S would be a decider for A_{TM}. A decider for A_{TM} cannot exist, so we know that E_{TM} must be undecidable.

TM Equality is Undecidable

THEOREM 5.4

EQ_{TM} is undecidable.

PROOF IDEA Show that if EQ_{TM} were decidable, E_{TM} also would be decidable by giving a reduction from E_{TM} to EQ_{TM}. The idea is simple. E_{TM} is the problem of determining whether the language of a TM is empty. EQ_{TM} is the problem of determining whether the languages of two TMs are the same. If one of these languages happens to be $\emptyset$, we end up with the problem of determining whether the language of the other machine is empty—that is, the E_{TM} problem. So in a sense, the E_{TM} problem is a special case of the EQ_{TM} problem wherein one of the machines is fixed to recognize the empty language. This idea makes giving the reduction easy.

PROOF We let TM R decide EQ_{TM} and construct TM S to decide E_{TM} as follows.

```
$S =$ "On input $\langle M \rangle$, where $M$ is a TM:
    1. Run $R$ on input $\langle M, M_1 \rangle$, where $M_1$ is a TM that rejects all
       inputs.
    2. If $R$ accepts, *accept*; if $R$ rejects, *reject*."
```

If R decides EQ_{TM}, S decides E_{TM}. But E_{TM} is undecidable by Theorem 5.2, so EQ_{TM} also must be undecidable.

TM Regularity is Undecidable

THEOREM 5.3

$REGULAR_{\mathsf{TM}}$ is undecidable.

PROOF We let R be a TM that decides $REGULAR_{\mathsf{TM}}$ and construct TM S to decide A_{TM}. Then S works in the following manner.

```
$S =$ "On input $\langle M, w \rangle$, where $M$ is a TM and $w$ is a string:
    1. Construct the following TM $M_2$.
       $M_2 =$ "On input $x$:
           1. If $x$ has the form $0^n 1^n$, *accept*.
           2. If $x$ does not have this form, run $M$ on input $w$ and
              *accept* if $M$ accepts $w$."
    2. Run $R$ on input $\langle M_2 \rangle$.
    3. If $R$ accepts, *accept*; if $R$ rejects, *reject*."
```

LBA Emptiness is Undecidable

A linear-bounded automaton (LBA) is a Turing machine which can use only a finite segment of its tape, say, the part occupied by input.

THEOREM 5.9

A_{LBA} is decidable.

PROOF The algorithm that decides A_{LBA} is as follows.

```
L = "On input ⟨M, w⟩, where M is an LBA and w is a string:
    1. Simulate M on w for qng^n steps or until it halts.
    2. If M has halted, accept if it has accepted and reject if it has
       rejected. If it has not halted, reject."
```

If M on w has not halted within qng^n steps, it must be repeating a configuration according to Lemma 5.8 and therefore looping. That is why our algorithm rejects in this instance.

Theorem:

$$E_{LBA} = \{\langle M\rangle \mid M \text{ is an LBA and } L(M) = \emptyset\}$$

is undecidable.

We reduce A_{TM} to E_{LBA}.

For a given M and w, it is possible to construct an LBA B which accepts exactly the accepting computation histories for M on w.

$$\#\underbrace{\qquad\quad}_{C_1}\#\underbrace{\qquad\quad}_{C_2}\#\quad\cdots\quad\#\underbrace{\qquad\quad}_{C_k}\#$$

B must check, in bounded space, that

- C_1 is the start configuration for M on w,
- each C_{i+1} follows legally from C_i,
- C_k is an accepting configuration.

Now for the reduction. Assume that R decides E_{LBA}.

We can decide A_{TM} as follows:

1. Given M and w, construct the LBA B.
2. Run R on $\langle B\rangle$.
3. If R rejects, accept; if it accepts, reject.

But we already know that A_{TM} is undecidable.

CFG ALL is Undecidable

$$ALL_{\mathsf{CFG}} = \{\langle G \rangle \mid G \text{ is a CFG and } L(G) = \Sigma^*\}.$$

THEOREM 5.13

ALL_{CFG} is undecidable.

Rice's Theorem

$$\boxed{\text{Rice's Theorem: Every interesting semantic Turing machine property is undecidable!}}$$

A property P is interesting whenever there exist some Turing machines, M_1 and M_2, for which

$$P(M_1) \text{ and not } P(M_2)\,.$$

It is semantic whenever

$$P(M_1) \text{ iff } P(M_2)$$

for all Turing machines M_1 and M_2 such that $L(M_1) = L(M_2)$.

Mapping Reducibility

Language (or decision problem) A mapping reduces to language B, $A \leq_m B$, iff there is a computable $f : \Sigma^* \to \Sigma^*$ such that

$$w \in A \Leftrightarrow f(w) \in B$$

for all $w \in \Sigma^*$ (we say f is a reduction of A to B).
'f is computable': some Turing machine computes $f(w)$ from each w

Mapping reducibility is the stricter concept;

For example, A_{TM} and $\overline{A_{TM}}$ are Turing reducible to each other (a solution to either can be used to answer the other by negating the answer). But $\overline{A_{TM}}$ is not mapping reducible to A_{TM}.

Complexity Theory

Complexity Classes

Let M be a deterministic Turing machine.

The time complexity of M is the function $t_M : \mathbb{N} \to \mathbb{N}$ defined by

$$t_M(n) = max \left\{ m \;\middle|\; \begin{array}{l} \text{There is some } w \in \Sigma^n \text{ for which} \\ M \text{ takes time } m \text{ to run on } w \end{array} \right\}$$

Let $t : \mathbb{N} \to \mathbb{N}$ be a function.

$$TIME(t(n)) = \left\{ L \;\middle|\; \begin{array}{l} L \text{ is decided by some deterministic Turing} \\ \text{machine } M \text{ with } t_M(n) = O(t(n)) \end{array} \right\}$$

A one-tape Turing machine can decide the language

$$A = \{0^k 1^k \mid k \geq 0\}$$

in $O(n \log n)$ time.

It works by repeatedly scanning across its input, crossing off every second 0 and every second 1, checking after each scan that the number of non-crossed symbols remains even.

A 2-tape deterministic machine can do better.

It can copy all the 1s to its second tape and then match them against the 0s in linear time.

The two kinds of Turing machine have the same computational power, but they have different complexity properties.

The Class P

Definition: P is the class of languages decidable by a deterministic Turing machine in polynomial time, viz.

$$P = \bigcup_k TIME(n^k)$$

Theorem: Every context-free language is in P.

We give an $O(n^3)$ dynamic programming algorithm to decide membership of a CFL L (in Chomsky Normal Form).

The Class NP

For $t(n)$ to be polynomial means to be $O(n^r)$ for some integer r.

Here we take "exponential" to mean $\Omega(2^{cn})$ for some constant $c > 0$.

Then there are functions, such as $2^{\sqrt{n}}$ and $n^{\log n}$ which grow faster than every polynomial function, but more slowly than every exponential function.

A verifier for language A is an algorithm V with

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some } \underbrace{\text{string } c}_{\text{certificate}}\}$$

A polynomial-time verifier runs in time that is polynomial in the length of w.

A is polynomially verifiable if it has a polynomial-time verifier.

Notice that polynomial verifiability need not be closed under complement.

For example, consider $\overline{HAMPATH}$.

There is no obvious (polynomial-length) certificate of the non-existence of a path.

Theorem: A is in NP iff A is decided by some nondeterministic polynomial time Turing machine.

We can also phrase this as

$$NP = \bigcup_k NTIME(n^k)$$

$$NTIME(t(n)) = \left\{ L \;\middle|\; \begin{array}{l} L \text{ is decided by a non-deterministic} \\ \text{Turing machine in } O(t(n)) \text{ time} \end{array} \right\}$$

It is not known whether NP is closed under complement. (Of course, if $P = NP$ then it is.)

The set of languages that are complements of the NP languages is called $co\text{-}NP$.

NP-Completeness

A is polynomial-time (mapping) reducible to B, $\boxed{A \leq_P B}$, iff there is some polynomial-time computable function $f : \Sigma^* \to \Sigma^*$, such that for all w,

$$w \in A \text{ iff } f(w) \in B.$$

We say that f is a polynomial-time reduction from A to B.

B is NP-hard iff *every* $A \in NP$ is reducible to B in polynomial time.

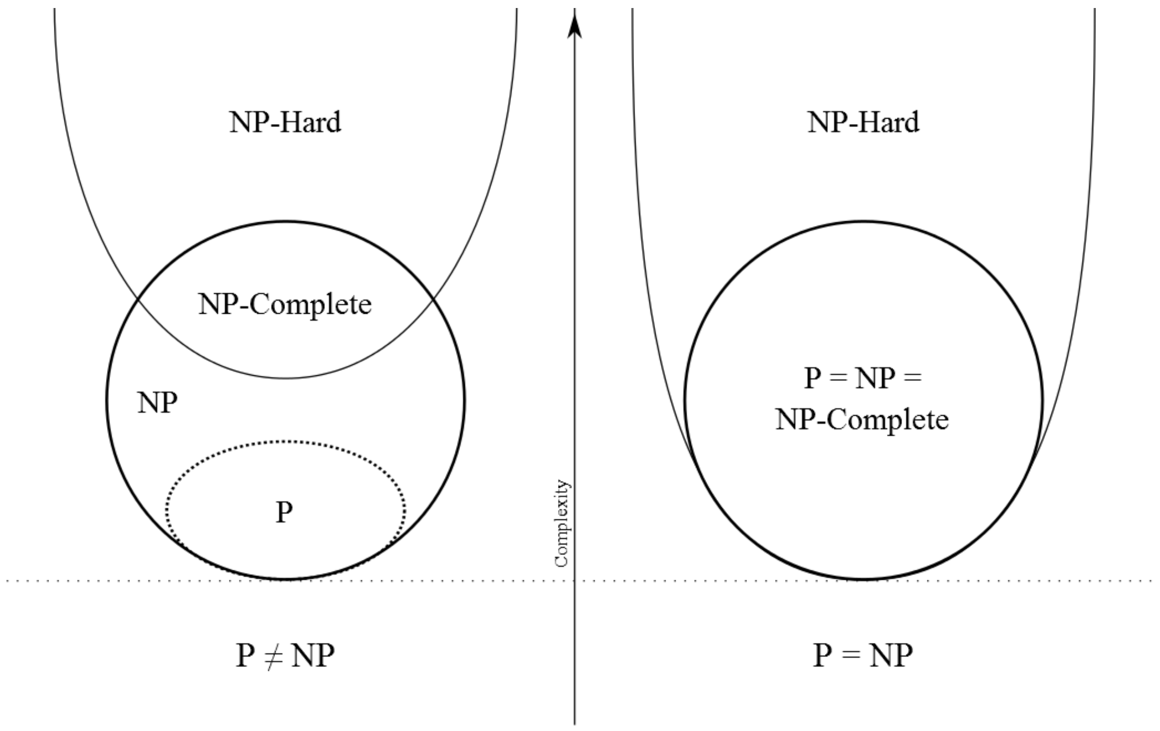
B is NP-complete iff

1. $B \in NP$, and
2. B is NP-hard.

We denote the class of NP-complete languages by NPC.

Many interesting problems have been shown to be NP-hard, without anybody being able to show that they are in NP.

Such a problem may well turn out to require exponential time, even if $P = NP$; indeed it may be undecidable.



NP-Complete Problems

- SAT: Prove by Cook-Levin

  φ is satisfiable if there is some truth assignment to the variables such that the truth value of formula φ is 1.

  SAT is the problem: Given φ in CNF, is φ satisfiable?

  Note that if φ has n variables then there are 2^n truth assignments.

- 3SAT: reduce SAT

  A propositional formula is in 3-CNF if it is in CNF and every clause has exactly three literals (no two of which involve the same variable). An example is

  $$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee \neg x_4)$$

  3SAT is the problem whether a 3-CNF formula is satisfiable.

- Vertex Cover: reduce 3SAT

  $V \subseteq N$ is a vertex cover of G iff $n_1 \in V$ or $n_2 \in V$ for all edges $(n_1, n_2) \in E$.

  We show that $\mathit{VERTEX\text{-}COVER} =$

  $$\{\ \langle G, k \rangle \mid G \text{ is a graph which has a vertex cover of size } k\ \}$$

  is NP-complete.

- Hamiltonian Path: reduce SAT

  Recall the Hamiltonian path problem $\mathit{HAMPATH} =$

  $$\left\{ \langle G, s, t \rangle \;\middle|\; \begin{array}{l} G \text{ is a directed graph with a} \\ \text{Hamiltonian path from } s \text{ to } t \end{array} \right\}$$

  where a Hamiltonian path visits each node in G exactly once.

- INDSET: reduce 3SAT

Given an undirected graph $G = (V, E)$, an independent (vertex) set is a set $S \subseteq V$ satisfying $S^2 \cap E = \emptyset$: no edges inside S.

Decision problem: Does a graph G has an independent set of size k?

$$INDSET = \left\{ \langle G, k \rangle \;\middle|\; \begin{array}{l} G \text{ is an undirected graph with} \\ \text{an independent set of size } k \end{array} \right\}$$

- Subset-Sum: reduce 3SAT

$$SUBSET\text{-}SUM = \{\langle S, t\rangle \mid \Sigma A = t \text{ for some } A \subseteq S\}$$

Closure Properties Summary

Closure Property	**Regular**	**Context-free**	**Decidable**	**Recognisable**
Union $L \cup P$	Yes	Yes	Yes	Yes
Concatenation $L \circ P$	Yes	Yes	Yes	Yes
Kleene star L^*	Yes	Yes	Yes	Yes
Intersection $L \cap P$	Yes	No	Yes	Yes
Set difference $L \backslash P$	Yes	No	Yes	No
Complement $\overline{L}$	Yes	No	Yes	No

Additional Exam Notes

- The intersection of a context-free language and a Regular language is context-free
- Any subset of a finite language is regular
- $(A \setminus B) \cup (A \cap B) = A$
- DNF is disjunction of conjunctions; CNF is conjunction of disjunctions