Models of Computation

Hanno Nickau

Oxford University Computing Laboratory

www.comlab.ox.ac.uk/oucl/work/hanno.nickau/

Course material developed by Luke Ong in 2003

Introduction: Automata, Computability and Complexity

Key question: *What are the capabilities and limitations of computers?*

We seek mathematically precise answers.

Complexity Theory. Easy problem: sorting. Hard problem: scheduling.

What makes some problems computationally hard and others easy?

Computability Theory. *Which problems are solvable by computers and which are not?*

Both Complexity Theory and Computability Theory require a precise definition of a *computer*.

Automata Theory deals with definitions and properties of mathematical models of *computation*.

Finite Automata

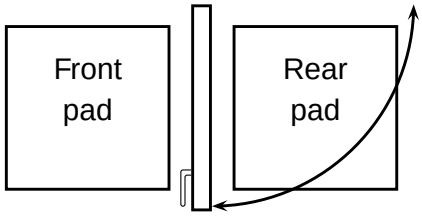
Theory of Computation begins with “What is a model of computation?”

A computational model may be accurate in some ways, but not in others.

We begin with the simplest model: *finite state machines* or *finite automata*.

Finite automata are good models for computers with an *extremely limited amount of memory*. They are nonetheless useful for many things!

Example: A Controller for Automatic Door

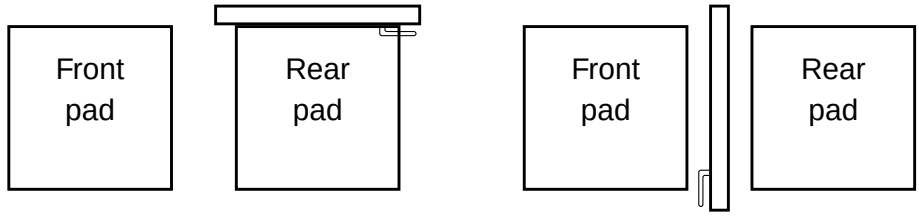


Correct behaviour:

- If a person is on the front pad, the door should open.
- It should remain open long enough for the person to pass all the way through.
- The door should not strike someone standing behind it (i.e. on the rear pad) as it opens.

Example: A Controller for Automatic Door (cont'd)

Two states: OPEN, CLOSED



OPEN *CLOSED*

Four "input conditions":

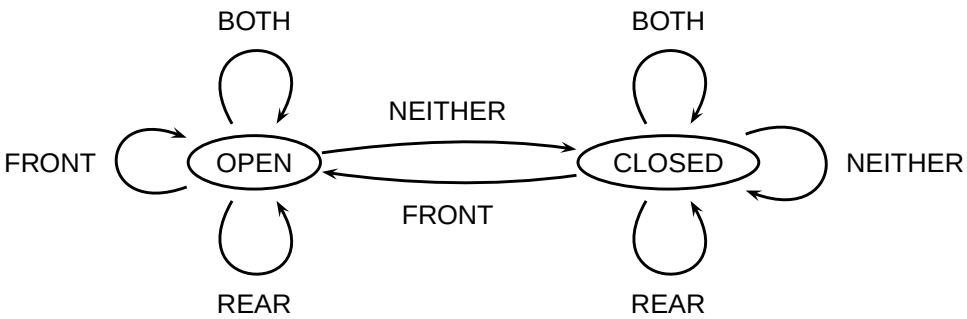
- FRONT: someone standing on front pad only
- REAR: someone standing on rear pad only
- BOTH: people standing on both pads
- NEITHER: no one standing on either pad

Example: A Controller for Automatic Door (cont'd)

State transition table:

	NEITHER	FRONT	REAR	BOTH
CLOSED	CLOSED	OPEN	CLOSED	CLOSED
OPEN	CLOSED	OPEN	OPEN	OPEN

Note: the table can be presented as a (state-transition) graph

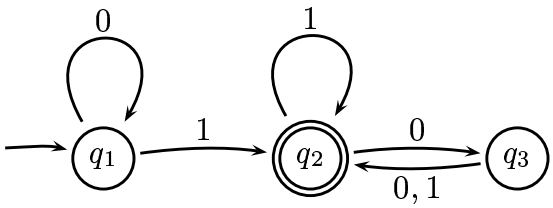


The controller is a rudimentary computer that has just a single bit of memory (for recording state information).

It is an example of a *finite automaton* (or a *finite-state machine*).

Other examples: controllers of dishwashers, electronic thermostats, parts of digital watches and calculators, etc.

Example: a finite automaton M_1



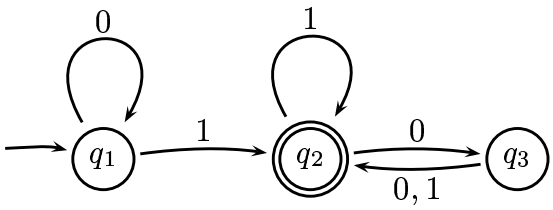
Key Features:

- There are only finitely different *states* a finite automaton can be in.

  The states in M_1 (= vertices of the graph) are q_1, q_2 and q_3.

- We do not care about the internal structure of automaton states. All we care about is which *transitions* the automaton can make between states.

- A symbol from some finite alphabet Σ is associated with each transition: we think of elements of Σ as *input symbols*. The alphabet of M_1 is $\{0, 1\}$.



- Thus all possible transitions can be specified by a *finite directed graph with Σ-labelled edges*.

  E.g. At state q_2, M_1 can

  - input 0 and enter state q_3 i.e. $q_2 \xrightarrow{0} q_3$, or
  - input 1 and remain in state q_2 i.e. $q_2 \xrightarrow{1} q_2$.

- There is a distinguished *start state*. In the graph, the start state is indicated by an arrow pointing at it from nowhere. The start state of M_1 is q_1.

- The states are partitioned into *accepting* states (or final states) and *non-accepting* states.

  An accepting state is indicated by a (double) circle. The accepting state of M_1 is q_2.

Why designate certain states *accepting*?

Notation. We write Σ^* as the set of all *strings* (or words) over Σ i.e.

$$\Sigma^* \stackrel{\text{def}}{=} \{\, a_1 \cdots a_k : a_i \in \Sigma, k \geq 0 \,\}.$$

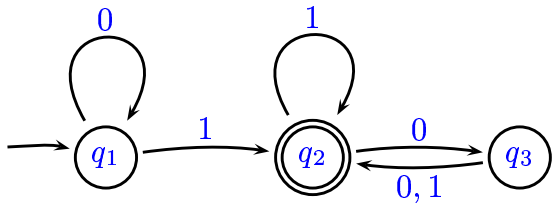
A *language* L is just a set of strings over Σ i.e. $L \subseteq \Sigma^*$.

> We use finite automaton to recognize whether or not a string $u \in \Sigma^*$ is in a particular language (= subset of Σ^*).

Given u, we begin in the start state, and traverse the state-transition graph, using up the symbols in u in the correct order, reading from left to right.

If we can consume all the symbols u in this way and reach an accepting state, then u is in the *language accepted* (or recognized) by the particular automaton; otherwise u is not in the language.

M_1 Revisited



What is the language accepted by M_1?

Answer: all binary strings that contain at least one 1, and an even number of 0s follow the last 1.

When M_1 receives an input string (say) 1101, it processes the string and produces either a "yes" (meaning: the input is accepted) or "no" result.

Beginning at the start state, M_1 receives the symbols from the input string one by one from left to right; after reading each symbol, M_1 moves from one state to another along the transition labelled by that symbol.

After the last symbol is read, M_1 returns "yes" if it is at a final state, and "no" otherwise.

E.g. In processing 1101, M_1 goes through the states q_1, q_2, q_2, q_3, q_2, and returns "yes".

Definition: Deterministic Finite Automaton (DFA)

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

(i) Q is a finite set called the *states*

(ii) Σ is a finite set called the *alphabet*

(iii) $\delta : Q \times \Sigma \to Q$ is the *transition function*

(iv) $q_0 \in Q$ is the *start state*

(v) $F \subseteq Q$ is the *set of accept states* (or *final states*).

We write $q \xrightarrow{a} q'$ to mean $\delta(q, a) = q'$, which we read as "there is an a-*transition* from q to q'".

State-transition graph of a DFA

Equivalently we can represent a DFA by its *state-transition graph*:

- the vertices are just the states
- Σ-labelled edges are the transitions.

Notation:

- The start state is indicated by an arrow pointing at it from nowhere.
- A final state is indicated by a (double) circle; the labelled arrows from one state to another are called *state-transitions*.

Note:

Such a state-transition graph represents a DFA, if for all $a \in \Sigma$ there is exactly one outgoing a-labelled edge from each vertex (state).

Definition: Language accepted by M, $L(M)$

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. $L(M)$, the *language recognized (or accepted) by the DFA* M, consists of all strings $w = a_1 a_2 \cdots a_n$ over Σ satisfying $q_0 \xrightarrow{w}{}^* q$ where q is a final state. Here

$$q_0 \xrightarrow{w}{}^* q$$

means that there exist states $q_1, \cdots, q_{n-1}, q_n = q$ (not necessarily all distinct) such that there are transitions of the form

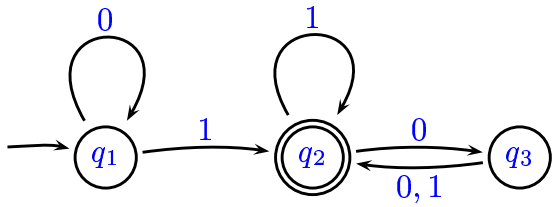
$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_n = q$$

Note

- case $n = 0$: $q \xrightarrow{\epsilon}{}^* q'$ iff $q = q'$
- case $n = 1$: $q \xrightarrow{a}{}^* q'$ iff $q \xrightarrow{a} q'$

A language is called *regular* if some DFA recognizes it.

Example: M_1 Revisited



Formally $M_1 = (Q, \Sigma, \delta, q_1, F)$ where

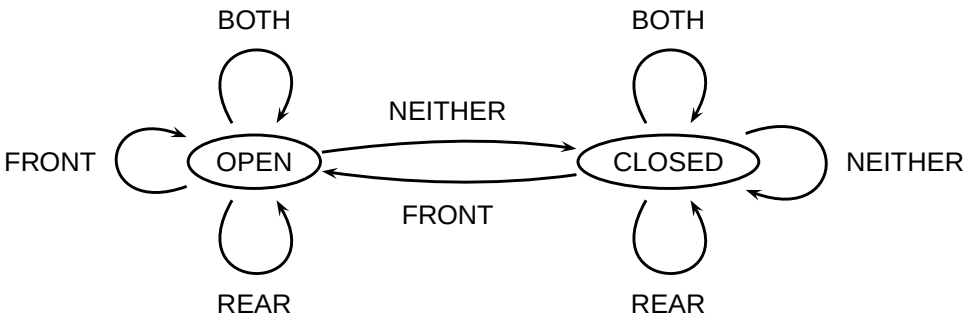
- $Q = \{ q_1, q_2, q_3 \}$
- $\Sigma = \{ 0, 1 \}$
- q_1 is the start state; $F = \{ q_2 \}$
- δ is given by

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

$L(M_1)$ is the set of all binary strings that contain at least one 1, and an even number of 0s follow the last 1.

More examples: automatic door controller

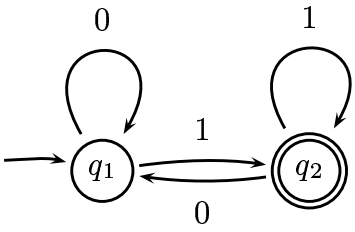
Is the controller for the automatic door a DFA?



So far, the control has no designated start state, or designated accepting states, but otherwise it is a DFA, with

- state set: {OPEN, CLOSED}
- input alphabet: {FRONT, REAR, BOTH, NEITHER}
- transition function: as determined by the state transition table given earlier

More examples: M_2



$M_2 = (\{q_1, q_2\}, \{0, 1\}, \delta, q_1, \{q_2\})$

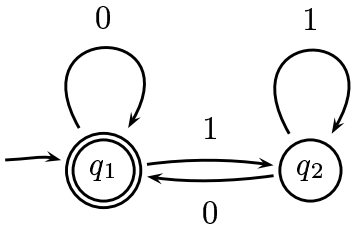
where δ is given by

	0	1
q_1	q_1	q_2
q_2	q_1	q_2

The language recognized by M_2 is

$L(M_2) = \{w \in \{0, 1\}^* : w \text{ ends in a } 1\}$

More examples: M_3



$M_3 = (\{q_1, q_2\}, \{0, 1\}, \delta, q_1, \{q_1\})$

where δ is given by

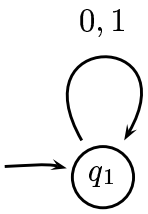
	0	1
q_1	q_1	q_2
q_2	q_1	q_2

The language recognized by M_3 is

$L(M_3) = \{w \in \{0, 1\}^* : w \text{ ends in a } 0\} \cup \{\epsilon\}$

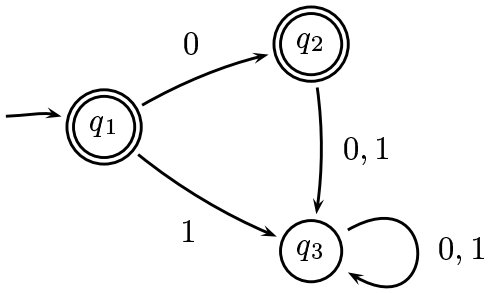
Designing DFAs over input alphabet $\{0, 1\}$

Find M such that $L(M) = \emptyset$.



no final state!

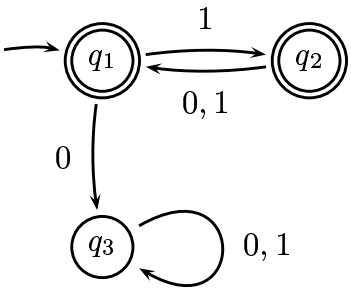
Find M such that $L(M) = \{\epsilon, 0\}$.



Designing DFAs over input alphabet $\{0, 1\}$

Find M such that $L(M) = \{w \in \{0, 1\}^* : \text{every odd position of } w \text{ is a } 1\}$.

E.g. $L(M) = \{\epsilon, 1, 10, 11, 101, 111, 1010, 1011, 1110, 1111, \ldots\}$



The Regular Operations: Union, Concatenation and Star

Let A and B be languages. Define

- **Union**: $A \cup B = \{ x : x \in A \text{ or } x \in B \}$
- **Concatenation**: $A \cdot B = \{ xy : x \in A \text{ and } y \in B \}$
- **Star**: $A^* = \{ x_1 x_2 \cdots x_k : k \geq 0 \text{ and each } x_i \in A \}$.

Note: ϵ (the empty string) is in A^* (the case of $k = 0$)

Example. Take $A = \{ good, bad \}$ and $B = \{ boy, girl \}$.

$A \cdot B = \{ goodboy, goodgirl, badboy, badgirl \}$

$A^* =$
$\{ \epsilon, good, bad, goodgood, goodbad, badgood, badbad, goodgoodgood, \cdots \}$.

Informally $A^* = \{ \epsilon \} \cup A \cup (A \cdot A) \cup (A \cdot A \cdot A) \cup \cdots$.

The Product Construction

Theorem Regular languages are closed under union i.e. if A_1 and A_2 are regular languages, so is $A_1 \cup A_2$.

Proof. Simulate M_1 and M_2 *simultaneously*!

Let $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1, and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize A_2.

We construct $M = (Q, \Sigma, \delta, q_0, F)$ to recognize $A_1 \cup A_2$:

- $Q = Q_1 \times Q_2$ (= $\{ (r_1, r_2) : r_1 \in Q_1, r_2 \in Q_2 \}$)
- $\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$
- $q_0 = (q_1, q_2)$
- $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$ (= $\{ (r_1, r_2) : r_1 \in F_1 \vee r_2 \in F_2 \}$)

We first show $L(M) \subseteq L(M_1) \cup L(M_2)$:

Take $w = a_1 \cdots a_n \in L(M)$. By definition, for some $q_0^1 = q_1, q_1^1, \cdots, q_n^1 \in Q_1$, for some $q_0^2 = q_2, q_1^2, \cdots, q_n^2 \in Q_2$, we have M-transitions

$$(q_0^1, q_0^2) \xrightarrow{a_1} (q_1^1, q_1^2) \xrightarrow{a_2} \cdots\cdots \xrightarrow{a_n} (q_n^1, q_n^2) \tag{1}$$

where $q_n^1 \in F_1$ or $q_n^2 \in F_2$. Suppose the former. Unpacking (**??**), we have M_1-transitions

$$q_0^1 \xrightarrow{a_1} q_1^1 \xrightarrow{a_2} \cdots\cdots \xrightarrow{a_n} q_n^1 \tag{2}$$

I.e. $q_0^1 (= q_1) \xrightarrow{w}{}^* q_n^1 \in F_1$. Hence $w \in L(M_1) \subseteq L(M_1) \cup L(M_2)$.

Next we show $L(M_1) \subseteq L(M)$ (argument for $L(M_2) \subseteq L(M)$ is similar):

Take $w = a_1 \cdots a_n \in L(M_1)$. By definition, for some $q_0^1 = q_1, q_1^1, \cdots, q_n^1 \in Q_1$, with $q_n^1 \in F_1$, we have

$$q_0^1 \xrightarrow{a_1} q_1^1 \xrightarrow{a_2} \cdots\cdots \xrightarrow{a_n} q_n^1 \tag{3}$$

Now since δ_2 is a function, for any $q \in Q_2, a \in \Sigma$, there is a q' s.t. $q \xrightarrow{a} q'$. Hence, there are $q_0^2 = q_2, q_1^2, \cdots, q_n^2 \in Q_2$ s.t.

$$(q_0^1, q_0^2) \xrightarrow{a_1} (q_1^1, q_1^2) \xrightarrow{a_2} \cdots\cdots \xrightarrow{a_n} (q_n^1, q_n^2) \tag{4}$$

are M-transitions. Since $(q_n^1, q_n^2) \in F_1 \times Q_2 \subseteq F$, we have $w \in L(M)$.

□

Remark. Closure under union can be proved (quite simply) using NFAs.

Motivation: Nondeterministic Finite Automata

Theorem Regular languages are closed under concatenation i.e. if A_1 and A_2 are regular languages, so is $A_1 \cdot A_2$.

Proof attempt:

Let $L(M_i) = A_i$. Aim to construct M that accepts w iff w can be broken into w_1 and w_2 (so that $w = w_1 w_2$) whereby M_1 accepts w_1 and M_2 accepts w_2.

But M does not know where to break w into two!

This motivates the introduction of *non-deterministic* finite automata.

References

1. M. D. Davis, R. Sigal and E. J. Weyuker. *Computability, Complexity and Languages*. Academic Press, 2nd edition, 1994. ISBN: 0122063821.
2. J. E. Hopcroft, R. Motwani and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 2nd edition, 2001.
3. D. C. Kozen. *Automata and Computability*. Springer-Verlag, 1997.
4. H. R. Lewis and C. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 2nd edition, 1997.
5. M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, January 1997.
6. T. A. Sudkamp. *Languages and Machines*. Addison-Wesley, 2nd edition, 1996. ISBN: 0201821362.

Non-deterministic Finite Automata (NFA)

NFA *versus* DFA

- In a DFA, at every state q, for every symbol a, there is a unique a-transition i.e. there is a unique q' such that $q \xrightarrow{a} q'$.

  This is not necessarily so in an NFA. At any state, an NFA may have multiple a-transitions, or none.

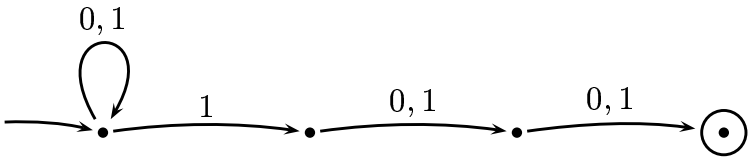
- In a DFA, transition arrows are labelled by symbols from Σ; in an NFA, they are labelled by symbols from $\Sigma \cup \{ \epsilon \}$. I.e. an NFA may have ϵ-transitions.

- We may think of the non-determinism as a kind of parallel computation wherein several processes can be running concurrently.

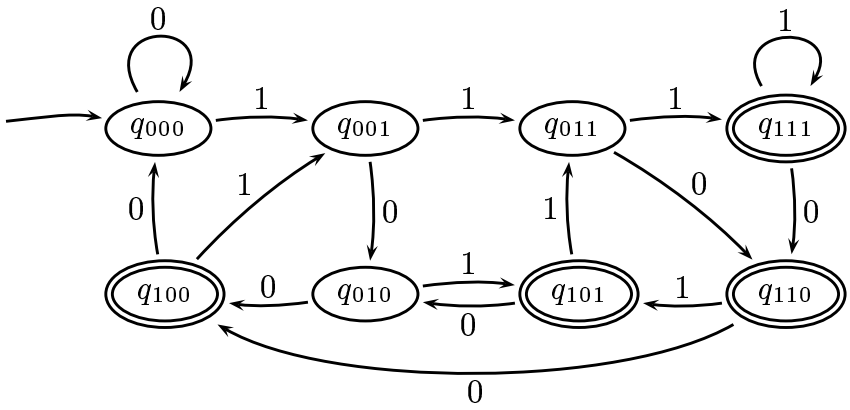
  When the NFA splits to follow several choices, that corresponds to a process "forking" into several children, each proceeding separately. If at least one of these accepts, then the entire computation accepts.

Example: All strings containing a 1 in third position from the end

NFA:

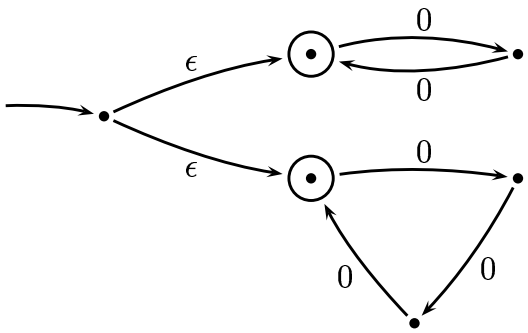


DFA:



NFAs are more compact - they generally require fewer states to recognize a language.

Example: $\{ 0^k : k$ is a multiple of 2 or 3 $\}$



Using ϵ-transitions and non-determinism, a language defined by an NFA can be easier to understand.

Definition: NFA

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

(i) Q is a finite set of states

(ii) Σ is a finite alphabet

(iii) $q_0 \in Q$ is the start state

(iv) $\delta : Q \times (\Sigma \cup \{ \epsilon \}) \to \mathcal{P}(Q)$ is the transition function

(v) $F \subseteq Q$ is the set of final states.

Note: $\mathcal{P}(Q) \stackrel{\text{def}}{=} \{ X : X \subseteq Q \}$ is the *power set* of Q. Equivalently δ can be presented as a relation, i.e. a subset of $(Q \times (\Sigma \cup \{ \epsilon \})) \times Q$.

For $a \in \Sigma \cup \{ \epsilon \}$ we define $q \stackrel{a}{\longrightarrow} q' \stackrel{\text{def}}{=} q' \in \delta(q, a)$.

Some definitions and notations

Fix an NFA $N = (Q, \Sigma, \delta, q_0, F)$.

$L(N)$, the *language accepted by* N, consists of all strings w over Σ satisfying $q_0 \stackrel{w}{\Longrightarrow} q$ where q is a final state. Here $\cdot \stackrel{-}{\Longrightarrow} \cdot$ is defined by:

- $q \stackrel{\epsilon}{\Longrightarrow} q'$ iff $q = q'$ or there is a sequence $q \stackrel{\epsilon}{\longrightarrow} \cdots \stackrel{\epsilon}{\longrightarrow} q'$ of one or more ϵ-transitions in N from q to q'.
- For $w = a_1 \cdots a_{n+1}$ where each $a_i \in \Sigma$, $q \stackrel{w}{\Longrightarrow} q'$ iff there are $q_1, q_1', \cdots, q_{n+1}, q_{n+1}'$ (not necessarily all distinct) such that

$$q \stackrel{\epsilon}{\Longrightarrow} q_1 \stackrel{a_1}{\longrightarrow} q_1' \stackrel{\epsilon}{\Longrightarrow} q_2 \stackrel{a_2}{\longrightarrow} q_2' \stackrel{\epsilon}{\Longrightarrow} \cdots q_n' \stackrel{\epsilon}{\Longrightarrow} q_{n+1} \stackrel{a_{n+1}}{\longrightarrow} q_{n+1}' \stackrel{\epsilon}{\Longrightarrow} q'$$

Intuitively $q \stackrel{w}{\Longrightarrow} q'$ means:

> "There is a sequence of transitions from q to q' in N in which the symbols in w occur in the correct order, but with 0 or more ϵ-transitions before or after each one".

We shall sometimes write $\hat{\delta}(q, w) = \{ q' \in Q : q \stackrel{w}{\Longrightarrow} q' \}$, for $w \in \Sigma^*$.

Note: In case N is a DFA, for any $q \in Q$ and $w \in \Sigma^*$, there is a *unique* q' such that $q \stackrel{w}{\Longrightarrow} q'$ (thus, by abuse of notation, we write $\hat{\delta}(q, w) = q'$).

Exercise. Writing $w = a_1 \cdots a_{n+1}$, we have $q \stackrel{w}{\Longrightarrow} q'$ is equivalent to: there exist $q_1, \cdots, q_n$ such that

$$q \stackrel{a_1}{\Longrightarrow} q_1 \stackrel{a_2}{\Longrightarrow} \cdots\cdots \stackrel{a_{n+1}}{\Longrightarrow} q'$$

Equivalence of NFAs and DFAs: The Subset Construction

Observation. Every DFA *is* an NFA!

Say two automata are *equivalent* if they accept the same language.

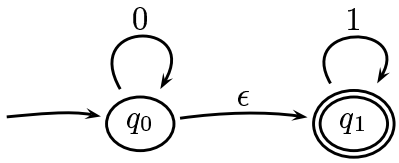
Theorem(Determinization). Every NFA has an equivalent DFA.

Proof. Fix an NFA $N = (Q_N, \Sigma_N, \delta_N, q_N, F_N)$, we construct an equivalent DFA $\mathcal{P}N = (Q_{\mathcal{P}N}, \Sigma_{\mathcal{P}N}, \delta_{\mathcal{P}N}, q_{\mathcal{P}N}, F_{\mathcal{P}N})$ such that $L(N) = L(\mathcal{P}N)$:

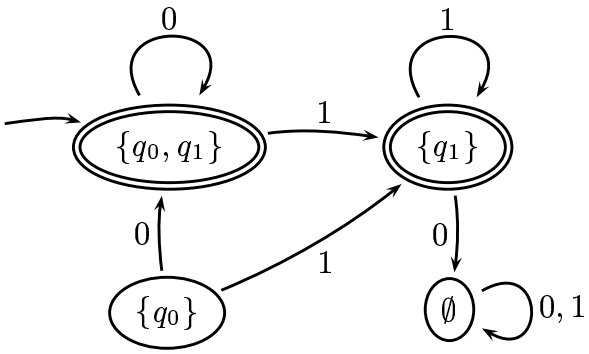
- $Q_{\mathcal{P}N} \stackrel{\text{def}}{=} \{ S : S \subseteq Q_N \}$
- $\Sigma_{\mathcal{P}N} \stackrel{\text{def}}{=} \Sigma_N$
- $S \stackrel{a}{\longrightarrow} S'$ in $\mathcal{P}N$ iff $S' = \{ q' : \exists q \in S.(q \stackrel{a}{\Longrightarrow} q' \text{ in } N) \}$
- $q_{\mathcal{P}N} \stackrel{\text{def}}{=} \{ q : q_N \stackrel{\epsilon}{\Longrightarrow} q \}$
- $F_{\mathcal{P}N} \stackrel{\text{def}}{=} \{ S \in Q_{\mathcal{P}N} : F_N \cap S \neq \emptyset \}$

Example. All words that begin with a string of 0's followed by a string of 1's.

NFA N



DFA $\mathcal{P}N$



Note. State $\{q_0\}$ is redundant.

Proof of "$L(N) \subseteq L(\mathcal{P}N)$":

Suppose $\epsilon \in L(N)$. Then $q_N \stackrel{\epsilon}{\Longrightarrow} q'$ for some $q' \in F_N$. Hence $q' \in q_{\mathcal{P}N}$, and so, $q_{\mathcal{P}N} = \{ q'' : q_N \stackrel{\epsilon}{\Longrightarrow} q'' \} \in F_{\mathcal{P}N}$ i.e. $\epsilon \in L(\mathcal{P}N)$.

Now take any non-null $u = a_1 \cdots a_n$. Suppose $u \in L(N)$. Then there is a sequence of N-transitions

$$q_N \stackrel{a_1}{\Longrightarrow} q_1 \stackrel{a_2}{\Longrightarrow} \cdots\cdots \stackrel{a_n}{\Longrightarrow} q_n \in F_N \qquad (1)$$

Since $\mathcal{P}N$ is deterministic, feeding $a_1, \cdots, a_n$ to it results in the sequence of $\mathcal{P}N$-transitions

$$q_{\mathcal{P}N} \stackrel{a_1}{\longrightarrow} S_1 \stackrel{a_2}{\longrightarrow} \cdots\cdots \stackrel{a_n}{\longrightarrow} S_n \qquad (2)$$

where

$$\begin{aligned} S_1 &= \{ q' : \exists q \in q_{\mathcal{P}N}.(q \stackrel{a_1}{\Longrightarrow} q' \text{ in } N) \} \\ S_2 &= \{ q' : \exists q \in S_1.(q \stackrel{a_2}{\Longrightarrow} q' \text{ in } N) \} \\ \vdots &\quad \vdots \end{aligned}$$

By definition of $\delta_{\mathcal{P}N}$, from (1), we have $q_1 \in S_1$, and so $q_2 \in S_2$, $\cdots$, and so

$q_n \in S_n$, and hence $S_n \in F_{\mathcal{P}N}$ because $q_n \in F_N$. Thus (2) shows that $u \in L(\mathcal{P}N)$.

Proof of "$L(\mathcal{P}N) \subseteq L(N)$":

Suppose $\epsilon \in L(\mathcal{P}N)$. Then $q_{\mathcal{P}N} \in F_{\mathcal{P}N}$ i.e. $F_N \cap \{ q : q_N \stackrel{\epsilon}{\Longrightarrow} q \} \neq \emptyset$, or equivalently, for some $q' \in F_N$, $q_N \stackrel{\epsilon}{\Longrightarrow} q'$. Hence $\epsilon \in L(N)$.

Now suppose some non-null $u = a_1 \cdots a_n \in L(\mathcal{P}N)$. I.e. there is a sequence of $\mathcal{P}N$-transitions of the form (2) with $S_n \in F_{\mathcal{P}N}$ i.e. with S_n containing some $q_n \in F_N$. Now since $q_n \in S_n$, by definition of $\delta_{\mathcal{P}N}$, there is some $q_{n-1} \in S_{n-1}$ with $q_{n-1} \stackrel{a_{n-1}}{\Longrightarrow} q_n$ in N. Working backwards in this way, we can build up a sequence of N-transitions like (1), until we deduce that $q_N \stackrel{a_1}{\Longrightarrow} q_1$. Thus we get a sequence of N-transitions with $q_n \in F_N$, and hence $u \in L(N)$.

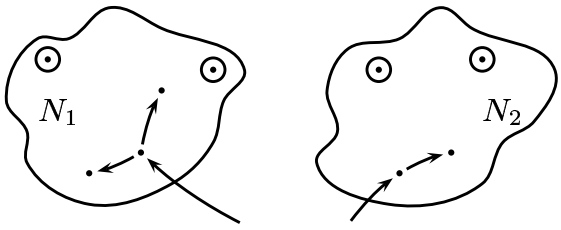
□

Closure under regular operations revisited

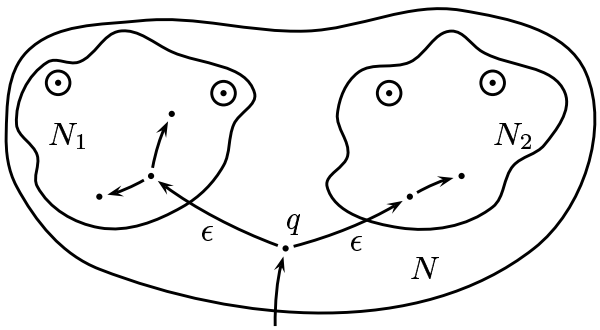
Using nondeterminism makes *some* proofs much easier.

Theorem. Regular languages are closed under union.

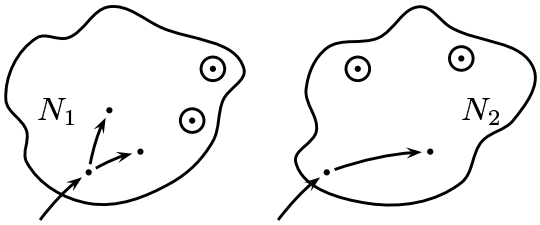
Take NFAs N_1 and N_2.



Define N that accepts $L(N_1) \cup L(N_2)$ by adding a new start state q to the disjoint union of (the respective state transition graphs of) N_1 and N_2, and a ϵ-transition from q to each start state of N_1 and N_2.

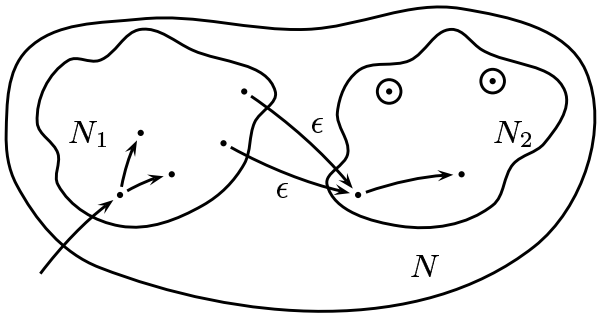


Theorem. Regular languages are closed under concatenation.



Take NFAs N_1 and N_2.

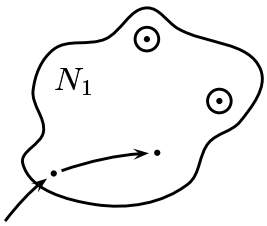
An NFA N that accepts $L(N_1) \cdot L(N_2)$ can be obtained from the disjoint union of N_1 and N_2 by making the start state of N_1 the start state of N, and by adding an ϵ-transition from each accepting state of N_1 to the start state of N_2. The accepting states of N are those of N_2.



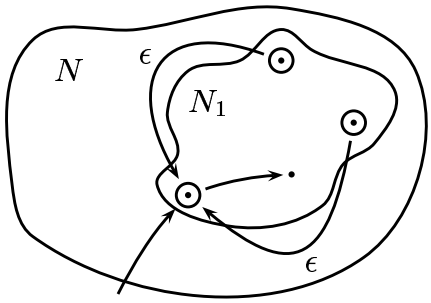
Theorem. Regular languages are closed under star.

First attempt:

Take an NFA $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ that accepts A_1. Construct N that accepts $A_1^* = \{\epsilon\} \cup A_1 \cup A_1\, A_1 \cup \cdots$.



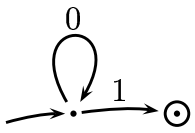
Obtain N from N_1 by making the start state accepting, and by adding a new ϵ-transition from each accepting state to the start state.



What is wrong with this?

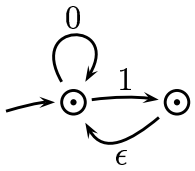
Consider the two-node two-edge NFA N_1
that accepts $\{ 0^i 1 : i \geq 0 \}$
(namely the language
defined by the regular expression 0^*1).

N_1



The above construction gives the NFA N

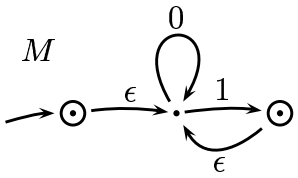
N



But N accepts $(0 + 0^*1)^* = (0 + 1)^* \neq (0^*1)^*$.
E.g. N accepts 010 which is not in $L(N_1)^*$

The NFA M accepts $L(N_1)^*$

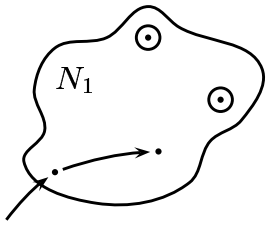
M



Proof: Regular languages are closed under star

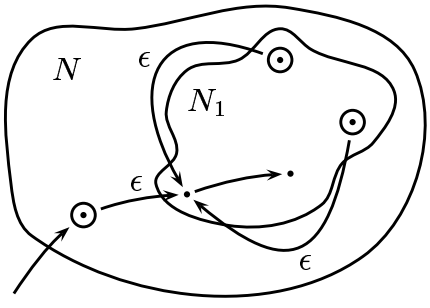
Second (correct) attempt:

Take an NFA $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ that accepts A_1.



Define $N = (Q_1 \cup \{ q_0 \}, \Sigma, \delta, q_0, F_1 \cup \{ q_0 \})$
where

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \epsilon \\ \delta_1(q,a) \cup \{ q_1 \} & q \in F_1 \text{ and } a = \epsilon \\ \{ q_1 \} & q = q_0 \text{ and } a = \epsilon \\ \emptyset & q = q_0 \text{ and } a \neq \epsilon \end{cases}$$



Regular Expressions

A notation to describe "finite-automaton" patterns

E.g. Binary strings that "begin with a string of 0's followed by a string of 1's".

Binary strings that "start and end with the same symbol".

Regular expressions are just such a compact notation to describe these patterns, which are described respectively as $0^* \cdot 1^*$ and
$0 + 1 + (0(0+1)^*0) + (1(0+1)^*1)$.

Regular expressions have many important applications in CS:

- Lexical analysis in compiler construction.
- Search facilties provided by text editors and databases; utilities such as `awk` and `grep` in Unix.
- Programming languages such as Perl and XML.

Regular expressions and their denotations

Fix a Σ. We define simultaneously *regular expression* E and the *language denoted* by E, written $L(E)$, by induction over the following rules:

- The constants ϵ and $\emptyset$ are regular expressions; $L(\epsilon) \stackrel{\text{def}}{=} \{\epsilon\}$ and $L(\emptyset) \stackrel{\text{def}}{=} \emptyset$.
- For $a \in \Sigma$, a is a regular expression; $L(a) \stackrel{\text{def}}{=} \{a\}$.
- If E and F are regular expressions, then so are $(E + F)$, $(E \cdot F)$ and (E^*); we have

union	$L((E + F))$	$\stackrel{\text{def}}{=}$	$L(E) \cup L(F)$
concatenation	$L((E \cdot F))$	$\stackrel{\text{def}}{=}$	$L(E) \cdot L(F)$
star	$L((E^*))$	$\stackrel{\text{def}}{=}$	$(L(E))^*$

Notations

$+$ is sometimes written $\cup$ or $|$, and $(E \cdot F)$ is sometimes simply written (EF).

Parentheses may sometimes be omitted. We assume:

(i) The regular expression operators have the following *order of precedence* (in decreasing order): star, concatenation, union.

E.g. 01^* means $0(1^*)$, not $(01)^*$; $0 + 10$ means $0 + (10)$, not $(0 + 1)0$.

(ii) Union and concatenation associate to the left i.e. $E \cdot F \cdot G$ means $(E \cdot F) \cdot G$. (Since union and concatenation are associative, the choice of left or right association does not really matter.)

Examples

(i) $01^* + 1$ is formally $((0 \cdot (1^*)) + 1)$.

(ii) $(0 + 1)01^*0$ is formally $((((0 + 1) \cdot 0) \cdot (1^*)) \cdot 0)$.

Examples: Languages over $\{0, 1\}$ denoted by regular expressions

1. 0^*10^* denotes words that have exactly one 1.
2. $(0+1)^*1(0+1)^*$ denote words that have at least one 1.
3. $0(0+1)^*0 + 1(0+1)^*1 + 0 + 1$ denotes words that start and end with the same symbol.

We shall say that a word w *matches* E just in case $w \in L(E)$.

Equivalence of regular expressions

We say that E and F are *equivalent*, written $E \equiv F$, just in case $L(E) = L(F)$.

Note that $\equiv$ is an equivalence relation.

Some identities

1. Associativity: $(E + F) + G \equiv E + (F + G)$ and $(EF)G \equiv E(FG)$.
2. Commutativity: $E + F \equiv F + E$
3. $E\,\emptyset \equiv \emptyset$.
4. $\emptyset^* \equiv \{\,\epsilon\,\}$.
5. $E + \emptyset \equiv E \equiv \emptyset + E$ and $E \cdot \epsilon \equiv E \equiv \epsilon \cdot E$.
6. But *in general* $E + \epsilon \not\equiv E$, and $E \cdot \emptyset \not\equiv E$.

   For which E do the equivalences hold?

Example: Verify $(a+b)^* \equiv a^*\,(b\,a^*)^*$

Proof. Observe that $L((a+b)^*)$ is the set of all strings over $\{a, b\}$, thus $L(a^*\,(b\,a^*)^*) \subseteq L((a+b)^*)$.

Note that any $s \in L((a+b)^*)$ can be written *uniquely* as

$$a^{n_0}\,b\,a^{n_1}\,b\,\cdots\,a^{n_{r-1}}\,b\,a^{n_r} \tag{1}$$

where a^n means $\underbrace{a\cdots a}_{n}$, each $n_i \geq 0$ and $r \geq 0$. (r is just the number of occurrences of b in s. E.g. in case s is b, $n_0 = n_1 = 0$; in case s is ϵ, $n_0 = 0$.)

Any string in $L((ba^*)^*)$ has the shape $\underbrace{(b\,a^{n_1})\,\cdots\,(b\,a^{n_r})}_{r}$, where each $n_i \geq 0$ and $r \geq 0$. It follows that any string in $L(a^*(ba^*)^*)$ has the shape $a^{n_0}\,b\,a^{n_1}\,\cdots\,b\,a^{n_r}$ where each $n_i \geq 0$ and $r \geq 0$ i.e. of the shape (**??**). □

Question: Is there a finite set of (equivalence) axioms and rules such that $(a+b)^* \equiv a^*\,(b\,a^*)^*$ (indeed any valid equivalence) is a theorem?

Kozen's Axioms for the Algebra of Regular Expressions

1.	$E+(F+G) \equiv (E+F)+G$	7.	$E(F+G) \equiv EF+EG$
2.	$E+F \equiv F+E$	8.	$(E+F)G \equiv EG+FG$
3.	$E+\emptyset \equiv E$	9.	$\emptyset E \equiv E\emptyset \equiv \emptyset$
4.	$E+E \equiv E$	10.	$\epsilon + EE^* \equiv E^*$
5.	$(EF)G \equiv E(FG)$	11.	$\epsilon + E^*E \equiv E^*$
6.	$\epsilon E \equiv E\epsilon \equiv E$		

and two rules:

12. $F + EG \leq G \Rightarrow E^*F \leq G$
13. $F + GE \leq G \Rightarrow FE^* \leq G$

Note: $E \leq F$ means $L(E) \subseteq L(F)$.

(Sound and) Complete Axiomatization of Equivalence

Soundness: Each axiom is a valid equivalence between regular expressions, and each rule is sound (i.e. if the premise is a valid equivalence, so is the conclusion).

E.g. to say that rule (13) is sound is to say that for any E, F and G, if $L(F + GE) \subseteq L(G)$, then $L(FE^*) \subseteq L(G)$.

Completeness: Further the axiomatization is *complete* i.e.

> **Kozen's Theorem**. All valid equivalences between regular expressions can be derived from Kozen's axioms and rules, using the laws of (in)equational logic i.e.
>
> if $E \leq F$ then $E \oplus G \leq F \oplus G, G \oplus E \leq G \oplus F$ and $E^* \leq F^*$ where $\oplus = +$ and $\cdot$.

On proving Kozen's Theorem

Soundness proof: For an illustration, we prove (13).

Suppose $L(F + GE) \subseteq L(G)$. Any $w \in L(FE^*)$ has the form $fe_1 \cdots e_n$ where $n \geq 0$, $f \in L(F)$ and $e_i \in L(E)$. We prove that $w \in L(G)$ by induction on n.

The base case of $n = 0$ follows from $L(F) \subseteq L(G)$.

For the inductive case, we need to show that $fe_1 \cdots e_{n+1} \in L(G)$. Now $fe_1 \cdots e_n \in L(FE^*)$; by the IH we have $fe_1 \cdots e_n \in L(G)$; and so $fe_1 \cdots e_n e_{n+1} \in L(GE)$ which is contained in $L(G)$ by supposition.

Completeness proof: beyond the scope of this course. □

Example: $(a+b)^* \equiv a^*\,(b\,a^*)^*$ revisited

We prove the harder direction "$\leq$" using Kozen's system.

Note that $\epsilon, a, b \leq a^*(ba^*)^*$. [Ex. Prove it using $\epsilon \leq E^*$!]

We have $a\,(a^*(ba^*)^*) \equiv (a\,a^*)(ba^*)^* \leq a^*(ba^*)^*$ by (5) and (10). Similarly

$$b\,(a^*(ba^*)^*) \equiv (ba^*)(ba^*)^* \leq (ba^*)^* \equiv \epsilon(ba^*)^* \leq a^*(ba^*)^*.$$

The last two steps follow from (6) and (10). Because of the preceding, by (7), we have $(a+b) + (a+b)(a^*(ba^*)^*) \leq a^*(ba^*)^*$, and so, by rule (12) – taking E and F to be $(a+b)$ – we have

$$(a+b)^*(a+b) \leq a^*(ba^*)^*.$$

Since $\epsilon \leq a^*(ba^*)^*$, by rule (10) we have $(a+b)^* \leq a^*\,(b\,a^*)^*$ as desired. □

Equivalence of regular expressions and finite automata

The equivalence of regular expressions and finite automata is a fundamental result in Automata Theory.

Kleene's Theorem. Let $L \subseteq \Sigma^*$. The following are equivalent:

(i) L is regular i.e. for some finite automaton M, $L = L(M)$.

(ii) L is denoted by some regular expression E i.e. $L = L(E)$.

Proof of Kleene's Theorem: "(i) $\Rightarrow$ (ii)"

We show that there is a systematic way to transform a regular expression E to an equivalent NFA N_E – so that $L(E) = L(N_E)$ – by recursion on the structure of E.

Base cases: For each of the three cases, namely $E = \epsilon, \emptyset$ and a where $a \in \Sigma$, there is an NFA N_E that accepts $L(E)$.

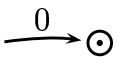
Inductive cases: Take regular expressions E and F. Suppose N_E and N_F are NFAs that accept $L(E)$ and $L(F)$ respectively. We have proved that regular languages are closed under union, concatenation and star by constructing NFAs that accept $L(N_E) \cup L(N_F)$, $L(N_E) \cdot L(N_F)$ and $(L(N_E))^*$ respectively. By definition, these NFAs are equivalent to $E + F$, $E \cdot F$ and E^* respectively.

□

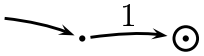
Example: Transforming regular expressions to NFAs

We construct the NFA that accepts $(0 + 01^*0)^*0$.

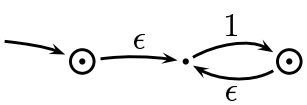
0



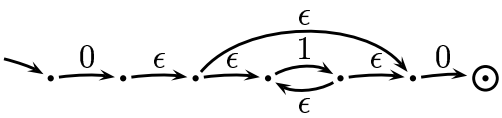
1



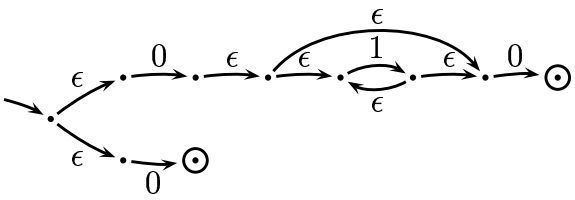
1^*



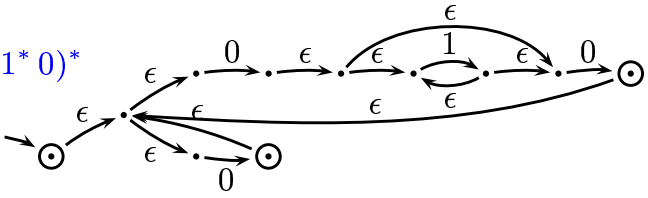
$0\,1^*\,0$



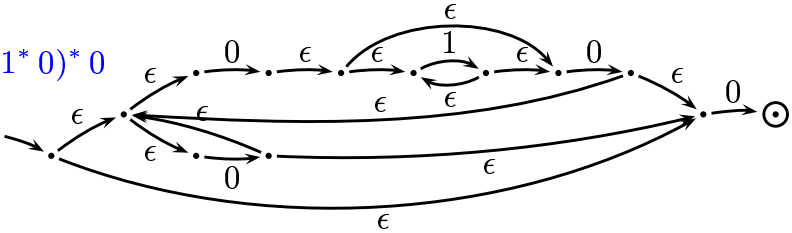
$(0 + 0\,1^*\,0)$



$(0 + 0\,1^*\,0)^*$



$(0 + 0\,1^*\,0)^*\,0$



Proof of Kleene's Theorem: "(ii) $\Rightarrow$ (i)"

Given an NFA $M = (Q, \Sigma, \delta, q_0, F)$, for $X \subseteq Q$ and $q, q' \in Q$, we construct, by induction on the size of X, a regular expression

$$E^X_{q,q'}$$

whose denotation is the set of all strings w such that

> there is a path from q to q' in M labelled by w (i.e. $q \stackrel{w}{\Longrightarrow} q'$) such that all intermediate states along that path lie in X.

It suffices to prove:

Lemma. For any $X \subseteq Q$, for any $q, q' \in Q$, there is a regular expression $E^X_{q,q'}$ satisfying $L(E^X_{q,q'}) =$

$$\{\, w \in \Sigma^* : q \stackrel{w}{\Longrightarrow} q' \text{ in } M \text{ with all intermediate states of seq. in } X \,\}$$

We prove the Lemma by induction on the size of X.

Basis: $X = \emptyset$. Let $a_1, \cdots, a_k$ be all the symbols in $\Sigma \cup \{\, \epsilon \,\}$ such that $q' \in \delta(q, a_i)$. For $q \neq q'$, take

$$E^{\emptyset}_{q,q'} \stackrel{\text{def}}{=} \begin{cases} a_1 + \cdots + a_k & \text{if } k \geq 1 \\ \emptyset & \text{if } k = 0 \end{cases}$$

and for $q = q'$, take

$$E^{\emptyset}_{q,q'} \stackrel{\text{def}}{=} \begin{cases} a_1 + \cdots + a_k + \epsilon & \text{if } k \geq 1 \\ \epsilon & \text{if } k = 0 \end{cases}$$

Inductive step: For a nonempty X, choose an element $r \in X$ - call it the *separating state*. Now any path from q to q' with all intermediate states in X, *either*

(1) never visits r, or

(2) visits r for the first time, followed by a finite number of loops from r back to itself without visiting r in between but staying in X, and finally followed by a path from r to q'.

Thus we take

$$E^X_{q,q'} \stackrel{\text{def}}{=} \underbrace{E^{X-\{r\}}_{q,q'}}_{(1)} + \underbrace{E^{X-\{r\}}_{q,r} \cdot (E^{X-\{r\}}_{r,r})^* \cdot E^{X-\{r\}}_{r,q'}}_{(2)}$$

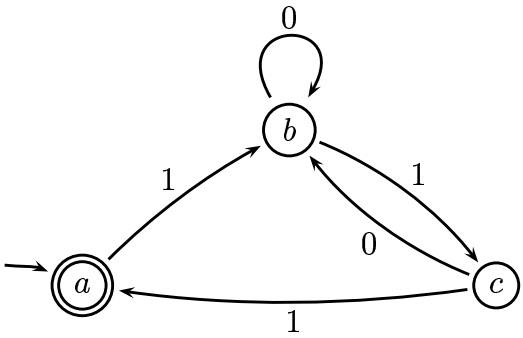
Finally the expression $\sum_{f \in F} E^Q_{q_0,f}$ has denotation $L(M)$. □

Hueristic: it is best to choose a separating state r that disconnects the automaton as much as possible.

Example: Transforming NFAs to regular expressions

Consider the NFA $M = (\{a,b,c\}, \{0,1\}, \delta, a, \{a\})$ where δ is given by

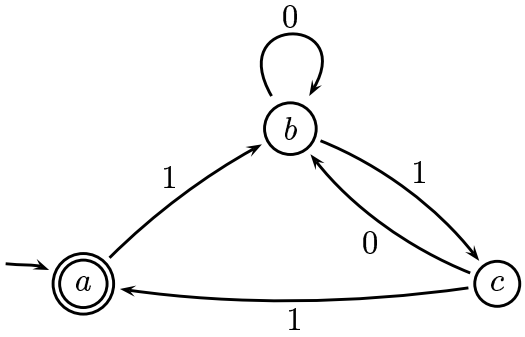
	0	1	ϵ
a	$\emptyset$	$\{b\}$	$\emptyset$
b	$\{b\}$	$\{c\}$	$\emptyset$
c	$\{b\}$	$\{a\}$	$\emptyset$



We pick b as the separating state: the required reg. exp. is

$$E^{\{a,b,c\}}_{a,a} = E^{\{a,c\}}_{a,a} + E^{\{a,c\}}_{a,b} \cdot (E^{\{a,c\}}_{b,b})^* \cdot E^{\{a,c\}}_{b,a}.$$

By inspection $E^{\{a,c\}}_{a,a} = \epsilon$, $E^{\{a,c\}}_{a,b} = 1$ and $E^{\{a,c\}}_{b,a} = 11$.



Picking c as the separaring state, we have

$$E_{b,b}^{\{a,c\}} = E_{b,b}^{\{a\}} + E_{b,c}^{\{a\}} \cdot (E_{c,c}^{\{a\}})^* \cdot E_{c,b}^{\{a\}}$$

where $E_{b,b}^{\{a\}} = 0 + \epsilon$, $E_{b,c}^{\{a\}} = 1$, $E_{c,c}^{\{a\}} = \epsilon$ and $E_{c,b}^{\{a\}} = 0 + 11$.

Hence putting it all together we have

$$E_{a,a}^{\{a,b,c\}} = \epsilon + 1(0 + \epsilon + 1\epsilon^*(0 + 11))^*11 \equiv \epsilon + 1\,(0 + 10 + 111))^*\,11$$

I.e. $L(M) = L(\epsilon + 1\,(0 + 10 + 111)^*\,11)$.

Question

Is there a procedure (algorithm) that, given a string s and a regular expression E, will decide whether or not s matches E?

The Pumping Lemma

The Pumping Lemma is a powerful technique for proving that certain languages are *not* regular.

Claim: $B = \{ 0^n 1^n : n \geq 0 \}$ is not regular

Informal argument.

If there were a DFA that recognizes B, it would need to remember the number of 0's read from the input string. This would require a way to store an arbitrarily large number, but any DFA has only a finite amount of memory (given by the fixed number of states). □

But need to be careful: both

$$L_1 = \{ w : w \text{ has an equal number of 0s and 1s} \}$$
$$L_2 = \{ w : w \text{ has an equal no. of occurrences of 01 and 10 as substrings} \}$$

seem to require infinite memory to recognize. Now L_1 is not regular but

Exercise(*Moderately hard*). Prove that L_2 is regular.

The Pumping Lemma, in poetic form

"Any regular language L has a magic number p
And any long-enough word in L has the following property:
Amongst its first p symbols is a segment you can find
Whose repetition or omission leaves x amongst its kind."

"So if you find a language L which fails this acid test,
And some long word you pump becomes distinct from all the rest,
By contradiction you have shown that language L is not
A regular guy, resiliant to the damage you have wrought."

"But if, upon the other hand, x stays within its L,
Then either L is regular, or else you chose not well.
For w is xyz, and y cannot be null,
And y must come before p symbols have been read in full."

"As mathematical postscript, an addendum to the wise:
The basic proof we outlined here does certainly generalize.
So there is a pumping lemma for all languages context-free,
Although we do not have the same for those that are r.e."

By Martin Cohn and Harry Mairson

The Pumping Lemma

Pumping Lemma. If A is a regular language, then there is a number p – the *pumping length* – such that if $s \in A$ of length at least p, then s may be divided into three pieces, $s = x\,y\,z$, satisfying:

(i) for each $i \geq 0$, $x\,y^i\,z \in A$
'Words "pumped up" from s belong to A.'

(ii) $|y| > 0$

(iii) $|x\,y| \leq p$.

Note: without (ii), the Lemma is vacuous (because $\epsilon^i = \epsilon$ for all $i \geq 0$).

The Pumping Lemma is a complex statement: it is equivalent to

$$\forall L \in \textbf{Reg} \,.\, \exists p \geq 1 \,.\, \forall s \in L \,.\, \exists x, y, z \in \Sigma^* \,.\, \forall i \geq 0 \,.\, \Xi$$

where Ξ is $|s| \geq p \;\rightarrow\; s = x\,y\,z \wedge |x\,y| \leq p \wedge |y| > 0 \wedge x\,y^i\,z \in L$.

Proof of the Pumping Lemma

Let $M = (Q, \Sigma, \delta, q_{\text{init}}, F)$ be a DFA that accepts A, and let $p = |Q|$.
Suppose $s = a_1 \cdots a_n \in L(M)$ where $n \geq p$. We have

$$\underbrace{q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_p} q_p}_{p+1 \text{ states}} \cdots\cdots \xrightarrow{a_n} q_n \in F$$

where $q_0 = q_{\text{init}}$. By the Pigeonhole Principle, $q_0, \cdots, q_p$ cannot all be distinct.
So $q_j = q_{j'}$ for some $0 \leq j < j' \leq p$. Thus the above transition sequence is

$$q_0 \xrightarrow{x}{}^* q_j \xrightarrow{y}{}^* q_{j'} (= q_j) \xrightarrow{z}{}^* q_n \in F$$

where $x = a_1 \cdots a_j$, $y = a_{j+1} \cdots a_{j'}$ and $z = a_{j'} \cdots a_n$. We have
$|x\,y| \leq p$ and $|y| > 0$, and for every $i \geq 0$, $x\,y^i\,z \in L(M)$ as

$$q_0 \xrightarrow{x}{}^* \underbrace{q_j \xrightarrow{y}{}^* q_j \cdots \xrightarrow{y}{}^* q_j}_{i} \xrightarrow{z}{}^* q_n \in F$$

□

Example: $B = \{ 0^i 1^i : i \geq 0 \}$ is not regular.

Proof. Suppose, for a contradiction, B is regular. Let the pumping length be p.

Take $s = 0^p 1^p \in B$. Since $|s| > p$, by the Lemma, there are x, y, z such that $s = x\,y\,z$ where $|x\,y| \leq p$ and $|y| > 0$. Hence $x = 0^a$, $y = 0^b$ where $b > 0$, and $a + b \leq p$.

The Lemma further asserts: for each $i \geq 0$, $0^a\ 0^{bi}\ 0^{p-a-b}\ 1^p \in B$. In particular (taking $i = 0$) $0^a 0^{p-a-b}\ 1^p = 0^{p-b} 1^p \in B$, a contradiction. □

Exercise. Convince yourself that the same argument above can be used to show that $\{ w : w$ has equal no. of 0s and 1s $\}$ is not regular.

The Pumping Lemma is not always easy to apply: the trick is to identify an appropriate word to "pump". It is often useful to "pump down" i.e. take $i = 0$.

A powerful characterization of regular languages

Let $x, y \in \Sigma^*$ be strings and let $L \subseteq \Sigma$.

We say that x and y are L-*indistinguishable*, written $x \equiv_L y$, if for every $z \in \Sigma^*$, $xz \in L$ iff $yz \in L$.

Fact. $\equiv_L$ is an equivalence relation.

We define the *index* of L to be the number of equivalence classes of L.

The index of L may be finite or infinite.

Examples

Take $\Sigma = \{ 0, 1 \}$.

(i) $L_3 = \{ w : w$ has even length $\}$.

$u \equiv_{L_3} v$ iff $|u| \equiv |v| (\text{mod } 2)$.

Now $\equiv_{L_3}$ has two equivalence classes:

$[\epsilon] = [00] = [10] = \cdots = \{ w : |w|$ even $\}$ and
$[1] = [010] = [110] = \cdots = \{ w : |w|$ odd $\}$.

(ii) $L_4 = \{ w : w$ has equal numbers of 0s and 1s $\}$.

For any $i, j \geq 0$, if $i \neq j$ then $0^i \not\equiv_{L_4} 0^j$ (because $0^i 1^i \in L_4$ but $0^j 1^i \notin L_4$).

Therefore the index of L_4 is infinite.

A powerful characterization of regular languages (con't)

Myhill-Nerode Theorem: A language L is regular iff $\equiv_L$ has finite index. Moreover the index is the size (= number of states) of the *smallest* DFA accepting L.

Note: The Pumping Lemma is *not* a characterization of regular languages: it is *not* an if-and-only-if statement.

Proof of the Myhill-Nerode Theorem

It suffices to prove:

(i) If L is accepted by a DFA with k states, then L has index at most k.

(ii) If L has a finite index k (say), then it is accepted by a DFA with k states.

(i): Suppose L is accepted by a DFA $M = (Q, \Sigma, \delta, q_0 F)$. We check that

$$\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y) \implies x \equiv_L y.$$

(Recall: for $x \in \Sigma^*$, we have $\hat{\delta}(q, x) = q'$ iff $q \xrightarrow{x}{}^* q'$.)

Take $x_1, \cdots, x_{k+1} \in \Sigma^*$, all distinct. Since M has only k states, by the Pigeonhole Principle, for some $1 \leq i < j \leq k+1$, we have $\hat{\delta}(q_0, x_i) = \hat{\delta}(q_0, x_j)$, and so, $x_i \equiv_L x_j$. It follows that $\equiv_L$ has at most k equivalence classes.

(ii) Assume that L has a finite index. Define a structure $M = (\Sigma, Q, q_0, \delta, F)$ as follows:

$$\begin{aligned} Q &= \{\, [x] : x \in \Sigma^* \,\} \\ q_0 &= [\epsilon] \\ \delta([x], a) &= [xa] \\ F &= \{\, [w] : w \in L \,\} \end{aligned}$$

We need to verify: (a) M is a DFA, (b) M accepts L.

For (a), $|Q|$ is the index of L which is finite, and δ is a well-defined function because $x \equiv_L y$ implies $xa \equiv_L ya$ for any $a \in \Sigma$.

For (b), for any $w \in \Sigma^*$, $w \in L(M)$ iff $[\epsilon] \xrightarrow{w}{}^* [w] \in F$ iff $w \in L$.

□

Example: $L = \{ ww : w \in \{ 0, 1 \}^* \}$ is not regular

Take any distinct $i, j \geq 0$. We have $0^i 1 \not\equiv_L 0^j 1$ because $0^i 10^i 1 \in L$ but $0^j 10^i 1 \notin L$. Hence $\equiv_L$ has an infinite index. Thus L is not regular by Myhill-Nerode.

Closure Properties of Regular Languages

Regular languages are closed under the following operations:

1. The regular operations: union, concatenation, star
2. Intersection
3. Complementation
4. Word reversal
5. Homomorphism: Given a function $\phi : \Sigma_1 \to \Sigma_2^*$. Define $\phi^*(a_1 \cdots a_n) = \phi(a_1) \cdots \phi(a_n)$, and for any language L_1 over Σ_1, define
$$\phi(L_1) \stackrel{\text{def}}{=} \{ \phi^*(w) : w \in L_1 \}.$$
If L_1 is regular, so is $\phi(L_1)$.

6. Inverse homomorphism: For any L_2 over Σ_2. Define

$$\phi^{-1}(L_2) \stackrel{\text{def}}{=} \{ w \in \Sigma_1^* : \phi(w) \in L_2 \}.$$

If L_2 is regular, so is $\phi^{-1}(L_2)$.

Context Free Grammars

Regular languages can be specified in terms of finite automata that accept or reject strings, equivalently, in terms of regular expressions, which strings are to match.

This section introduces a new, *generative* means of specifying sets of strings.

Context-free grammars (CFG): A way of generating words

Ingredients of a CFG:

$$(\{\text{ variables }\}, \{\text{ terminals }\}, \{\text{ productions (or rules) }\}, \text{start symbol})$$

The start symbol is a special variable.

A CFG generates strings over the alphabet $\Sigma = \{\text{ terminals }\}$.

Example. $G = (\{ A, B \}, \{ 0, 1 \}, \mathcal{R}, A)$ where $\mathcal{R}$ consists of three rules:

$$\begin{cases} A & \to & 0\,A\,1 \\ A & \to & B \\ B & \to & \epsilon \end{cases}$$

How to generate strings using a CFG

1. Set w to be the start symbol.
2. Choose an occurrence of a variable X in w if any, otherwise STOP.
3. Pick a production whose lhs is X, replace the chosen occurrence of X in w by the rhs.
4. GOTO 2.

Example $G = (\{A, B\}, \{0, 1\}, \{A \to 0\,A\,1 \mid B, \quad B \to \epsilon\}, A)$ generates $\{0^i\,1^i : i \geq 0\}$.

$$\begin{aligned} A &\Rightarrow 0\,A\,1 \\ &\Rightarrow 0\,0\,A\,1\,1 \\ &\Rightarrow 0\,0\,B\,1\,1 \\ &\Rightarrow 0\,0\,\epsilon\,1\,1 = 0^2\,1^2 \end{aligned}$$

Such sequences are called *derivations*.

Definition: Context-free Grammar

A *context-free grammar* is a 4-tuple $G = (V, \Sigma, \mathcal{R}, S)$ where

(i) V is a finite set of *variables* (or *non-terminals*)

(ii) Σ (the alphabet) is a finite set of *terminals*

(iii) $\mathcal{R}$ is a finite set of *productions*. A *production* (or *rule*) is an element of $V \times (V \cup \Sigma)^*$, written $A \to w$.

(iv) $S \in V$ is the *start symbol*.

We define a binary relation $\Rightarrow$ over $(\{\,V \cup \Sigma\,\})^*$ by: for each $u, v \in (\{\,V \cup \Sigma\,\})^*$, for each $A \to w$ in $\mathcal{R}$

$$u\,A\,v \Rightarrow u\,w\,v$$

We write $\Rightarrow^*$ for the reflexive and transitive closure of $\Rightarrow$.

The *language of the grammar*, written $L(G)$, is $\{\,w \in \Sigma^* : S \Rightarrow^* w\,\}$.

Examples

Palindromes over $\{a, b, c\}$: generated by $(\{S, T\}, \{a, b, c\}, \mathcal{R}, S)$ where $\mathcal{R}$ consists of eight rules:

$$\begin{array}{lcl} S & \to & a\,T\,a \mid b\,T\,b \mid c\,T\,c \mid T \\ T & \to & S \mid a \mid b \mid c \mid \epsilon \end{array}$$

Note: Use $\mid$ to save writing. The above can be simplified: one variable suffices.

Well-balanced parentheses: generated by $(\{S\}, \{\texttt{(}, \texttt{)}\}, \mathcal{R}, S)$ where $\mathcal{R}$ consists of

$$S \to \texttt{(}\,S\,\texttt{)} \mid S\,S \mid \epsilon$$

E.g. **(( ) (( ))) ( )**

Exercise. Prove that the grammar generates precisely all well-balanced parentheses. [*Hint*. Define the "balance index" of a string, and argue by induction on length.]

Regular languages are context-free

A language is *context-free* just in case it is generated by some CFG.

A CFG is *right-linear* if every rule is either of the form $R \to wT$ or of the form $R \to w$ where w ranges over strings of terminals, and R and T over variables.

Theorem. A language is regular iff it is generated by a right-linear CFG.

Proof idea. Say a CFG is *strongly right-linear* if each rule has one of the following forms: $R \to aT$, $R \to T$ or $R \to \epsilon$ where a ranges over terminals, and R and T over variables.

Fact. Each right-linear CFG is equivalent to a strongly right-linear one.

For each rule $R \to wT$ with $w = a_0 a_1 \cdots a_n$ we add n fresh variables, say $V_1, \ldots V_n$, and replace the rule $R \to wT$ by the rules

$$R \to a_0 V_1, \quad V_1 \to a_1 V_2, \quad \ldots \quad V_n \to a_n T$$

"$\Rightarrow$": We map DFAs $M = (Q, \Sigma, \delta, q_0, F)$ to strongly right-linear CFGs $G_M = (Q, \Sigma, \mathcal{R}, q_0)$ where

$$\begin{aligned} q \to a\, q' \in \mathcal{R} \quad &\Longleftrightarrow \quad \delta(q, a) = q' \\ q \to \epsilon \in \mathcal{R} \quad &\Longleftrightarrow \quad q \in F \end{aligned}$$

"$\Leftarrow$": We map strongly right-linear CFGs $G = (V, \Sigma, \mathcal{R}, S)$ to NFAs $N_G = (V, \Sigma, \delta, S, F)$ where

$$\begin{aligned} R \xrightarrow{a} R' \text{ (i.e. } R' \in \delta(R, a)) \quad &\Longleftrightarrow \quad R \to a\, R' \in \mathcal{R} \\ R \xrightarrow{\epsilon} R' \text{ (i.e. } R' \in \delta(R, \epsilon)) \quad &\Longleftrightarrow \quad R \to R' \in \mathcal{R} \\ R \in F \quad &\Longleftrightarrow \quad R \to \epsilon \in \mathcal{R} \end{aligned}$$

□

Easy exercise. Give a right-linear CFG that generates $0^*\, 1^*\, 0^*$.

Parse trees

Parse trees: Each derivation determines a *parse tree*.

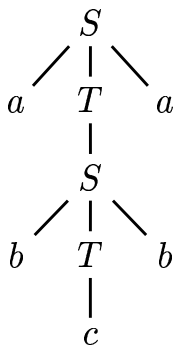
Parse trees are *ordered* trees: the children at each node are ordered.

The parse tree of a derivation abstracts away from the order in which variables are replaced in the sequence.

derivation

$$\begin{aligned} S &\Rightarrow a\, T\, a \\ &\Rightarrow a\, S\, a \\ &\Rightarrow a\, b\, T\, b\, a \\ &\Rightarrow a\, b\, c\, b\, a \end{aligned}$$

parse tree



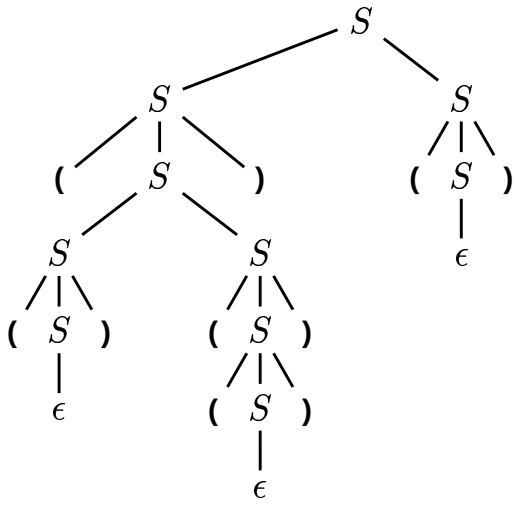
Example Parentheses

$S \rightarrow (\,S\,) \mid S\,S \mid \epsilon$

derivation

$$
\begin{aligned}
S &\Rightarrow \underline{S}\,S \\
&\Rightarrow \overline{(\underline{S})}\,S \\
&\Rightarrow (\overline{\underline{S}\,\underline{S}})\,S \\
&\Rightarrow^2 \overline{(\underline{S})(\underline{S})}\,\underline{S} \\
&\Rightarrow^3 (\underline{\epsilon})(\overline{(\underline{S})})\,\overline{(\underline{S})} \\
&\Rightarrow^2 (\epsilon)((\epsilon))\,(\epsilon) \\
&= ()(())\,()
\end{aligned}
$$

parse tree

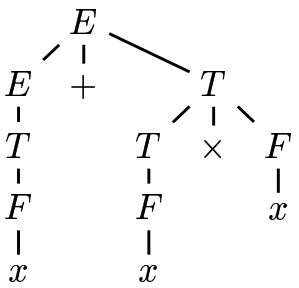


Example: Arithmetic expressions

$(\{E, T, F\}, \{+, \times, (,), x\}, \mathcal{R}, E)$ where $\mathcal{R}$ consists of 6 rules:

$E \rightarrow E + T \mid T \qquad T \rightarrow T \times F \mid F \qquad F \rightarrow (\,E\,) \mid x$

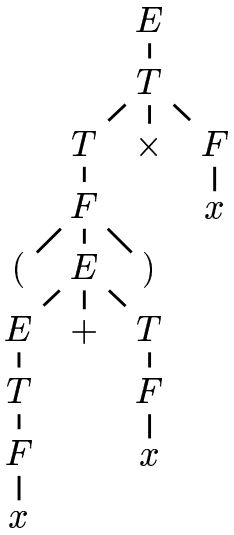
$$
\begin{aligned}
E &\Rightarrow E + \underline{T} \\
&\Rightarrow \underline{E} + T \times F \\
&\Rightarrow T + \underline{T} \times F \\
&\Rightarrow \underline{T} + F \times F \\
&\Rightarrow F + F \times \underline{F} \\
&\Rightarrow F + F \times x \\
&\Rightarrow^* x + x \times x
\end{aligned}
$$



includegraphics[bb=-205 24 819 768,scale=.3]3.jpg

$$E \to E + T \mid T \qquad T \to T \times F \mid F \qquad F \to (\,E\,) \mid x$$

$$\begin{aligned}
E &\Rightarrow \underline{T} \\
&\Rightarrow \underline{T} \times F \\
&\Rightarrow \underline{F} \times F \\
&\Rightarrow (\,\underline{E}\,) \times F \\
&\Rightarrow (\,\underline{E} + T\,) \times F \\
&\Rightarrow (\,\underline{T} + T\,) \times F \\
&\Rightarrow (\,\underline{F} + T\,) \times F \\
&\Rightarrow (\,x + \underline{T}\,) \times \underline{F} \\
&\Rightarrow^* (\,x + x\,) \times x
\end{aligned}$$



Example: A small English language

$$\begin{aligned}
\langle \text{SENTENCE} \rangle &\to \langle \text{NOUN-PHRASE} \rangle\ \langle \text{VERB-PHRASE} \rangle \\
\langle \text{NOUN-PHRASE} \rangle &\to \langle \text{CMPLX-NOUN} \rangle \mid \langle \text{CMPLX-NOUN} \rangle\ \langle \text{PREP-PHRASE} \rangle \\
\langle \text{VERB-PHRASE} \rangle &\to \langle \text{CMPLX-VERB} \rangle \mid \langle \text{CMPLX-VERB} \rangle\ \langle \text{PREP-PHRASE} \rangle \\
\langle \text{PREP-PHRASE} \rangle &\to \langle \text{PREP} \rangle\ \langle \text{CMPLX-NOUN} \rangle \\
\langle \text{CMPLX-NOUN} \rangle &\to \langle \text{ARTICLE} \rangle\ \langle \text{NOUN} \rangle \\
\langle \text{CMPLX-VERB} \rangle &\to \langle \text{VERB} \rangle \mid \langle \text{VERB} \rangle\ \langle \text{NOUN-PHRASE} \rangle \\
\langle \text{ARTICLE} \rangle &\to \text{a} \mid \text{the} \\
\langle \text{NOUN} \rangle &\to \text{boy} \mid \text{girl} \mid \text{flower} \\
\langle \text{VERB} \rangle &\to \text{touches} \mid \text{like} \mid \text{see} \\
\langle \text{PREP} \rangle &\to \text{with}
\end{aligned}$$

10 variables, 9 terminals and 18 rules.

$\langle$SENTENCE$\rangle$ $\Rightarrow$ $\langle$NOUN-PHRASE$\rangle$ $\langle$VERB-PHRASE$\rangle$

$\Rightarrow$ $\langle$CMPLX-NOUN$\rangle$ $\langle$PREP-PHRASE$\rangle$ $\langle$VERB-PHRASE$\rangle$

$\Rightarrow$ $\langle$ARTICLE$\rangle$ $\langle$NOUN$\rangle$ $\langle$PREP-PHRASE$\rangle$ $\langle$VERB-PHRASE$\rangle$

$\Rightarrow$ a girl $\langle$PREP$\rangle$ $\langle$CMPLX-NOUN$\rangle$ $\langle$VERB-PHRASE$\rangle$

$\Rightarrow$ a girl with $\langle$CMPLX-NOUN$\rangle$ $\langle$VERB-PHRASE$\rangle$

$\Rightarrow$ a girl with $\langle$ARTICLE$\rangle$ $\langle$NOUN$\rangle$ $\langle$VERB-PHRASE$\rangle$

$\Rightarrow$ a girl with a flower $\langle$VERB-PHRASE$\rangle$

$\Rightarrow$ a girl with a flower $\langle$CMPLX-VERB$\rangle$

$\Rightarrow$ a girl with a flower $\langle$VERB$\rangle$ $\langle$NOUN-PHRASE$\rangle$

$\Rightarrow$ a girl with a flower likes $\langle$CMPLX-NOUN$\rangle$

$\Rightarrow$ a girl with a flower likes $\langle$ARTICLE$\rangle$ $\langle$NOUN$\rangle$

$\Rightarrow$ a girl with a flower likes the boy

Ambiguity

Say that two derivations are *essentially different* if they determine distinct parse trees. In some CFGs, the *same* string may have essentially different derivations. Call these CFGs ambiguous.

Examples: Ambiguous arithmetic expressions

$$E \rightarrow E + E \mid E \times E \mid x$$

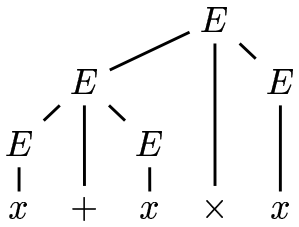
The string $x + x \times x$ has two essentially different derivations:

$$\begin{aligned} E &\Rightarrow \underline{E} \times E \\ &\Rightarrow \underline{E} + E \times E \\ &\Rightarrow x + \underline{E} \times E \\ &\Rightarrow x + x \times \underline{E} \\ &\Rightarrow x + x \times x \end{aligned} \qquad \begin{aligned} E &\Rightarrow \underline{E} + E \\ &\Rightarrow x + \underline{E} \\ &\Rightarrow x + \underline{E} \times E \\ &\Rightarrow x + x \times \underline{E} \\ &\Rightarrow x + x \times x \end{aligned}$$

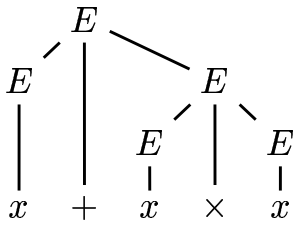
Example: Ambiguous arithmetic expressions

$$E \rightarrow E + E \mid E \times E \mid x$$

$$\begin{aligned} E &\Rightarrow \underline{E} \times E \\ &\Rightarrow \underline{E} + E \times E \\ &\Rightarrow x + \underline{E} \times E \\ &\Rightarrow x + x \times \underline{E} \\ &\Rightarrow x + x \times x \end{aligned}$$



$$\begin{aligned} E &\Rightarrow \underline{E} + E \\ &\Rightarrow x + \underline{E} \\ &\Rightarrow x + \underline{E} \times E \\ &\Rightarrow x + x \times \underline{E} \\ &\Rightarrow x + x \times x \end{aligned}$$



Leftmost derivations

A *leftmost derivation* is one in which at every step, the leftmost occurring variable is the one chosen for replacement.

Example. The two derivations for $x + x \times x$

Definition. A CFG G is *ambiguous* just in case there is some word in $L(G)$ which has two (or more) different leftmost derivations.

Note: Each parse tree of a string identifies a leftmost derivation of it. There is a 1-1 correspondence between parse trees and leftmost derivations.

Exercise. Prove that the 6-rule arithmetic expressions on page 8 is unambiguous.

In general the questions of whether a given CFG is ambiguous, or whether two CFGs are equivalent, are very difficult to answer. (In fact these are *undecidable* decision problems.)

Sometimes the language generated by an ambiguous grammar has an equivalent unambiguous grammar.

Some languages can only be generated by ambiguous grammars. They are called *inherently ambiguous*.

Exercise [*Hard*]. Prove that $\{ 0^i 1^j 2^k : i = j \vee j = k \}$ is inherently ambiguous.

Chomsky Normal Forms

A CFG is in *Chomsky normal form* is every rule has one of the forms:

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

where a is any terminal, and A, B and C are any variables, except that B and C may not be the start variable. In addition we permit $S \rightarrow \epsilon$ where S is the start variable.

Theorem. Any CFG is generated by a CFG in Chomsky normal form.

Non-deterministic pushdown automata

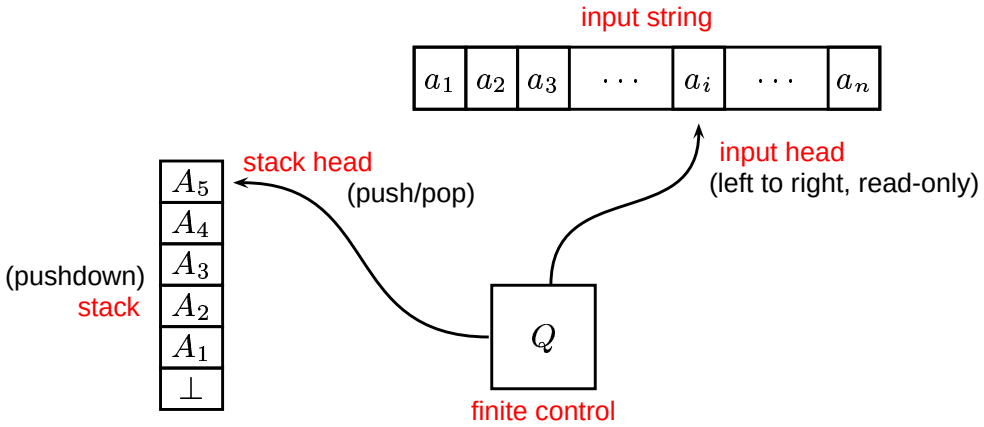
Regular languages are recognized by finite automata,

context free languages are recognized by non-deterministic pushdown automata.

Non-deterministic pushdown automata

(We mostly follow the definitions in Kozen's *Automata and Computability*.)

A (non-deterministic) pushdown automaton is like an NFA, except it has a *stack* (pushdown store) for recording a potentially unbounded amount of information, in a last-in-first-out (LIFO) fashion.



The workings of an NPDA

In each step, the NPDA pops the top symbol off the stack; based on (1) this symbol, (2) the input symbol currently reading, and (3) its current state, it can

1. push a sequence of symbols (possibly ϵ) onto the stack
2. move its read head one cell to the right, and
3. enter a new state

according to the transition rule δ of the machine.

We allow *ϵ-transition*: an NPDA can pop and push without reading the next input symbol or moving its read head.

Note: an NPDA can only access the top of stack symbol in each step.

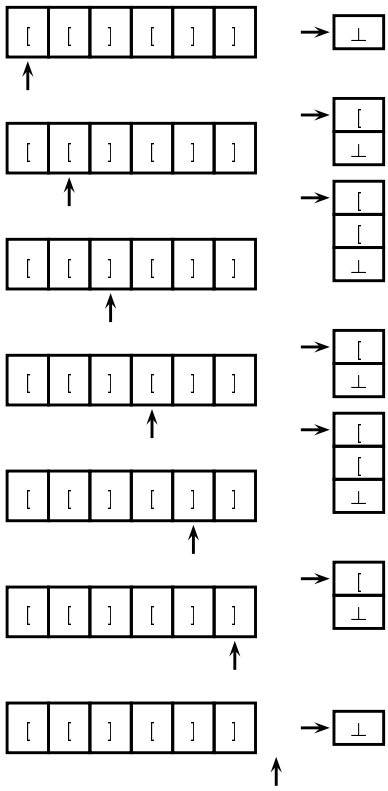
Example: Balanced strings of parentheses

Intuitive description of an NPDA:

1. WHILE <input symbol is "["> DO <push "[" onto the stack>.
2. WHILE <input symbol is "]"> and <top of stack is "["> DO <pop>.
3. IF <all of input read> and <top of stack is "⊥"> THEN <accept>. ("⊥" is initial stack symbol.)

Example: input is "[[ ] [ ]]"

Think of an NPDA as (representing) an algorithm (for a decision problem) with memory access in the form of a stack.



Definition of an NPDA

A *non-deterministic pushdown automaton* (NPDA) is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$ where $Q, \Sigma, \Gamma, \delta$ and F are all finite sets, and

- Q is the set of states
- Σ is the *input alphabet*
- Γ is the *stack alphabet*
- $\delta : Q \times (\Sigma \cup \{ \epsilon \}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ is the transition function
- $q_0 \in Q$ is the start state
- $\perp \in \Gamma$ is the initial stack symbol
- $F \subseteq Q$ is the set of accept states.

Note: An NPDA is strictly more powerful than a *deterministic* PDA. We shall not consider the latter specifically here.

Configuration

A configuration of M is an element of $Q \times \Sigma^* \times \Gamma^*$ describing (1) the current state, (2) the portion of the input yet unread (i.e. under and to the right of the input head) and (3) the current stack contents.

The *start configuration* is $(q_0, w, \perp)$. I.e. M always starts in the start state with its input head scanning the leftmost input symbol and the stack containing only $\perp$.

The *next-configuration relation* $\to$ describes how M moves from one configuration to another in one step. Formally

- If $(q, \gamma) \in \delta(p, a, A)$ then for any $v \in \Sigma^*$ and $\beta \in \Gamma^*$,
$$(p, av, A\beta) \ \to \ (q, v, \gamma\beta)$$
(The input symbol a has been "consumed"; A was popped and γ was pushed, and the new state is q.)
- If $(q, \gamma) \in \delta(p, \epsilon, A)$ then for any $v \in \Sigma^*$ and $\beta \in \Gamma^*$,
$$(p, v, A\beta) \ \to \ (q, v, \gamma\beta)$$
(no input symbol has been "consumed".)

$L(M)$: The language accepted by NPDA M

We define the reflexive, transitive closure of $\rightarrow$, written $\stackrel{*}{\rightarrow}$, as follows:

$$\begin{aligned} C \stackrel{0}{\rightarrow} D &\iff C = D \\ C \stackrel{n+1}{\rightarrow} D &\iff \exists E \,.\, C \stackrel{n}{\rightarrow} E \ \wedge \ E \rightarrow D \end{aligned}$$

and define $C \stackrel{*}{\rightarrow} D$ just if $C \stackrel{n}{\rightarrow} D$ for some $n \geq 0$. I.e. $C \stackrel{*}{\rightarrow} D$ iff D follows from C in 0 or more steps of the relation $\rightarrow$.

Formally we say that M *accepts an input x by final state* if for some $q \in F$ and $\gamma \in \Gamma^*$, we have $(q_0, x, \perp) \stackrel{*}{\rightarrow} (q, \epsilon, \gamma)$. Configurations of the form (q, ϵ, γ) where $q \in F$ and $\gamma \in \Gamma^*$ are called *accepting*.

The *language* of M, written $L(M)$, is defined to be the set of strings accepted by M.

There is another accepting convention:

M *accepts an input x by empty stack* if for some $q \in Q$,

$$(q_0, x, \perp) \stackrel{*}{\rightarrow} (q, \epsilon, \epsilon).$$

N.B. F is irrelevant in the definition of acceptance by empty stack.

The two accepting conventions are equivalent.

Example: An NPDA accepting $\{ ww^R : w \in \{ 0, 1 \}^* \}$

High-level description:

1. Push the input symbols onto the stack, one at a time.
2. Non-deterministically guess that the middle of the string has been reached at some point during 1, and then change into popping off the stack for each symbol read, checking to see if they (i.e. symbols just popped and just read) are the same.
3. If they are always the same symbols, and the stack empties at the same time as the input is finished, accept.

Example: An NPDA accepting $\{ ww^R : w \in \{ 0, 1 \}^* \}$

Implementation-level description:

$$(\{ q_1, q_0, q_2 \}, \underbrace{\{ 0, 1 \}}_{\Sigma}, \underbrace{\{ 0, 1, \perp \}}_{\Gamma}, \delta, q_0, \perp, \{ q_2 \})$$

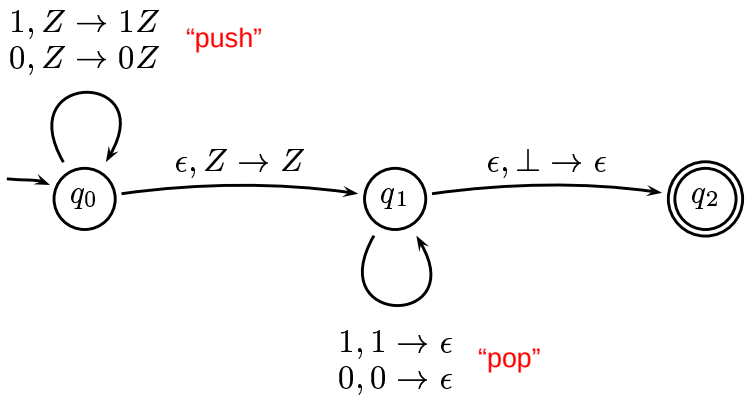
where

$$\delta : \begin{cases} (q_0, 0, Z) & \mapsto & \{ (q_0, 0Z) \} \\ (q_0, 1, Z) & \mapsto & \{ (q_0, 1Z) \} \\ (q_0, \epsilon, Z) & \mapsto & \{ (q_1, Z) \} \\ (q_1, 0, 0) & \mapsto & \{ (q_1, \epsilon) \} \\ (q_1, 1, 1) & \mapsto & \{ (q_1, \epsilon) \} \\ (q_1, \epsilon, \perp) & \mapsto & \{ (q_2, \epsilon) \} \end{cases}$$

where $Z = 0, 1$.

Example: An NPDA accepting $\{ ww^R : w \in \{0,1\}^* \}$

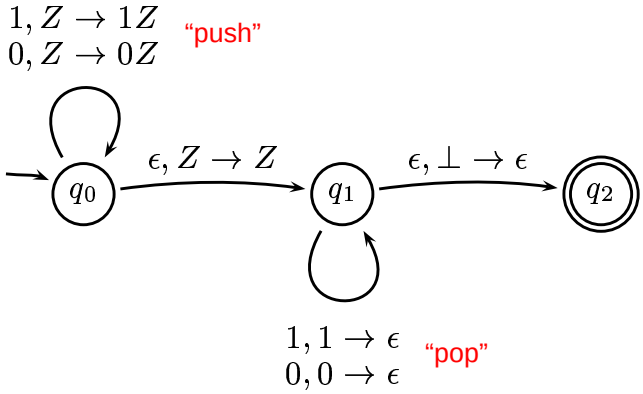
Transition graph:



Notation: In the transition graph, we represent the transition $(q', \gamma) \in \delta(q, a, Z)$ by an edge, labelled by "$a, Z \to \gamma$", that joins node q to q'.

Example: A run accepting the input 011110

$$
\begin{array}{rrrr}
 & (q_0, & 011110, & \perp) \\
\to & (q_0, & 11110, & 0\perp) \\
\to & (q_0, & 1110, & 10\perp) \\
\to & (q_0, & 110, & 110\perp) \\
\to & (q_1, & 110, & 110\perp) \\
\to & (q_1, & 10, & 10\perp) \\
\to & (q_1, & 0, & 0\perp) \\
\to & (q_1, & \epsilon, & \perp) \\
\to & (q_2, & \epsilon, & \epsilon)
\end{array}
$$



Example: An NPDA accepting balanced strings of parentheses

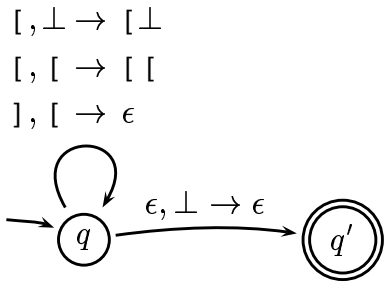
Implementation-level description:

$(\{q, q'\}, \{[,]\}, \{\bot, [\}, \delta, q, \bot, \{q'\})$

where

$$\delta \begin{cases} (q, [, \bot) & \mapsto & \{(q, [\bot)\} \\ (q, [, [) & \mapsto & \{(q, [[)\} \\ (q,], [) & \mapsto & \{(q, \epsilon)\} \\ (q, \epsilon, \bot) & \mapsto & \{(q', \epsilon)\} \end{cases}$$

Transition diagram:



Equivalence between NPDAs and context-free languages

A major result in automata theory is:

Theorem. A language is context-free iff some NPDA accepts it.

Proof overview We are breaking down the proof into the following steps:

A Given a CFG G, there is an equivalent NPDA P_G.

B Given an NPDA N, there is an equivalent CFG G_N generating $L(N)$.

1 Every NPDA can be simulated by an NPDA with one state

2 Every NPDA with one state has an equivalent CFG.

Lemma. Given a CFG G, there is an equivalent NPDA P_G.

Proof idea: The stack alphabet of P_G consists of the terminal and variable symbols and $\perp$. We describe the action of P_G informally:

1. Place the start variable symbol on the stack.
2. Repeat forever: Pop top-of-stack x. Cases of:
   (a) x is a variable A: Nondeterministically select a rule for A and replace A by the string w (say) on the rhs of the rule (so that the leftmost symbol of w is at the top of stack).
   (b) x is a terminal a: Read the next input symbol and compare it with a. If they do not match, then exit (and reject this branch of the nondeterminism).
   (c) $x = \perp$: Enter the accept state.

Claim: P_G accepts $L(G)$.

NPDA that accepts CFL generated by $S \to a\,S\,b\,S \mid b\,S\,a\,S \mid \epsilon$

Implementation-level description:

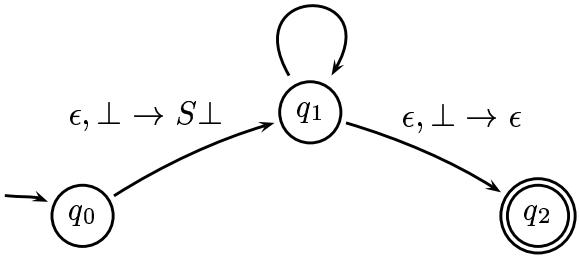
$(\{q_0, q_1, q_2\}, \{a, b\}, \{\perp, S, a, b\}, \delta, q_0, \perp, \{q_2\})$

where δ is given by

$$
\begin{aligned}
\delta(q_0, \epsilon, \perp) &= \{(q_1, S\perp)\} \\
\delta(q_1, \epsilon, S) &= \{(q_1, aSbS), \\
&\qquad (q_1, bSaS), \\
&\qquad (q_1, \epsilon)\} \\
\delta(q_1, a, a) &= \{(q_1, \epsilon)\} \\
\delta(q_1, b, b) &= \{(q_1, \epsilon)\} \\
\delta(q_1, \epsilon, \perp) &= \{(q_2, \epsilon)\}
\end{aligned}
$$

Transition diagram:

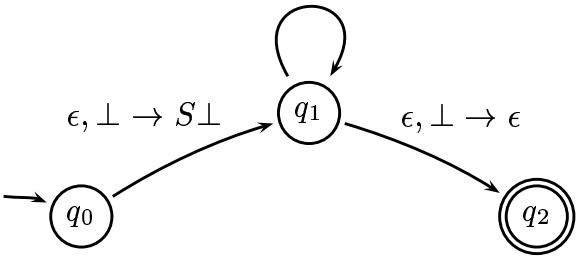
$\epsilon, S \to aSbS$
$\epsilon, S \to bSaS$
$\epsilon, S \to \epsilon$
$a, a \to \epsilon$
$b, b \to \epsilon$



Example: A run accepting the input $abab$

$$
\begin{array}{lllll}
 & (q_0, & abab, & \bot) & \\
\to & (q_1, & abab, & S\bot) & \\
\to & (q_1, & abab, & aSbS\bot) & (1) \\
\to & (q_1, & bab, & SbS\bot) & \\
\to & (q_1, & bab, & bSaSbS\bot) & (2) \\
\to & (q_1, & ab, & SaSbS\bot) & \\
\to & (q_1, & ab, & aSbS\bot) & (3) \\
\to & (q_1, & b, & SbS\bot) & \\
\to & (q_1, & b, & bS\bot) & (3) \\
\to & (q_1, & \epsilon, & S\bot) & \\
\to & (q_1, & \epsilon, & \bot) & (3) \\
\to & (q_2, & \epsilon, & \epsilon) &
\end{array}
$$

(1) $\epsilon, S \to aSbS$
(2) $\epsilon, S \to bSaS$
(3) $\epsilon, S \to \epsilon$
$a, a \to \epsilon$
$b, b \to \epsilon$



Leftmost derivation: $S \to aSbS \to abSaSbS \to abaSbS \to ababS \to abab$

Simulating NPDAs by CFGs

We do this in two steps:

1. Every NPDA can be simulated by an NPDA with one state
2. Every NPDA with one state has an equivalent CFG.

For 2: Take a one-state NPDA $M = (\{ q \}, \Sigma, \Gamma, \delta, q, \bot, \emptyset)$ that accepts by empty stack. Define

$$G_M \;=\; (\Gamma, \Sigma, P, \bot)$$

where P contains a rule

$$A \;\to\; c\, B_1 \cdots B_k$$

for every transition $(q, B_1 \cdots B_k) \in \delta(q, c, A)$ where $c \in \Sigma \cup \{ \epsilon \}$. Then we have $L(M) = L(G_M)$.

Every NPDA can be simulated by a one-state NPDA

Idea: maintain all state information on the stack. W.l.o.g. we assume M is of the form $(Q, \Sigma, \Gamma, \delta, s, \perp, \{ t \})$, and M can empty its stack after entering final state t.

Set $\Gamma' = Q \times \Gamma \times Q$ (elements are written $\langle p\, A\, q \rangle$). We construct a new NPDA

$$M' \;=\; (\{ * \}, \Sigma, \Gamma', \delta', *, \langle s \perp t \rangle, \emptyset)$$

that accepts by empty stack. For each transition

$$(q_0, B_1 \cdots B_k) \in \delta(p, c, A)$$

where $c \in \Sigma \cup \{ \epsilon \}$, include in δ' the transition

$$(*, \langle q_0\, B_1\, q_1 \rangle \langle q_1\, B_2\, q_2 \rangle \cdots \langle q_{k-1}\, B_k\, q_k \rangle) \in \delta'(*, c, \langle p\, A\, q_k \rangle)$$

for all possible choices of $q_1, \cdots, q_k$.

Intuitively M' simulates M, guessing non-deterministically what state M will be in at certain future points in the computation, saving those guesses on the stack, and then verifying later that the guesses were correct.

Lemma. M' can scan a string x starting with only $\langle p\, A\, q \rangle$ on its stack and end up with an empty stack, if and only if M can scan x starting in stack p with only A on its stack and end up in state q with an empty stack. I.e. we have

$$(p, x, A) \;\stackrel{n}{\rightarrow}_M\; (q, \epsilon, \epsilon) \qquad \Longleftrightarrow \qquad (*, x, \langle p\, A\, q \rangle) \;\stackrel{n}{\rightarrow}_{M'}\; (*, \epsilon, \epsilon)$$

It then follows that $L(M) = L(M')$.

Closure Properties of Context-Free Languages

Theorem: Context-free languages are closed under the regular operations: union, concatenation and star..

Proof idea. Let $G_1 = (\Gamma_1, \Sigma, \mathcal{R}_1, S_1)$ and $G_2 = (\Gamma_2, \Sigma, \mathcal{R}_2, S_2)$ be context free grammars with $\Gamma_1 \cap \Gamma_2 = \emptyset$. Consider the following context free grammars

$$\begin{aligned} G_{\text{union}} &= (\Gamma_1 \cup \Gamma_2 \cup \{ S \}, \Sigma, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{ S \to S_1, S \to S_2 \}, S) \\ G_{\text{concat}} &= (\Gamma_1 \cup \Gamma_2 \cup \{ S \}, \Sigma, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{ S \to S_1\, S_2 \}, S) \\ G_{\text{star}} &= (\Gamma_1 \cup \{ S \}, \Sigma, \mathcal{R}_1 \cup \{ S \to S_1\, S, S \to \epsilon \}, S) \end{aligned}$$

where S is a fresh variable. Then

$$\begin{aligned} L(G_{\text{union}}) &= L(G_1) \cup L(G_2) \\ L(G_{\text{concat}}) &= L(G_1) \cdot L(G_2) \\ L(G_{\text{star}}) &= L(G_1)^* \end{aligned}$$

Concluding remarks for Part One
Outlook on Part Two

Regular languages

- Finite automata, regular expressions and right-linear grammars
- Pumping Lemma and Myhill-Nerode Theorem

Contex-free languages

- Non-deterministic push down automata and context-free grammars
- Pumping Lemma for context-free languages

Decidable and recursively enumerable languages

- Turing machines [and unrestricted grammars]
- Halting Problem and other undecidable problems

Language	**regular**	**context-free**	**semi-decidable/r.e.**
Grammar rules	right-linear $A \to wB, A \to w$	context-free $A \to \alpha$	unrestricted $\alpha \to \beta$
Machine memory	DFA or NFA finite	NPDA finite + one stack	Turing machine unrestricted
Other Descr^{n}	Regular expression Myhill-Nerode-Thm		λ-calculus μ-recursive funct^{s} Hugs, Oberon, ...
Example	$\{ w : w$ contains $10 \}$ $\{ 0^n 1^m : n, m \geq 0 \}$	$\{ w\, w^R \}$ $\{ 0^n 1^n : n \geq 0 \}$	$\{ 0^n 1^m 0^{n+m} : n \geq 0 \}$ $\{ Mx : M$ halts on $x \}$ HP
Counterexample	$\{ 0^n 1^n : n \geq 0 \}$	$\{ 0^n 1^n 0^n : n \geq 0 \}$ $\{ w\, w \}$	$\{ M : M$ halts on every $x \}$ TP
Application	Tokenizer Model checker	Parser	General computing

The Pumping Lemma for context-free languages

Pumping Lemma for CFGs. If L is a context free language, then there is a number p – the *pumping length* – such that if $w \in L$ of length at least p, then w may be divided into five pieces, $w = u\,x\,y\,z\,v$, satisfying:

(i) for each $i \geq 0$, $u\,x^i\,y\,z^i\,v \in L$

(ii) $|xz| > 0$

(iii) $|x\,y\,z| \leq p$.

This can be used to show that the following languages are not context-free

- $\{\, w\,w : w \in \{0,1\}\,\}$ **Idea**: use $w = 0^p\,1^p\,0^p\,1^p$
- $\{\, 0^n 1^n 0^n : n \geq 0\,\}$ **Idea**: use $w = 0^p\,1^p\,0^p$

Regular Operations and Context-Free Languages

Theorem: Context-free languages are closed under the regular operations: union, concatenation and star.

Proof idea. Let $G_1 = (\Gamma_1, \Sigma, \mathcal{R}_1, S_1)$ and $G_2 = (\Gamma_2, \Sigma, \mathcal{R}_2, S_2)$ be context free grammars with $\Gamma_1 \cap \Gamma_2 = \emptyset$. Consider the following context free grammars

$$\begin{aligned} G_{\text{union}} &= (\Gamma_1 \cup \Gamma_2 \cup \{\, S \,\}, \Sigma, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{\, S \to S_1, S \to S_2 \,\}, S) \\ G_{\text{concat}} &= (\Gamma_1 \cup \Gamma_2 \cup \{\, S \,\}, \Sigma, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{\, S \to S_1\,S_2 \,\}, S) \\ G_{\text{star}} &= (\Gamma_1 \cup \{\, S \,\}, \Sigma, \mathcal{R}_1 \cup \{\, S \to S_1\,S, S \to \epsilon \,\}, S) \end{aligned}$$

where S is a fresh variable. Then

$$\begin{aligned} L(G_{\text{union}}) &= L(G_1) \cup L(G_2) \\ L(G_{\text{concat}}) &= L(G_1) \cdot L(G_2) \\ L(G_{\text{star}}) &= L(G_1)^* \end{aligned}$$

Intersection and context-free languages

1. Context-free languages are not closed under intersection.
2. The intersection of a context-free language with a regular set is context-free.

Proof Idea

1. Both $A = \{ 0^n 1^n 0^m : n, m \geq 0 \}$ and $B = \{ 0^n 1^m 0^m : n, m \geq 0 \}$ are context-free. But $A \cap B = \{ 0^n 1^n 0^n : n \geq 0 \}$ is not context-free.
2. Let A be a regular set accepted by NFA $M = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and let B be a context-free language accepted by NPDA $N = (Q_2, \Sigma, \Gamma, \delta_2, q_2, \bot, F_2)$.
   Define the "product"-automaton
   $P = (Q_1 \times Q_2, \Sigma, \Gamma, \delta, (q_1, q_2), \bot, F_1 \times F_2$ where δ is given by

   $$\delta((r,t), a, A) = \{ ((r', t'), \alpha) : r' \in \delta_1(r, a), (t', \alpha) \in \delta_2(t, a, A) \}$$

   then P is a NPDA that accepts $A \cap B$.

Complementation and context-free languages

Context-free languages are not closed under complementation.

Proof Idea The set $A = \{ w\, w : w \in \{0,1\} \}$ is not context-free, but its complement $B = \{0,1\}^* \setminus A$ is context-free, as it is generated by the grammar $(\{ S, A, B, C \}, \{ 0, 1 \}, \mathcal{R}, S)$

whith rule set $\mathcal{R} = \{ \begin{array}{lcl} S & \to & AB \mid BA \mid A \mid B, \\ A & \to & CAC \mid 0, \\ B & \to & CBC \mid 1, \\ C & \to & 0 \mid 1 \ \} \end{array}$

This grammar generates
all strings of odd length starting with productions $S \to A$ or $S \to B$ or
strings of the form $x0yu1v$ or $u1vx0y$ where $x, y, u, v \in \{0,1\}^*$, $|x| = |y|$ and $|u| = |v|$. None of these strings can be of the form $w\, w$.

Turing Machines and Effective Computability

We introduce the most powerful of the automata we will study: Turing machines (TMs). TMs can compute any function normally considered computable; indeed we can define *computable* to mean computable by a TM.

Informal description of a Turing Machine

A (one-tape deterministic) TM consists of:

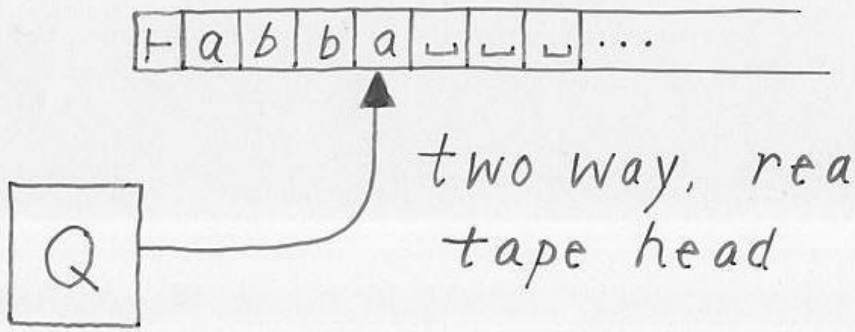
- a finite *input alphabet* Σ, a finite *tape alphabet* Γ such that $\Sigma \subseteq \Gamma$
- a finite set of *states* Q
- a *semi-infinite tape* of cells (infinite to the right)
- a *tape head* that can move left and right over the tape, reading and writing symbols onto tape cells

At the start of computation, contents of the tape are

$$\vdash \; w_1 \cdots w_n \sqcup \sqcup \sqcup \cdots$$

where $w = w_1 \cdots w_n$ is the input string, and $\vdash \in \Gamma$ is the left endmarker. The tape head is over $\vdash$, and the infinitely many cells to the right of the input all contain a special blank symbol $\sqcup \in \Gamma$.

Turing Machine



Workings of a Turing Machine

The machine starts in its start state with its head scanning the leftmost cell.

At each step, it reads the symbol under its head, and depending on that symbol and the current state, it writes a new symbol on that tape cell, then moves its head either left or right one cell, and enters a new state. The action is determined by a *transition function* δ.

It accepts its input by entering the accept state, and rejects by entering the reject state.

On some input, it may run infinitely without ever accepting or rejecting i.e. it *loops* on that input.

Note. Definitions of TMs in the literature differ slightly in nonessential details. Here we use Kozen's definition.

Definition of a Turing Machine

A (one-tape deterministic) *Turing machine* (TM) is a 9-tuple

$$(Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}})$$

where

- Q is a finite set (the *states*)
- Σ is a finite set (the *input alphabet*)
- Γ is a finite set (the *tape alphabet*) containing Σ
- $\vdash \in \Gamma - \Sigma$, the *left endmarker*
- $\sqcup \in \Gamma - \Sigma$, the *blank symbol*
- $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{ L, R \}$ is a finite function (the *transition function*)
- $q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}} \in Q$ are respectively the *start*, *accept* and *reject* states, with $q_{\mathrm{acc}} \neq q_{\mathrm{rej}}$.

Some conventions

Intuitively $\delta(q, a) = (q', b, L)$ means "when in state q scanning symbol a, write b over the tape cell, move the head left by one cell, and enter state q'".

Restrictions

1. The left endmarker is never overwritten, and the machine never moves left of the endmarker. I.e. for all $p \in Q$ there exists $q \in Q$ such that

$$\delta(p, \vdash) = (q, \vdash, R).$$

2. Once the machine enters q_{acc} it never leaves it, and similarly for q_{rej}. I.e. for all $b \in \Gamma$, there exist $c, c' \in \Gamma$ and $D, D' \in \{ L, R \}$ such that

$$\begin{aligned} \delta(q_{\mathrm{acc}}, b) &= (q_{\mathrm{acc}}, c, D) \\ \delta(q_{\mathrm{rej}}, b) &= (q_{\mathrm{rej}}, c', D') \end{aligned}$$

Example: A TM that accepts $\{ a^n b^n c^n : n \geq 0 \}$ (not context-free)

Informal high-level description.

At start state, it scans right over the input string, checking that it matches $a^* b^* c^*$, and *writing nothing* (i.e. overwriting with same letter) on the way across. When it sees the first $\sqcup$, it overwrites it with a right endmarker $\dashv$.

Now it scans left, *erasing* the first c (i.e. overwriting it with $\sqcup$) it sees, then the first b it sees, then the first a it sees, until it reaches $\vdash$.

It then scans right, erasing one a, one b and one c. It continues to scan left and right, erasing one occurrence of each letter in one pass.

If in some pass, it sees at least one occurrence of one letter and no occurrence of another, it rejects. Otherwise it eventually erases all letters and makes one pass between $\vdash$ and $\dashv$ seeing only blanks, at which point it accepts.

Implementation-level description. Formally the TM is

$$(\underbrace{\{1, 2, \cdots, 10, q_0, q_{\text{acc}}, q_{\text{rej}}\}}_{Q}, \underbrace{\{a, b, c\}}_{\Sigma}, \underbrace{\Sigma \cup \{\vdash, \sqcup, \dashv\}}_{\Gamma}, \vdash, \sqcup, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$$

where δ is

	$\vdash$	a	b	c	$\sqcup$	$\dashv$
q_0	$(q_0, \vdash, R)$	(q_0, a, R)	$(1, b, R)$	$(2, c, R)$	$(3, \dashv, L)$	$-$
1	$-$	$(q_{\text{rej}}, -, -)$	$(1, b, R)$	$(2, c, R)$	$(3, \dashv, L)$	$-$
2	$-$	$(q_{\text{rej}}, -, -)$	$(q_{\text{rej}}, -, -)$	$(2, c, R)$	$(3, \dashv, L)$	$-$
3	$(q_{\text{acc}}, -, -)$	$(q_{\text{rej}}, -, -)$	$(q_{\text{rej}}, -, -)$	$(4, \sqcup, L)$	$(3, \sqcup, L)$	$-$
4	$(q_{\text{rej}}, -, -)$	$(q_{\text{rej}}, -, -)$	$(5, \sqcup, L)$	$(4, c, L)$	$(4, \sqcup, L)$	$-$
5	$(q_{\text{rej}}, -, -)$	$(6, \sqcup, L)$	$(5, b, L)$	$-$	$(5, \sqcup, L)$	$-$
6	$(7, \vdash, R)$	$(6, a, L)$	$-$	$-$	$(6, \sqcup, L)$	$-$
7	$-$	$(8, \sqcup, R)$	$(r, -, -)$	$(r, -, -)$	$(7, \sqcup, R)$	$(q_{\text{acc}}, -, -)$
8	$-$	$(8, a, R)$	$(9, \sqcup, R)$	$(r, -, -)$	$(8, \sqcup, R)$	$(q_{\text{rej}}, -, -)$
9	$-$	$-$	$(9, b, R)$	$(10, \sqcup, R)$	$(9, \sqcup, R)$	$(q_{\text{rej}}, -, -)$
10	$-$	$-$	$-$	$(10, c, R)$	$(10, \sqcup, R)$	$(3, \dashv, L)$

E.g. Input tape at start is $\vdash\ a\, b\, c\, \sqcup\, \sqcup\, \cdots$.

An accepting run:

$$
\begin{array}{l}
(\epsilon, q_0, \vdash a\, b\, c) \\
(\vdash, q_0, a\, b\, c) \\
(\vdash a, q_0, b\, c) \\
(\vdash a\, b, 1, c) \\
(\vdash a\, b\, c, 2, \sqcup) \\
(\vdash a\, b, 3, c\ \dashv) \\
(\vdash a, 4, b\, \sqcup\, \dashv) \\
(\vdash, 5, a\, \sqcup\, \sqcup\, \dashv) \\
(\epsilon, 6, \vdash \sqcup\, \sqcup\, \sqcup\, \dashv) \\
(\vdash, 7, \sqcup\, \sqcup\, \sqcup\, \dashv) \\
(\vdash \sqcup, 7, \sqcup\, \sqcup\, \dashv) \\
(\vdash \sqcup\, \sqcup, 7, \sqcup\, \dashv) \\
(\vdash \sqcup\, \sqcup\, \sqcup, 7, \dashv) \\
(\vdash \sqcup\, \sqcup\, \sqcup, q_{\mathrm{acc}}, \dashv)
\end{array}
$$

Configurations

As a TM computes, changes occur in the current state, the current tape contents, and the current head position. A description of these items is called a *configuration* of the TM, often represented as a triple

$$(u, q, v)$$

for the configuration where the current state is q, current tape content is $u\,v$ and the current head location is over the first symbol of v.

Let C and C' be configurations. We define the *next-configuration relation* $\rightarrow$ (we read $C \rightarrow C$ as "C *yields* C'", meaning that the TM can legally go from C to C' in one step) formally as:

- $(ua, q, bv) \rightarrow (u, q', acv)$ if $\delta(q, b) = (q', c, L)$
- $(ua, q, bv) \rightarrow (uac, q', v)$ if $\delta(q, b) = (q', c, R)$

Special case: If the head is at the right end of the configuration, (ua, q, ϵ) is equivalent to $(ua, q, \sqcup)$.

We define $\xrightarrow{*}$, the reflexive, transitive closure of $\longrightarrow$, inductively:

- $C \xrightarrow{0} C$, for all configurations C
- $C \xrightarrow{n+1} C''$ if $C \xrightarrow{n} C'$ and $C' \longrightarrow C''$

We define $C \xrightarrow{*} C'$ (read "C can yield C' in finitely many steps") to be $C \xrightarrow{n} C'$ for some $n \geq 0$.

The language accepted by a TM

The *start configuration* of TM M on input w is $(\epsilon, q_0, \vdash w)$.

An *accepting configuration* is any configuration with state q_{acc}; a *rejecting configuration* is any configuration with state q_{rej}.

Turing machine M is said to *accept* input w just in case $(\epsilon, q_0, \vdash w) \xrightarrow{*} C$ where C is some accepting configuration.

The *language* of M, written $L(M)$, is defined to be the collection of strings that M accepts.

Call a language *recursively enumerable* (or simply *r.e.*), if some TM accepts it.

When we start a TM M on an input, three outcomes are possible: M may accept, reject or loop (i.e. never terminate). Call M *total* (or a *decider*) if it halts on all inputs i.e. it never loops.

Call a language *decidable* (or *recursive*) if some total Turing machine accepts it.

Example: A TM that accepts $\{ ww : w \in \{ a, b \}^* \}$

This is a non-context-free but decidable language.

High-level description: $\Gamma = \{ a, b, \vdash, \dashv, \sqcup, \grave{a}, \grave{b}, \acute{a}, \acute{b} \}$

Two stages:

Stage I: Marking.

On input x, it scans out to the first blank symbol, making sure that x is of even length, and rejecting immediately if not. It then lays down a right endmarker $\dashv$, and repeatedly scans back and forth over the input.

In each pass from right to left, it marks the first unmarked a or b it sees with $\acute{}$ (i.e. overwriting a and b by $\acute{a}$ and $\acute{b}$ respectively). In each pass from left to right, it marks the first unmarked a or b it sees with $\grave{}$. It continues until all symbols are marked.

The reason: so that the centre of the string can be identified.

E.g. initially the tape contents are

$$\vdash \quad a \quad a \quad b \quad b \quad a \quad a \quad a \quad b \quad b \quad a \quad \sqcup \quad \sqcup \quad \sqcup \quad \cdots$$

At the end of the marking stage:

$$\vdash \quad \grave{a} \quad \grave{a} \quad \grave{b} \quad \grave{b} \quad \grave{a} \quad \acute{a} \quad \acute{a} \quad \acute{b} \quad \acute{b} \quad \acute{a} \quad \dashv \quad \sqcup \quad \sqcup \quad \cdots$$

Stage II: Erasing.

Then it repeatedly scans left and right: In each pass from the left, it erases the first symbol it sees marked with ` but remembers that symbol. It then scans forward until it sees the first symbol marked with ´, checks that it is the same, and erases it, otherwise it rejects.

When it has erased all symbols, it accepts.

Stage II of the example:

$$
\begin{array}{cccccccccccccc}
\vdash & \grave{a} & \grave{a} & \grave{b} & \grave{b} & \grave{a} & \acute{a} & \acute{a} & \acute{b} & \acute{b} & \acute{a} & \dashv & \sqcup & \sqcup & \cdots \\
\vdash & \sqcup & \grave{a} & \grave{b} & \grave{b} & \grave{a} & \sqcup & \acute{a} & \acute{b} & \acute{b} & \acute{a} & \dashv & \sqcup & \sqcup & \cdots \\
\vdash & \sqcup & \sqcup & \grave{b} & \grave{b} & \grave{a} & \sqcup & \sqcup & \acute{b} & \acute{b} & \acute{a} & \dashv & \sqcup & \sqcup & \cdots \\
\vdash & \sqcup & \sqcup & \grave{b} & \grave{b} & \grave{a} & \sqcup & \sqcup & \acute{b} & \acute{b} & \acute{a} & \dashv & \sqcup & \sqcup & \cdots \\
\vdash & \sqcup & \sqcup & \sqcup & \grave{b} & \grave{a} & \sqcup & \sqcup & \sqcup & \acute{b} & \acute{a} & \dashv & \sqcup & \sqcup & \cdots \\
\vdash & \sqcup & \sqcup & \sqcup & \sqcup & \grave{a} & \sqcup & \sqcup & \sqcup & \sqcup & \acute{a} & \dashv & \sqcup & \sqcup & \cdots \\
\vdash & \sqcup & \sqcup & \sqcup & \sqcup & \sqcup & \sqcup & \sqcup & \sqcup & \sqcup & \sqcup & \dashv & \sqcup & \sqcup & \cdots
\end{array}
$$

Non-deterministic Turing Machines (NDTM)

At any point of a computation, a NDTM may proceed in one of several possible ways, so that the transition function has type

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

The computation of a NDTM is a tree, whose branches correspond to different possible runs of the machine.

If any branch leads to the accept state, the machine is said to accept its input.

Lemma. A language is acceptable by some TM iff it is acceptable by some NDTM.

Deterministic Multitape TMs

k-tape TMs have k semi-infinite tapes (numbered $1, 2, \cdots, k$), each with its own independent read/write tape head. Initially the input occupies the first tape, and the other tapes are blank. In each step, the machine reads the k symbols under its heads, and depending on this information and the current state, it writes a symbol on each tape, moves the heads (they don't all have to move in the same direciton). amd enters a new state. Its transition function is of type

$$\delta : Q \times \Gamma^3 \rightarrow Q \times \Gamma^3 \times \{ L, R \}^3$$

Simulting a 3-tape TM. Given a 3-tape TM M, we build a single-tape TM N with $\Sigma_N = \Sigma_M$, and an expanded tape alphabet

$$\Gamma_N = \Sigma_M \cup \{ \vdash \} \cup (\Gamma_M \cup \widehat{\Gamma_M})^3$$

where $\widehat{\Gamma_M} = \{ \widehat{a} : a \in \Gamma_M \}$, allowing us to think of its tape as divided into three tracks. Each track will contain the contents of one of M's tape.

Each track has only one marked symbol $\widehat{a}$, indicating the position of the corresponding tape head. M configuration might be simulated by the following configuration of N:

On input $x = a_1 \cdots a_n$, N starts with tape contents $\vdash a_1 \cdots a_n \sqcup\sqcup \cdots$. It first copies the input to its top track, and fills in the bottom two tracks with blanks; it also shifts everything right one cell so that it can fill in the leftmost cell with the simulated left end of M.

Each step of M is simulated by several steps of N: N starts at the left of the tape, then scans out until it sees all three marks, remembering the marked symbols in its finite control. It then determines what to do according to the encoded δ_M: it goes back to all three marks, rewriting the symbols on each track and moving the marks appropriately. It then returns to the left end of the tape, ready to simulate the next step of M.

The definition of Algorithm

Informally an *algorithm* is a collection of simple instructions for performing some task. They are sometimes called *procedures* or *recipes*.

To have an algorithm for a problem is to know a way of *effectively computing* or solving the problem.

There are many examples: long division, “Sieve of Eratosthenes” (for finding prime numbers), Euclid’s greatest common divisor algorithm, etc.

What are algorithms? We know it when we see one.

But can we define precisely what they are?

Why is it important to have a definition of algorithm?

A story: Hilbert's 10th Problem

At the International Congress of Mathematicians, Paris, 1900, David Hilbert famously identified 23 problems in Mathematics and posed them as a challenge for the coming century.

Hilbert's 10th Problem. Devise an *algorithm*[a] that tests whether a polynomial has an integral root.

[a] In Hilbert's words: "...a process according to which it can be determined by a finite number of operations."

E.g. $p(x, y) = 6x^3yz^2 + 3xy^2 - x^3 - 10$ has four terms over the variables x, y, z, and has a root at $(x, y, z) = (5, 3, 0)$.

Hilbert apparently assumed that an algorithm must exist - someone need only find it.

Effective computability

Various algorithms for computing certain problems effectively were known, but a general definition of “effectively computable” that could distinguish the computable and the noncomputable was sought.

Various formalisms have been proposed:

- Turing machines (Alan Turing 1936)
- Post systems (Emil Post)
- μ-recursive functions (Kurt Gθdel, Jacques Herbrand)
- λ-calculus (Alonzo Church, Stephen C. Kleene)
- Cominbatory logic (Moses Schθnfinkel and Haskell B. Curry)

Church-Turing Thesis

Amazingly

Theorem. All the preceding formalisms are equivalent. I.e. a problem is *solvable* (or *decidable*) w.r.t. one formalism iff it is solvable by any of the other.

This remarkable coincidence says that there is only one notion of computability.

Church-Turing Thesis. Intuitive notions of algorithms = Turing machines. □

The Church-Turing Thesis is not an assertion that can be proved - it is not a theorem. Rather, it may be regarded as a definition (of algorithm).

The Thesis is widely accepted by mathematicians and computer scientists.

Hilbert's 10th Problem: Matijasevich's breakthrough

We now know:

Theorem. (Matijasevich, 1970) Hilbert's 10th Problem is algorithmically unsolvable. I.e. no algorithm (= Turing machine) exists that solves the Problem.

Without a clear definition of algorithms, it would have been impossible for mathematicians of Hilbert's era to come to the same conclusion.

We need to know what algorithms are i.e. have an accepted definition of algorithm, before we can prove that none exists for solving a given problem.

Phrasing Hilbert's 10th Problem in our terminology

Let $D = \{ p : p$ is a polynomial with an integral root $\}$.

Hilbert's 10th Problem asks in essence whether the set D is *decidable*. [Formally we fix an alphabet Σ and code p as a string over it, to obtain a *language* corresponding to D.]

The answer (thanks to Matijasevich) is negative. However D is r.e.

We consider a simpler problem:

$$D_1 = \{ p : p \text{ is a polynomial over } x \text{ with an integral root } \}.$$

We define a TM M_a that accepts D_1: On input p (a polynomial over x)

> Evaluate p with x set successively to $0, 1, -1, 2, -2, \cdots$. If at any point the polynomial evaluates to 0, *accept*.

If D_1 has an integral root, M_1 will eventually find it, and accept. If not, M_1 will run forever. There is a similar TM M that accepts D: here M runs through all

possible settings of its variables to integral values.

Both M_1 and M are not deciders.

But M_1 can be converted to a decider because we can calculate bounds within which roots of a single-variable polynomial must lie and restrict the search accordingly. They are

$$\left[-k\frac{c_{\max}}{c_1}, k\frac{c_{\max}}{c_1}\right]$$

where k is the number of terms, $c_{\max}$ is the coefficient with the largest absolute value and c_1 is the coefficient of the highest order term.

E.g. the bounds for $4x^3 - 7$ are $[-2, 2]$.

Matijasevich's theorem shows that no such bounds for multivariate polynomials can be calculated.

Decidability

We investigate the power of algorithms to solve problems. We demonstrate that certain problems can be solved algorithmically and others cannot. Our objective is to explore the limits of algorithmic solvability.

Coding decision problems as languages

Decision problems are problems that expect a Yes/No answer. E.g.

1. PRIME: Given a number, is it prime?
2. CYCLIC: Given a graph, is it cyclic?
3. DFA ACCEPTANCE: Given a DFA B and an input w, does B accept w?

We represent decision problems as languages.

E.g. the language representing CYCLIC is

$$\{ \langle G \rangle : G \text{ is a cyclic graph} \}$$

where $\langle G \rangle$ denotes an encoding of G as a string over $\Sigma = \{ 0, 1, (,), \# \}$(say). The graph $(\{ 1, 2, 3 \}, \{ (1, 2), (2, 3), (3, 1) \})$ can be encoded as

$$(1\#10\#11)\#((1\#10)\#(10\#11)\#(11\#1)).$$

We say that a decision problem is *decidable* if the corresponding language is (Turing-machine) decidable.

DFA Acceptance Problem

DFA Acceptance Problem: Given a DFA B and an input w, does B accept w?

The corresponding language is

$$A_{\mathrm{DFA}} = \{ \langle B, w \rangle : B \text{ is a DFA that accepts input string } w \}.$$

Lemma. A_{DFA} is a decidable language.

Proof. We construct a TM M that decides A_{DFA}: On input $\langle B, w \rangle$ where B is a DFA and w an input

1. Simulate B on input w
2. If the simulation ends in an an accept state, *accept*. If it ends in a non-accepting state, *reject*.

Convince yourself that it follows that A_{NFA} and A_{regexp} (acceptance problems for NFA and regular expressions respectively) are also decidable.

Emptiness Problem for DFA: Given a DFA A, is $L(A)$ empty?

The corresponding language is

$$E_{\mathrm{DFA}} = \{ \langle A \rangle : A \text{ is a DFA such that } L(A) = \emptyset \}$$

Lemma. E_{DFA} is a decidable language.

Proof. We design a TM T that uses a marking algorithm. On input $\langle A \rangle$ where A is a DFA:

1. Mark the start state of A.
2. Repeat until no new states get marked:

   Mark any state that has a transition coming into it from any marked state.
3. If no accept state is marked, *accept*; otherwise *reject*.

Equivalence Problem for DFA: Given two DFAs, are they equivalent?

The corresponding language:

$EQ_{\mathrm{DFA}} = \{ \langle A, B \rangle : A \text{ and } B \text{ are DFA s.t. } L(A) = L(B) \}$.

Lemma. EQ_{DFA} is a decidable language.

Proof. Key idea: For sets P and Q, $P \subseteq Q$ iff $P \cap \overline{Q} = \emptyset$. We use T, the algorithm for E_{DFA}.

On input $\langle A, B \rangle$, where A and B are DFAs:

1. Construct $C = (A \cap \overline{B}) \cup (B \cap \overline{A})$. (DFAs are closed under union, intersection and complementation.)
2. Run T on input $\langle C \rangle$.
3. If T accepts, accept; if T rejects, reject.

Acceptance problem for CFG

Consider the language:

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

Lemma. A_{CFG} is a decidable language.

We use a fact: If G is in Chomsky normal form, any derivation of w has $2n - 1$ steps, where n is the length of w.

Proof. The TM S for A_{CFG} is: On input $\langle G, w \rangle$, where G is a CFG and w is a string:

1. Convert G to an equivalent Chomsky normal form.
2. List all derivations with $2n - 1$ steps, where n is the length of w.
3. If any of these derivations generate w, accept; otherwise, reject.

This is of course extremely inefficient, and will not be used in practice.

Emptiness Problem for CFG: Given a CFG G, is $L(G)$ empty?

Lemma. E_{CFG} is a decidable language.

Proof. The TM R for E_{CFG} is: On input $\langle G \rangle$, where $G = (V, \Sigma, R, S)$ is a CFG:

1. Mark all terminal symbols in G.
2. Repeat until no new variables get marked:

   Mark any variable A where G contains a rule $A \to U_1 \cdots U_k$ and each symbol $U_i \in \Sigma \cup V$ has already been marked.
3. If the start symbol is not marked, accept; otherwise, reject.

Equiv. Problem for CFG: Given two CFGs, are they equivalent?

Obvious attempt: Use the strategy for deciding EQ_{DFA}.

Unfortunately CFGs are neither closed under intersection nor complementation!

In fact EQ_{DFA} is *undecidable*.

Universal Turing Machines

Acceptance Problem for TM

$$A_{\mathrm{TM}} = \{ \langle M, w \rangle : M \text{ is a TM that accepts input } w \}$$

Theorem. A_{TM} is r.e.

Proof. We define a TM U that accepts A_{TM}: On input $\langle M, w \rangle$ where M is a TM and w is a string

1. *Simulate M on input w*.
2. If M ever enters its accept state, accept; if M ever enters its reject state, reject.

Note that U loops on $\langle M, w \rangle$ if M loops on w. This is why U is not a decider.

If the algorithm had some way to determine that M was not halting on w, it could reject.

Universal Turing Machines: Encoding Turing machines

A *universal* TM is a TM U that can simulate the actions of any Turing machine. I.e. for any TM M and any input x of M, U accepts (respectively rejects or loops on) $\langle M, x \rangle$, if and only if, M accepts (respectively rejects or loops on) x.

This is equivalent to an interpreter of C written in in C.

An encoding scheme over $\{ 0, 1 \}^*$: $\langle M, x \rangle$.

E.g. if the string begins with the prefix

$$0^n \, 1 \, 0^m \, 1 \, 0^k \, 1 \, 0^s \, 1 \, 0^t \, 1 \, 0^r \, 1 \, 0^u \, 1 \, 0^v$$

this might indicate that the machine has n states ($0, 1, \cdots, n-1$); it has m tapes symbols ($0, 1, \cdots, m-1$), of which the first k are input symbols; the start, accept and reject states are s, t and r respectively, and the endmarker and blank symbols are u and v respectively. The remainder of the string can consist of a sequence of substrings specifying the transitions in δ.

E.g. $0^p\,1\,0^a\,1\,0^q\,1\,0^b 1\,0$ might indicate that δ contains the transition

$$((p, a), (q, b, L)).$$

The exact details are unimportant.

Simulation. The tape of U is partitioned into 3 tracks:

- The top track holds the transition function δ_M of the input TM M.
- The middle track will be used to hold the simulated contents of M's tape.
- The bottom track will hold the current state of M, and the current position of M's tape head.

U simulates M on input x one step at a time, shuttling back and forth between the three tracks. In each step, it updates M's state and simulated tape contents and head position as dictated by δ_M. If ever M halts and accepts or halts and rejects, then U does the same.

Note: M's input alphabet and U's may be different.

There are languages that are not r.e.

There are only countably many TMs.

Observe that Σ^* is countable, for any alphabet Σ. Since $\langle M \rangle \in \Sigma^*$ for each TM M, we can enumerate the set of TMs by simply omitting those strings that are not encodings of TMs.

The set of languages over Σ, $\mathcal{P}(\Sigma^*)$, is uncountable.

Suppose there were an enumeration of $\mathcal{P}(\Sigma^*)$, namely, $L_1, L_2, L_3, \cdots$. Let $x_1, x_2, x_3, \cdots$ be an enumeration of Σ^*. Define a new language L by: for $i \geq 1$

$$x_i \in L \iff x_i \notin L_i$$

Now $L \in \mathcal{P}(\Sigma^*)$, but L is not equal to any of the L_is.

Thus $\mathcal{P}(\Sigma^*)$ is not in 1-1 correspondence with the set of Turing machines (there are more languages than TMs). It follows that there is some language that is not accepted by any TM.

A_{TM}: Acceptance Problem for TMs

Theorem. A_{TM} is undecidable.

Proof. Suppose, for a contradiction, TM H is a decider for A_{TM}. I.e.

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

We construct a new TM D with H as a subroutine. D: On input $\langle M \rangle$, where M is a TM

1. Run H on input $\langle M, \langle M \rangle \rangle$
2. Output the opposite of H. I.e. if H accepts, *reject*, and if H rejects, *accept*.

Thus

$$D(\langle M \rangle) = \begin{cases} \textit{reject} & \text{if } M \text{ accepts } \langle M \rangle \\ \textit{accept} & \text{if } M \text{ does not accept } \langle M \rangle \end{cases}$$

Now we run D on input $\langle D \rangle$. We have

$$D(\langle D \rangle) = \begin{cases} \textit{reject} & \text{if } D \text{ accepts } \langle D \rangle \\ \textit{accept} & \text{if } D \text{ does not accept } \langle D \rangle \end{cases}$$

I.e. D accepts $\langle D \rangle$ iff D rejects $\langle D \rangle$, which is a contradiction. Thus neither D nor H can exist. □

To summarize, A_{TM} is (r.e. but) undecidable.

A language that is not r.e.

We say that L is *co-r.e.* if $\overline{L}$ is r.e.

Theorem. A language is decidable iff it is both r.e. and co-r.e.

Proof. Suppose L and $\overline{L}$ are acceptable by M and M' respectively. Define a TM N: On input x

1. Write x on the input tapes of M and M'.
2. Run M and M' in parallel (e.g. by dovetailing the two computation).
3. If M accepts, *accept*; if M' accepts, *reject*.

N is a decider for L because for any input x, either $x \in L$ or $x \in \overline{L}$: if the former then N must accept because M must halt and accept at some point, if the latter, then N must reject because M' must halt and reject at some point.

Corollary. $\overline{A_{\mathrm{TM}}}$ is not r.e. □

The Halting Problem: "Given M and x, does M halt on x?"

$$\mathit{HALT}_{\mathrm{TM}} = \{ \langle M, w \rangle : M \text{ is a TM and } M \text{ halts on input } w \}$$

Theorem. The Halting Problem, $\mathit{HALT}_{\mathrm{TM}}$, is undecidable.

Proof. Suppose, for a contradiction, H is a decider for $\mathit{HALT}_{\mathrm{TM}}$. We use H to construct a TM S to decide A_{TM} as follows: On input $\langle M, w \rangle$ where M is a TM w is an input

1. Run TM H on input $\langle M, w \rangle$
2. If H rejects, *reject*.
3. If H accepts, run M on w until it halts.
4. If M has accepted, *accept*; if M has rejected, *reject*.

Since A_{TM} is undecidable, H cannot be a decider for $\mathit{HALT}_{\mathrm{TM}}$. □

The Emptiness Problem for TM is undecidable

$E_{\mathrm{TM}} = \{ \langle M \rangle : \ M$ is a TM and $L(M) = \emptyset \}$

Theorem. The Emptiness Problem for TM is undecidable.

Proof. Suppose, for a contradiction, E_{TM} is decidable by a TM E. We shall use E to construct a TM S that decides A_{TM}.

Definition of S: On input $\langle M, x \rangle$ where M is a TM and x an input

1. Construct a TM M_x defined as: On input y
   (a) If $y \neq x$ then *reject*.
   (b) Run M (input is x) and *accept* if M does.
2. Run E on input $\langle M_x \rangle$. If E accepts, *reject*, if E rejects, *accept*.

Now we have

$$M \text{ accepts } x \quad \Rightarrow \quad L(M_x) = \{ x \} \qquad \text{i.e. } S \text{ accepts } \langle M, x \rangle$$

$$M \text{ does not accept } x \quad \Rightarrow \quad L(M_x) = \emptyset \qquad \text{i.e. } S \text{ rejects } \langle M, x \rangle$$

Thus S decides A_{TM}, which is a contradiction. $\square$

The Equivalence Problem for TM is undecidable

$EQ_{\mathrm{TM}} = \{ \langle M_1, M_2 \rangle : \ M_1$ and M_2 are TMs and $L(M_1) = L(M_2) \}$

Theorem. The Equivalence Problem for TM is undecidable.

Proof. Suppose, for a contradiction, EQ_{TM} is decidable by Q. We define a TM that decides E_{TM} by: On input M

1. Run Q on input $\langle M, M_1 \rangle$ where M_1 is a TM that accepts nothing i.e. $L(M_1) = \emptyset$.
2. If Q accepts, *accept*; if Q rejects, *reject*.

This gives a contradiction. □

PCP: Post Correspondence Problem

A puzzle: Given a collection of dominos such as e.g.

$$P = \left\{ \left[\frac{b}{ca}\right], \left[\frac{a}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{abc}{c}\right] \right\}$$

find a list of dominos from P (repetitions permitted) so that the string we get by reading off the symbols on the top is the same as the sting of symbols on the bottom. This list is called a *match*.

E.g. the following is a match

$$\left\{ \left[\frac{a}{ab}\right], \left[\frac{b}{ca}\right], \left[\frac{ca}{a}\right], \left[\frac{a}{ab}\right], \left[\frac{abc}{c}\right], \right\}$$

(Reading off the top string we get $abcaaabc$.)

A formal statement of the PCP

Instance: A collection P of *dominos*:

$$P = \left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \cdots, \left[\frac{t_k}{b_k}\right] \right\}$$

where $k \geq 1$, and each t_i and b_j are strings over Σ.

Question: Is there a *match* for P? I.e. a non-empty sequence $i_1, \cdots, i_l$ where each $1 \leq i_j \leq k$ and

$$t_{i_1}\, t_{i_2}\, \cdots\, t_{i_l} \;=\; b_{i_1}\, b_{i_2}\, \cdots\, b_{i_l}$$

Theorem. PCP is undecidable.

Examples

1. Let $\Sigma = \{ 0, 1 \}$, and

$$P = \left\{ \left[\frac{11}{111}\right], \left[\frac{111}{11}\right], \left[\frac{100}{001}\right] \right\}.$$

Here is a match for P:

$$P = \left\{ \left[\frac{11}{111}\right], \left[\frac{100}{001}\right], \left[\frac{111}{11}\right] \right\}.$$

2. For some collection of dominos, find a match may not be possible. E.g.

$$\left\{ \left[\frac{abc}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{acc}{bc}\right] \right\}$$

(because every top string is longer than the bottom string)

Undecidability of PCP: outline proof

We modify *PCP* to

$$\textit{MPCP} \quad = \quad \{P \text{ is an instance of PCP with a match that starts with the first domino}\}$$

1. Suppose, for a contradiction, there is a TM R that decides the PCP. We use R to construct a TM S that decide A_{TM}. Let

$$M \;=\; (Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}})$$

Take an input w for M. We construct an instance P' of the *MPCP* (over the alphabet $\Gamma \cup \{\,\#\,\}$) that simulates M on w i.e.

$$P' \in \textit{MPCP} \quad \Longleftrightarrow \quad \langle\, M, w \,\rangle \in A_{\mathrm{TM}}.$$

Idea. Writing a configuration (u_i, q_i, v_i) as $\#u_i\ q_i\ v_i$, a match of P' gives a string; the first part

$$\#q_0 \vdash w\#u_1\ q_1\ v_1\#u_2\ q_2\ v_2\#\cdots\#u_i\ q_i\ v_i\#\cdots\#u_n\ q_{\mathrm{acc}}\ v_n$$

corresponds exactly to an accepting configuration; the rest of the string

$$\#\cdots\#\cdots\cdots\#q_{\mathrm{acc}}\#\#$$

consists of progressively shorter segments of the form $\#\cdots$ that finally ends in $\#\#$.

2. We convert P' to P, an instance of *PCP* which still simulates M on w.

Definition of P'

Part 1. Put $\left[\frac{\#}{\#q_0 \vdash w_1 \cdots w_n \#}\right]$ **into** P' **as the first domino** $\left[\frac{t_1}{b_1}\right]$

Part 2. For each $a, b \in \Gamma$, $q, r \in Q$, **if** $\delta(q, a) = (r, b, R)$, **put** $\left[\frac{qa}{br}\right]$ **into** P'

Part 3. For each $a, b \in \Gamma$, $q, r \in Q$, **if** $\delta(q, a) = (r, b, L)$, **put** $\left[\frac{cqa}{rcb}\right]$ **into** P'

Part 4. For every $a \in \Gamma$, **put** $\left[\frac{a}{a}\right]$ **into** P'.

Part 5. Put $\left[\frac{\#}{\#}\right]$ **and** $\left[\frac{\#}{\sqcup \#}\right]$ **into** P'.

Part 6. For every $a \in \Gamma$, **put** $\left[\frac{a q_{\text{acc}}}{q_{\text{acc}}}\right]$ **and** $\left[\frac{q_{\text{acc}} a}{q_{\text{acc}}}\right]$ **into** P'.

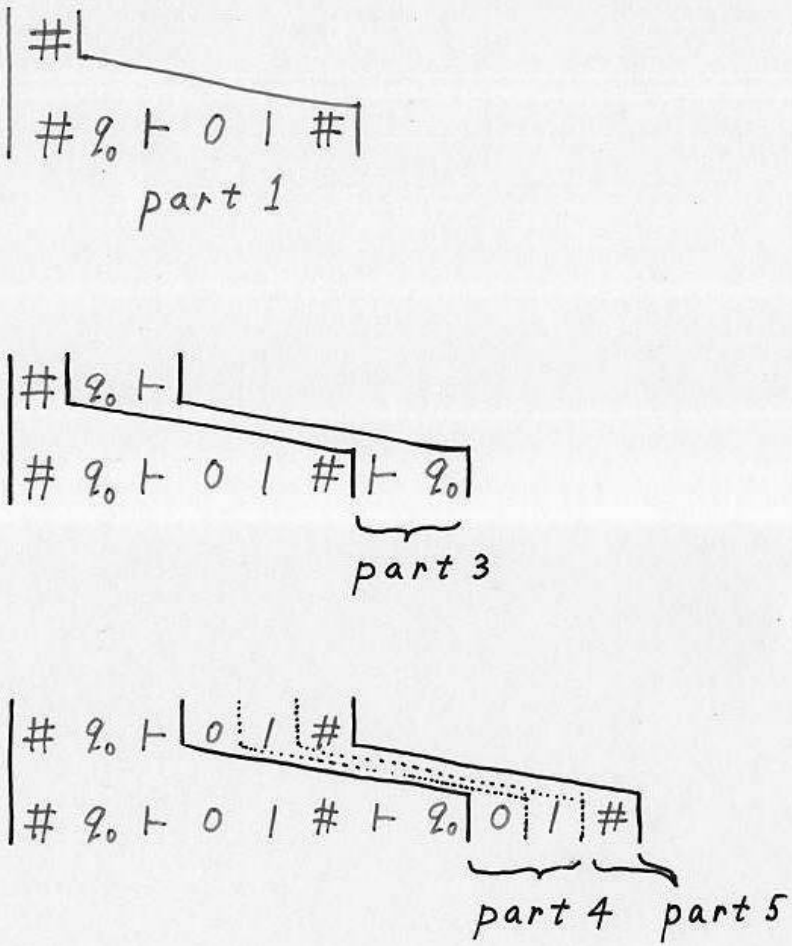
Part 7. Add $\left[\frac{q_{\text{acc}} \# \#}{\#}\right]$ **to** P'.

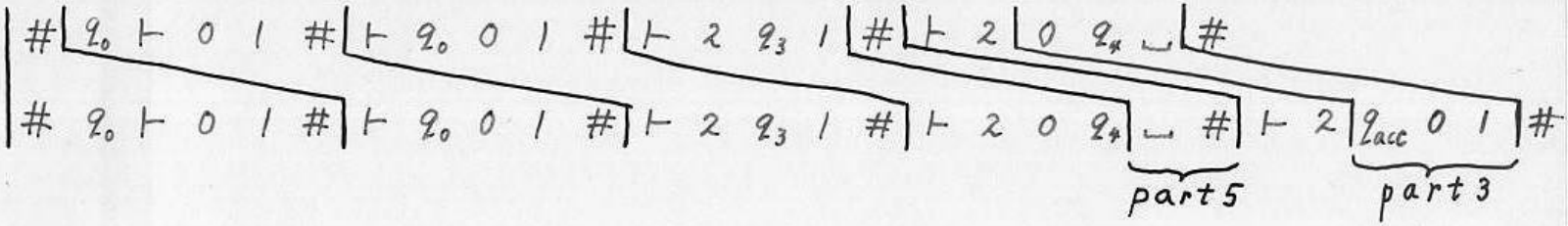
E.g. $\Sigma = \{0, 1, 2\}$ Input: 0 1

$$\delta: \begin{cases} (q_0, \vdash) \longmapsto (q_0, \vdash, R) \\ (q_0, 0) \longmapsto (q_3, 2, R) \\ (q_3, 1) \longmapsto (q_4, 0, R) \\ (q_4, \sqcup) \longmapsto (q_{acc}, 1, L) \end{cases}$$

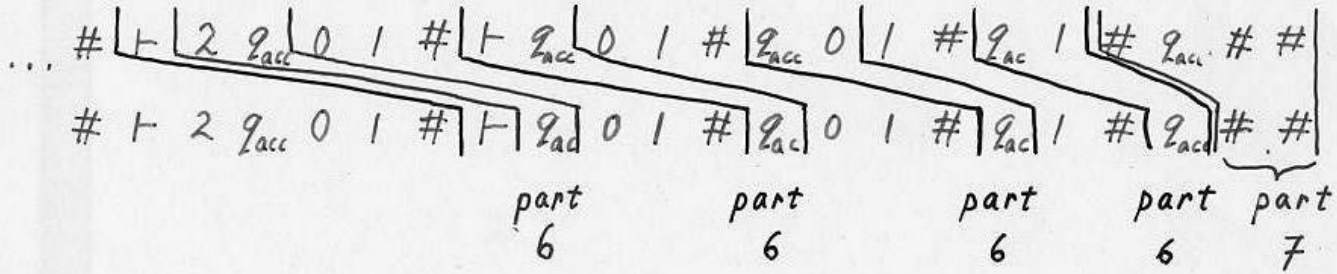
An accepting computation

$$\begin{aligned} (\varepsilon, q_0, \vdash 0\,1) &\to (\vdash, q_0, 0\,1) \\ &\to (\vdash 2, q_3, 1) \\ &\to (\vdash 2\,0, q_4, \sqcup) \\ &\to (\vdash 2, q_{acc}, 0\,1) \end{aligned}$$


#
q0 ⊢ 0 1
part 1
q0 ⊢
q0 ⊢ 0 1 # ⊢ q0
part 3
q0 ⊢ 0 1
q0 ⊢ 0 1 # ⊢ q0 0 1
part 4
part 5



Tidying up



Converting a MPCP instance to a PCP instance

Finally we show how to convert P' to an instance of PCP which still simulates M on w. The idea is to build the requirement of starting with the first domino directly into the problem.

Notation. Take a string $u = u_1 \cdots u_n$. Define

$$\begin{aligned} \star u &= *\,u_1 * u_2 * \cdots * u_n \\ u\star &= u_1 * u_2 * \cdots * u_n * \\ \star u \star &= *\,u_1 * u_2 * \cdots * u_n * \end{aligned}$$

Suppose P' is the MPCP instance (over alphabet Θ)

$$\left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \left[\frac{t_3}{b_3}\right], \ldots, \left[\frac{t_k}{b_k}\right] \right\}$$

Now define P to be the PCP instance (over alphabet $\Theta \cup \{ *, \diamond \}$)

$$\left\{ \left[\frac{\star\, t_1}{\star\, b_1\, \star}\right], \left[\frac{\star\, t_2}{b_2\, \star}\right], \left[\frac{\star\, t_3}{b_3\, \star}\right], \ldots, \left[\frac{\star\, t_k}{b_k\, \star}\right], \left[\frac{\star \diamond}{\diamond}\right] \right\}$$

Note that the only domino that can start a match is the first one, because it is the only one where both the top and bottom start with the same symbol, namly, $*$.

The original symbols in Θ now occur in even positions of the match.

The domino $\left[\frac{\star \diamond}{\diamond}\right]$ is to allow the top to add the extra $*$ at the end of the match.