Introduction to Theoretical Computer Science

Motivation

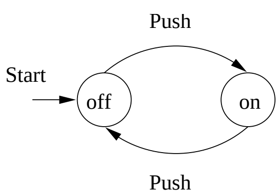
- Automata = abstract computing devices
- Turing studied Turing Machines (= computers) before there were any real computers
- We will also look at simpler devices than Turing machines (Finite State Automata, Pushdown Automata, ...), and specification means, such as grammars and regular expressions.
- NP-hardness = what cannot be efficiently computed.
- Undecidability = what cannot be computed at all.

Finite Automata

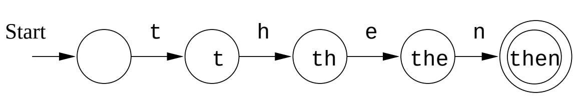
Finite Automata are used as a model for

- Software for designing digital cicuits
- Lexical analyzer of a compiler
- Searching for keywords in a file or on the web.
- Software for verifying finite state systems, such as communication protocols.

- Example: Finite Automaton modelling an on/off switch



- Example: Finite Automaton recognizing the string `then`



Structural Representations

These are alternative ways of specifying a machine

Grammars: A rule like $E \Rightarrow E + E$ specifies an arithmetic expression

- $Lineup \Rightarrow Person.Lineup$
  $Lineup \Rightarrow Person$

says that a lineup is a single person, or a person in front of a lineup.

Regular Expressions: Denote structure of data, e.g.

`'[A-Z][a-z]*[][A-Z][A-Z]'`

matches `Ithaca NY`

does not match `Palo Alto CA`

Question: What expression would match
`Palo Alto CA`

Central Concepts

Alphabet: Finite, nonempty set of symbols

Example: $\Sigma = \{0, 1\}$ binary alphabet

Example: $\Sigma = \{a, b, c, \ldots, z\}$ the set of all lower case letters

Example: The set of all ASCII characters

Strings: Finite sequence of symbols from an alphabet Σ, e.g. 0011001

Empty String: The string with zero occurrences of symbols from Σ

- The empty string is denoted ϵ

Length of String: Number of positions for symbols in the string.

$|w|$ denotes the length of string w

$|0110| = 4, |\epsilon| = 0$

Powers of an Alphabet: Σ^k = the set of strings of length k with symbols from Σ

Example: $\Sigma = \{0, 1\}$

$\Sigma^1 = \{0, 1\}$

$\Sigma^2 = \{00, 01, 10, 11\}$

$\Sigma^0 = \{\epsilon\}$

Question: How many strings are there in Σ^3

The set of all strings over Σ is denoted Σ^*

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$

Also:

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots$

$\Sigma^* = \Sigma^+ \cup \{\epsilon\}$

Concatenation: If x and y are strings, then xy is the string obtained by placing a copy of y immediately after a copy of x

$x = a_1 a_2 \ldots a_i$
$y = b_1 b_2 \ldots b_j$

$xy = a_1 a_2 \ldots a_i b_1 b_2 \ldots b_j$

Example: $x = 01101, y = 110, xy = 01101110$

Note: For any string x

$$x\epsilon = \epsilon x = x$$

Languages:

If Σ is an alphabet, and $L \subseteq \Sigma^*$
then L is a language

Examples of languages:

- The set of legal English words
- The set of legal C programs
- The set of strings consisting of n 0's followed by n 1's

$$\{\epsilon, 01, 0011, 000111, \ldots\}$$

- The set of strings with equal number of 0's and 1's
  $$\{\epsilon, 01, 10, 0011, 0101, 1001, \ldots\}$$

- L_P = the set of binary numbers whose value is prime
  $$\{10, 11, 101, 111, 1011, \ldots\}$$

- The empty language $\emptyset$

- The language $\{\epsilon\}$ consisting of the empty string

Note: $\emptyset \neq \{\epsilon\}$

Note2: The underlying alphabet Σ is always finite

Problem: Is a given string w a member of a language L?

Example: Is a binary number prime = is it a meber in L_P

Is $11101 \in L_P$? What computational resources are needed to answer the question.

Usually we think of problems not as a yes/no decision, but as something that transforms an input into an output.

Example: Parse a C-program = check if the program is correct, and if it is, produce a parse tree.

Let L_X be the set of all valid programs in prog lang X. If we can show that determining membership in L_X is hard, then parsing programs written in X cannot be easier.

Question: Why?

Finite Automata Informally

Protocol for e-commerce using e-money

Allowed events:

1. The customer can *pay* the store (=send the money-file to the store)

2. The customer can *cancel* the money (like putting a stop on a check)

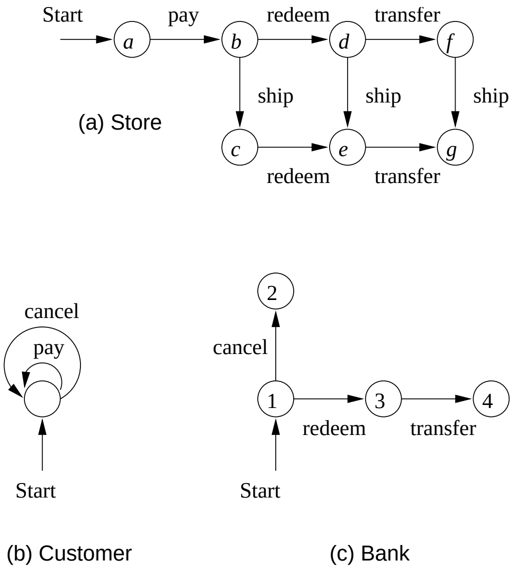
3. The store can *ship* the goods to the customer

4. The store can *redeem* the money (=cash the check)

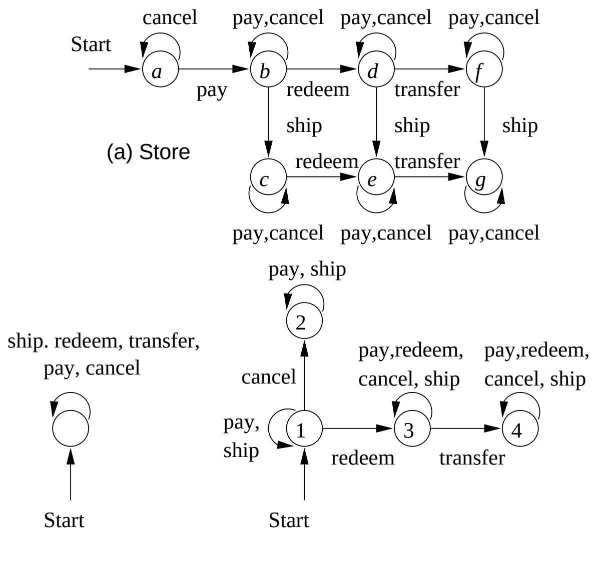
5. The bank can *transfer* the money to the store

e-commerce

The protocol for each participant:



Completed protocols:

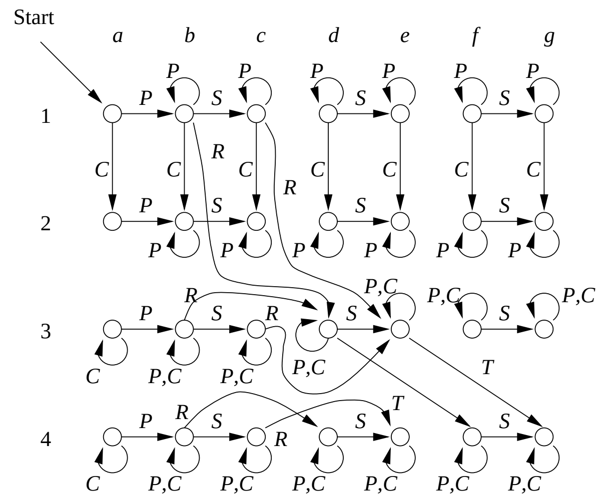


(a) Store

(b) Customer

(c) Bank

The entire system as an Automaton:



Deterministic Finite Automata

A DFA is a quintuple

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

- Q is a finite set of *states*
- Σ is a *finite alphabet* (=input symbols)
- δ is a *transition function* $(q, a) \mapsto p$
- $q_0 \in Q$ is the *start state*
- $F \subseteq Q$ is a set of *final states*

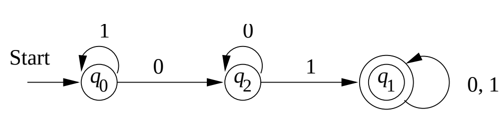
Example: An automaton A that accepts

$$L = \{x01y : x, y \in \{0,1\}^*\}$$

The automaton $A = (\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_1\})$ as a *transition table*:

δ	0	1
$\rightarrow q_0$	q_2	q_0
$\star q_1$	q_1	q_1
q_2	q_2	q_1

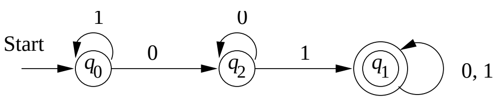
The automaton A as a *transition diagram*:



An FA *accepts* a string $w = a_1a_2 \dots a_n$ if there is a path in the transition diagram that

1. Begins at a start state
2. Ends at an accepting state
3. Has sequence of labels $a_1a_2 \dots a_n$

Example: The FA



accepts e.g. the string 1100101

• The transition function δ can be extended to $\hat{\delta}$ that operates on states and strings (as opposed to states and symbols)

Basis: $\hat{\delta}(q, \epsilon) = q$

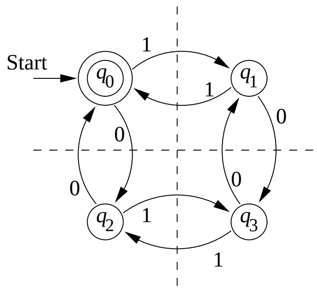
Induction: $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$

• Now, fomally, the *language accepted by* A is

$$L(A) = \{w : \hat{\delta}(q_0, w) \in F\}$$

• The languages accepted by FA:s are called *regular languages*

Example: DFA accepting all and only strings with an even number of 0's and an even number of 1's

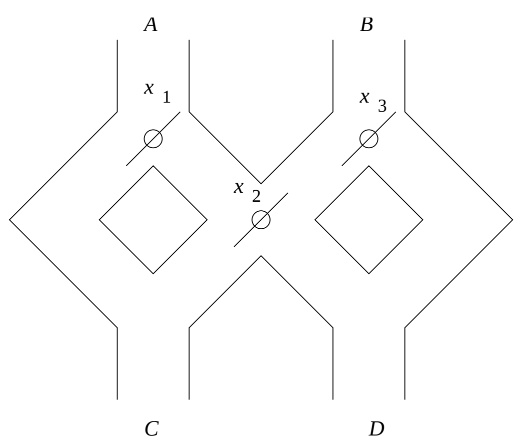


Tabular representation of the Automaton

δ	0	1
$\star \to q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Example

Marble-rolling toy from p. 53 of textbook



A state is represented as sequence of three bits followed by r or a (previous input *rejected* or *accepted*)

For instance, $010a$, means
left, right, left, accepted

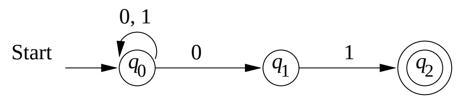
Tabular representation of DFA for the toy

	A	B
$\rightarrow 000r$	$100r$	$011r$
$\star 000a$	$100r$	$011r$
$\star 001a$	$101r$	$000a$
$010r$	$110r$	$001a$
$\star 010a$	$110r$	$001a$
$011r$	$111r$	$010a$
$100r$	$010r$	$111r$
$\star 100a$	$010r$	$111r$
$101r$	$011r$	$100a$
$\star 101a$	$011r$	$100a$
$110r$	$000a$	$101a$
$\star 110a$	$000a$	$101a$
$111r$	$001a$	$110a$

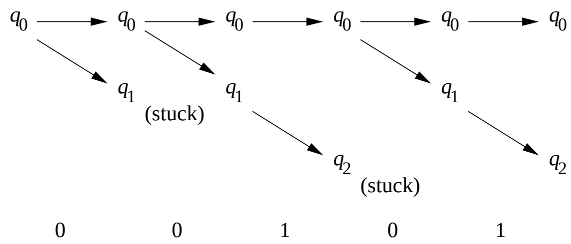
Nondeterministic Finite Automata

A NFA can be in several states at once, or, viewded another way, it can "guess" which state to go to next

Example: An automaton that accepts all and only strings ending in 01.



Here is what happens when the NFA processes the input 00101



Formally, a NFA is a quintuple

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

- Q is a finite set of states
- Σ is a finite alphabet
- δ is a transition function from $Q \times \Sigma$ to the *powerset* of Q
- $q_0 \in Q$ is the *start state*
- $F \subseteq Q$ is a set of *final states*

Example: The NFA from the previous slide is

$$(\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$$

where δ is the transition function

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

Extended transition function $\widehat{\delta}$.

Basis: $\widehat{\delta}(q,\epsilon) = \{q\}$

Induction:

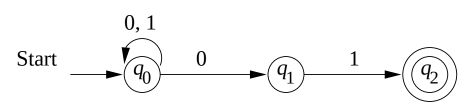
$$\widehat{\delta}(q,xa) = \bigcup_{p \in \widehat{\delta}(q,x)} \delta(p,a)$$

Example: Let's compute $\widehat{\delta}(q_0, 00101)$ on the blackboard

- Now, fomally, the *language accepted by* A is

$$L(A) = \{w : \widehat{\delta}(q_0,w) \cap F \neq \emptyset\}$$

Let's prove formally that the NFA



accepts the language $\{x01 : x \in \Sigma^*\}$. We'll do a mutual induction on the three statements below

0. $w \in \Sigma^* \Rightarrow q_0 \in \widehat{\delta}(q_0,w)$

1. $q_1 \in \widehat{\delta}(q_0,w) \Leftrightarrow w = x0$

2. $q_2 \in \widehat{\delta}(q_0,w) \Leftrightarrow w = x01$

Basis: If $|w| = 0$ then $w = \epsilon$. Then statement (0) follows from def. For (1) and (2) both sides are false for ϵ

Induction: Assume $w = xa$, where $a \in \{0,1\}$, $|x| = n$ and statements (0)–(2) hold for x. We will show on the blackboard in class that the statements hold for xa.

Equivalence of DFA and NFA

- NFA's are usually easier to "program" in.

- Surprisingly, for any NFA N there is a DFA D, such that $L(D) = L(N)$, and vice versa.

- This involves the *subset construction*, an important example how an automaton B can be generically constructed from another automaton A.

- Given an NFA

$$N = (Q_N, \Sigma, \delta_N, q_0, F_N)$$

we will construct a DFA

$$D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$$

such that

$$L(D) = L(N)$$

.

The details of the subset construction:

- $Q_D = \{S : S \subseteq Q_N\}$.

Note: $|Q_D| = 2^{|Q_N|}$, although most states in Q_D are likely to be garbage.

- $F_D = \{S \subseteq Q_N : S \cap F_N \neq \emptyset\}$

- For every $S \subseteq Q_N$ and $a \in \Sigma$,

$$\delta_D(S, a) = \bigcup_{p \in S} \delta_N(p, a)$$

Let's construct δ_D from the NFA on slide 26

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

Note: The states of D correspond to subsets of states of N, but we could have denoted the states of D by, say, $A - F$ just as well.

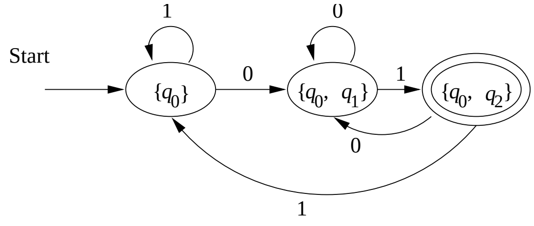
	0	1
A	A	A
$\rightarrow B$	E	B
C	A	D
$\star D$	A	A
E	E	F
$\star F$	E	B
$\star G$	A	D
$\star H$	E	F

We can often avoid the exponential blow-up by constructing the transition table for D only for accessible states S as follows:

Basis: $S = \{q_0\}$ is accessible in D

Induction: If state S is accessible, so are the states in $\bigcup_{a \in \Sigma} \delta_D(S, a)$.

Example: The "subset" DFA with accessible states only.



Theorem 2.11: Let D be the "subset" DFA of an NFA N. Then $L(D) = L(N)$.

Proof: First we show on an induction on $|w|$ that

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

Basis: $w = \epsilon$. The claim follows from def.

Induction:

$$\begin{aligned}
\widehat{\delta}_D(\{q_0\}, xa) &\stackrel{\text{def}}{=} \delta_D(\widehat{\delta}_D(\{q_0\}, x), a) \\
&\stackrel{\text{i.h.}}{=} \delta_D(\widehat{\delta}_N(q_0, x), a) \\
&\stackrel{\text{cst}}{=} \bigcup_{p \in \widehat{\delta}_N(q_0, x)} \delta_N(p, a) \\
&\stackrel{\text{def}}{=} \widehat{\delta}_N(q_0, xa)
\end{aligned}$$

Now (**why?**) it follows that $L(D) = L(N)$.

Theorem 2.12: A language L is accepted by some DFA if and only if L is accepted by some NFA.

Proof: The "if" part is Theorem 2.11.

For the "only if" part we note that any DFA can be converted to an equivalent NFA by modifying the δ_D to δ_N by the rule

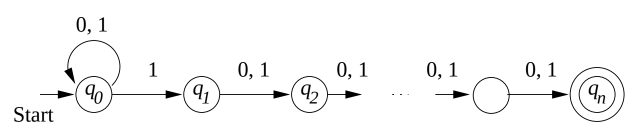
- If $\delta_D(q, a) = p$, then $\delta_N(q, a) = \{p\}$.

By induction on $|w|$ it will be shown in the tutorial that if $\widehat{\delta}_D(q_0, w) = p$, then $\widehat{\delta}_N(q_0, w) = \{p\}$.

The claim of the theorem follows.

Exponential Blow-Up

There is an NFA N with $n+1$ states that has no equivalent DFA with fewer than 2^n states



$$L(N) = \{x1c_2c_3 \cdots c_n : x \in \{0,1\}^*, c_i \in \{0,1\}\}$$

Suppose an equivalent DFA D with fewer than 2^n states exists.

D must remember the last n symbols it has read. There are 2^n bitsequences $a_1a_2 \ldots a_n$. Since D has fewer that 2^n states

$\exists\, q,\ a_1a_2 \ldots a_n,\ b_1b_2 \ldots b_n :$

$$a_1a_2 \ldots a_n \neq b_1b_2 \ldots b_n$$

$$\widehat{\delta}_D(q_0, a_1a_2 \ldots a_n) = \widehat{\delta}_D(q_0, b_1b_2 \ldots b_n) = q$$

Since $a_1a_2\ldots a_n \neq b_1b_2\ldots b_n$ they must differ in at least one position.

Case 1:

$1a_2\ldots a_n$
$0b_2\ldots b_n$

Then q has to be both an accepting and a nonaccepting state.

Case 2:

$a_1\ldots a_{i-1}1a_{i+1}\ldots a_n$
$b_1\ldots b_{i-1}0b_{i+1}\ldots b_n$

Now $\hat{\delta}_D(q_0, a_1\ldots a_{i-1}1a_{i+1}\ldots a_n0^{i-1}) =$
$\hat{\delta}_D(q_0, b_1\ldots b_{i-1}0b_{i+1}\ldots b_n0^{i-1})$

and $\hat{\delta}_D(q_0, a_1\cdots a_{i-1}1a_{i+1}\cdots a_n0^{i-1}) \in F_D$

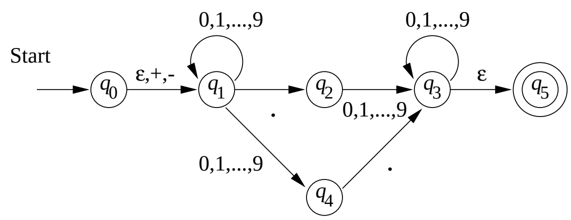
$\hat{\delta}_D(q_0, b_1\cdots b_{i-1}0b_{i+1}\cdots b_n0^{i-1}) \notin F_D$

FA's with Epsilon-Transitions

An ϵ-NFA accepting decimal numbers consisting of:

1. An optional + or - sign
2. A string of digits
3. a decimal point
4. another string of digits

One of the strings (2) are (4) are optional



An ϵ-NFA is a quintuple $(Q, \Sigma, \delta, q_0, F)$ where δ is a function from $Q \times \Sigma \cup \{\epsilon\}$ to the powerset of Q.

Example: The ϵ-NFA from the previous slide

$$E = (\{q_0, q_1, \ldots, q_5\}, \{., +, -, 0, 1, \ldots, 9\}\ \delta, q_0, \{q_5\})$$

where the transition table for δ is

	ϵ	+,-	.	$0,\ldots,9$
$\rightarrow q_0$	$\{q_1\}$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_3	$\{q_5\}$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$	$\{q_3\}$	$\emptyset$
$\star q_5$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

ECLOSE

We close a state by adding all states reachable by a sequence $\epsilon\epsilon\cdots\epsilon$

Inductive definition of ECLOSE(q)

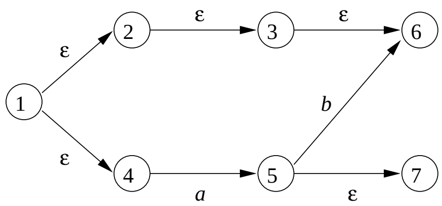
Basis:

$q \in$ ECLOSE(q)

Induction:

$p \in$ ECLOSE(q) and $r \in \delta(p, \epsilon)$ $\Rightarrow$
$r \in$ ECLOSE(q)

Example of ϵ-closure



For instance,

$$\text{ECLOSE}(1) = \{1, 2, 3, 4, 6\}$$

- Inductive definition of $\widehat{\delta}$ for ϵ-NFA's

Basis:

$$\widehat{\delta}(q, \epsilon) = \text{ECLOSE}(q)$$

Induction:

$$\widehat{\delta}(q, xa) = \bigcup_{p \in \delta(\widehat{\delta}(q,x),a)} \text{ECLOSE}(p)$$

Let's compute on the blackboard in class $\widehat{\delta}(q_0, 5.6)$ for the NFA on slide 38

Given an ϵ-NFA

$$E = (Q_E, \Sigma, \delta_E, q_0, F_E)$$

we will construct a DFA

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

such that

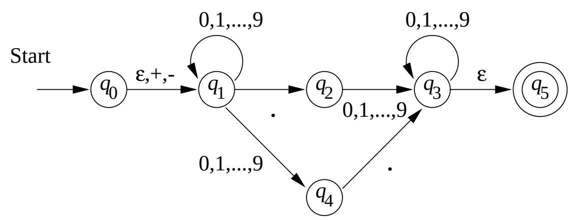
$$L(D) = L(E)$$

Details of the construction:

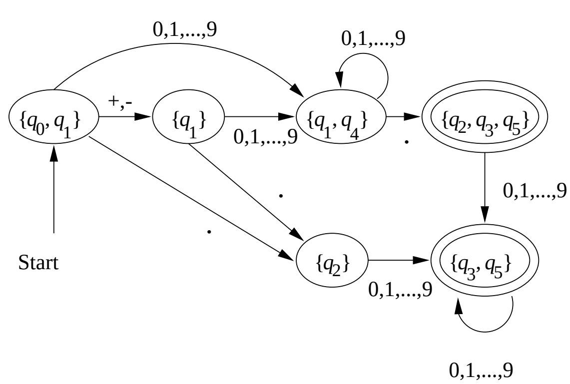
- $Q_D = \{S : S \subseteq Q_E \text{ and } S = \text{ECLOSE}(S)\}$
- $q_D = \text{ECLOSE}(q_0)$
- $F_D = \{S : S \in Q_D \text{ and } S \cap F_E \neq \emptyset\}$
- $\delta_D(S, a) =$

  $\bigcup\{\text{ECLOSE}(p) : p \in \delta(t, a) \text{ for some } t \in S\}$

Example: ϵ-NFA E



DFA D corresponding to E



Theorem 2.22: A language L is accepted by some ϵ-NFA E if and only if L is accepted by some DFA.

Proof: We use D constructed as above and show by induction that $\widehat{\delta}_D(q_0, w) = \widehat{\delta}_E(q_D, w)$

Basis: $\widehat{\delta}_E(q_0, \epsilon) = \text{ECLOSE}(q_0) = q_D = \widehat{\delta}(q_D, \epsilon)$

Induction:

$$\begin{aligned}
\widehat{\delta}_E(q_0, xa) &= \bigcup_{p \in \delta_E(\widehat{\delta}_E(q_0,x),a)} \text{ECLOSE}(p) \\
&= \bigcup_{p \in \delta_D(\widehat{\delta}_D(q_D,x),a)} \text{ECLOSE}(p) \\
&= \bigcup_{p \in \widehat{\delta}_D(q_D,xa)} \text{ECLOSE}(p) \\
&= \widehat{\delta}_D(q_D, xa)
\end{aligned}$$

Regular expressions

A FA (NFA or DFA) is a "blueprint" for contructing a machine recognizing a regular language.

A *regular expression* is a "user-friendly," declarative way of describing a regular language.

Example: $\mathbf{01^* + 10^*}$

Regular expressions are used in e.g.

1. UNIX `grep` command

2. UNIX Lex (Lexical analyzer generator) and Flex (Fast Lex) tools.

Operations on languages

Union:

$L \cup M = \{w : w \in L \text{ or } w \in M\}$

Concatenation:

$L.M = \{w : w = xy, x \in L, y \in M\}$

Powers:

$L^0 = \{\epsilon\},\ L^1 = L,\ L^{k+1} = L.L^k$

Kleene Closure:

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

Question: What are $\emptyset^0$, $\emptyset^i$, and $\emptyset^*$

Building regex's

Inductive definition of regex's:

Basis: $\boldsymbol{\epsilon}$ is a regex and $\boldsymbol{\emptyset}$ is a regex.
$L(\boldsymbol{\epsilon}) = \{\epsilon\}$, and $L(\boldsymbol{\emptyset}) = \emptyset$.

If $a \in \Sigma$, then $\boldsymbol{a}$ is a regex.
$L(\boldsymbol{a}) = \{a\}$.

Induction:

If E is a regex's, then (E) is a regex.
$L((E)) = L(E)$.

If E and F are regex's, then $E + F$ is a regex.
$L(E + F) = L(E) \cup L(F)$.

If E and F are regex's, then $E.F$ is a regex.
$L(E.F) = L(E).L(F)$.

If E is a regex's, then $E^\star$ is a regex.
$L(E^\star) = (L(E))^*$.

Example: Regex for

$L = \{w \in \{0,1\}^* : \text{ 0 and 1 alternate in } w\}$

$$\mathbf{(01)^* + (10)^* + 0(10)^* + 1(01)^*}$$

or, equivalently,

$$(\boldsymbol{\epsilon} + \mathbf{1})(\mathbf{01})^*(\boldsymbol{\epsilon} + \mathbf{0})$$

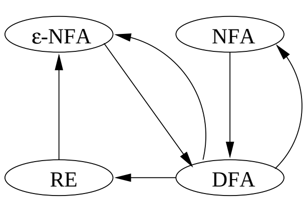
Order of precedence for operators:

1. Star
2. Dot
3. Plus

Example: $\mathbf{01^* + 1}$ is grouped $(\mathbf{0}(\mathbf{1})^*) + \mathbf{1}$

Equivalence of FA's and regex's

We have already shown that DFA's, NFA's, and ϵ-NFA's all are equivalent.



To show FA's equivalent to regex's we need to establish that

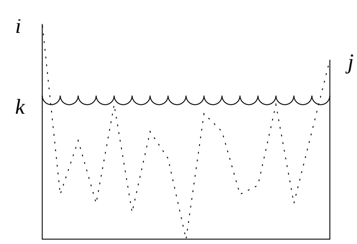
1. For every DFA A we can find (construct, in this case) a regex R, s.t. $L(R) = L(A)$.

2. For every regex R there is a ϵ-NFA A, s.t. $L(A) = L(R)$.

Theorem 3.4: For every DFA $A = (Q, \Sigma, \delta, q_0, F)$ there is a regex R, s.t. $L(R) = L(A)$.

Proof: Let the states of A be $\{1, 2, \ldots, n\}$, with 1 being the start state.

• Let $R_{ij}^{(k)}$ be a regex describing the set of labels of all paths in A from state i to state j going through intermediate states $\{1, \ldots, k\}$ only.



$R_{ij}^{(k)}$ will be defined inductively. Note that

$$L\left(\bigoplus_{j\in F} R_{1j}{}^{(n)}\right) = L(A)$$

Basis: $k=0$, i.e. no intermediate states.

- *Case 1:* $i \neq j$

$$R_{ij}^{(0)} = \bigoplus_{\{a\in\Sigma : \delta(i,a)=j\}} \boldsymbol{a}$$

- *Case 2:* $i = j$

$$R_{ii}^{(0)} = \left(\bigoplus_{\{a\in\Sigma : \delta(i,a)=i\}} \boldsymbol{a}\right) + \boldsymbol{\epsilon}$$

Induction:

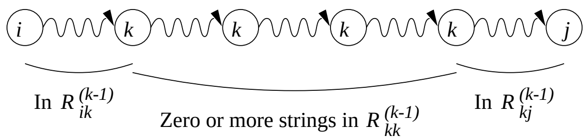
$$R_{ij}^{(k)}$$

$$=$$

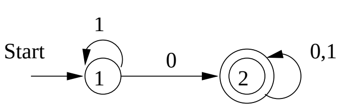
$$R_{ij}^{(k-1)}$$

$$+$$

$$R_{ik}^{(k-1)}\left(R_{kk}^{(k-1)}\right)^* R_{kj}^{(k-1)}$$



Example: Let's find R for A, where
$L(A) = \{x0y : x \in \{1\}^* \text{ and } y \in \{0,1\}^*\}$



$R_{11}^{(0)}$	$\boldsymbol{\epsilon}+\mathbf{1}$
$R_{12}^{(0)}$	$\mathbf{0}$
$R_{21}^{(0)}$	$\emptyset$
$R_{22}^{(0)}$	$\boldsymbol{\epsilon}+\mathbf{0}+\mathbf{1}$

We will need the following *simplification rules:*

- $(\boldsymbol{\epsilon} + R)^* = R^*$
- $R + RS^* = RS^*$
- $\emptyset R = R\emptyset = \emptyset$ (Annihilation)
- $\emptyset + R = R + \emptyset = R$ (Identity)

$R_{11}^{(0)}$	$\epsilon+1$
$R_{12}^{(0)}$	$\mathbf{0}$
$R_{21}^{(0)}$	$\emptyset$
$R_{22}^{(0)}$	$\epsilon+0+1$

$$R_{ij}^{(1)} = R_{ij}^{(0)} + R_{i1}^{(0)}\left(R_{11}^{(0)}\right)^* R_{1j}^{(0)}$$

	By direct substitution	Simplified
$R_{11}^{(1)}$	$\epsilon+1+(\epsilon+1)(\epsilon+1)^*(\epsilon+1)$	1^*
$R_{12}^{(1)}$	$0+(\epsilon+1)(\epsilon+1)^*0$	1^*0
$R_{21}^{(1)}$	$\emptyset+\emptyset(\epsilon+1)^*(\epsilon+1)$	$\emptyset$
$R_{22}^{(1)}$	$\epsilon+0+1+\emptyset(\epsilon+1)^*0$	$\epsilon+0+1$

	Simplified
$R_{11}^{(1)}$	1^*
$R_{12}^{(1)}$	1^*0
$R_{21}^{(1)}$	$\emptyset$
$R_{22}^{(1)}$	$\epsilon+0+1$

$$R_{ij}^{(2)} = R_{ij}^{(1)} + R_{i2}^{(1)}\left(R_{22}^{(1)}\right)^* R_{2j}^{(1)}$$

	By direct substitution
$R_{11}^{(2)}$	$1^*+1^*0(\epsilon+0+1)^*\emptyset$
$R_{12}^{(2)}$	$1^*0+1^*0(\epsilon+0+1)^*(\epsilon+0+1)$
$R_{21}^{(2)}$	$\emptyset+(\epsilon+0+1)(\epsilon+0+1)^*\emptyset$
$R_{22}^{(2)}$	$\epsilon+0+1+(\epsilon+0+1)(\epsilon+0+1)^*(\epsilon+0+1)$

	By direct substitution
$R_{11}^{(2)}$	$1^*+1^*0(\epsilon+0+1)^*\emptyset$
$R_{12}^{(2)}$	$1^*0+1^*0(\epsilon+0+1)^*(\epsilon+0+1)$
$R_{21}^{(2)}$	$\emptyset+(\epsilon+0+1)(\epsilon+0+1)^*\emptyset$
$R_{22}^{(2)}$	$\epsilon+0+1+(\epsilon+0+1)(\epsilon+0+1)^*(\epsilon+0+1)$

	Simplified
$R_{11}^{(2)}$	1^*
$R_{12}^{(2)}$	$1^*0(0+1)^*$
$R_{21}^{(2)}$	$\emptyset$
$R_{22}^{(2)}$	$(0+1)^*$

The final regex for A is

$$R_{12}^{(2)} = 1^*0(0+1)^*$$

Observations

There are n^3 expressions $R_{ij}^{(k)}$

Each inductive step grows the expression 4-fold

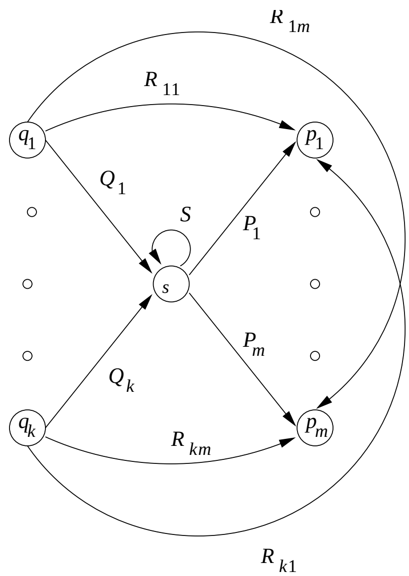
$R_{ij}^{(n)}$ could have size 4^n

For all $\{i,j\} \subseteq \{1,\ldots,n\}$, $R_{ij}^{(k)}$ uses $R_{kk}^{(k-1)}$
so we have to write n^2 times the regex $R_{kk}^{(k-1)}$

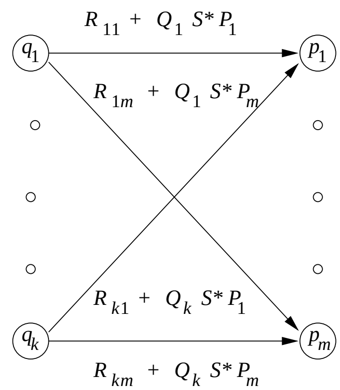
We need a more efficient approach:
the state elimination technique

The state elimination technique

Let's label the edges with regex's instead of symbols

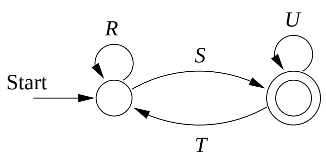


Now, let's eliminate state s.



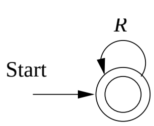
For each accepting state q eliminate from the original automaton all states exept q_0 and q.

For each $q \in F$ we'll be left with an A_q that looks like



that corresponds to the regex $E_q = (R + SU^*T)^*SU^*$

or with A_q looking like

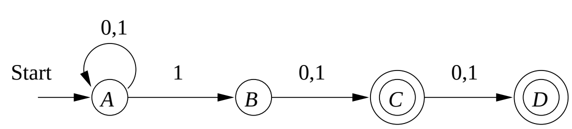


corresponding to the regex $E_q = R^*$

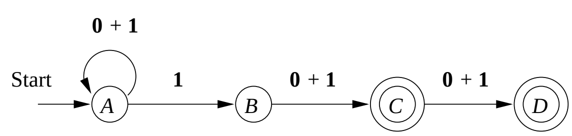
- The final expression is

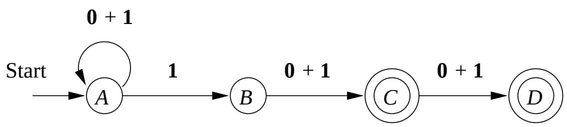
$$\bigoplus_{q \in F} E_q$$

Example: $\mathcal{A}$, where $L(\mathcal{A}) = \{W : w = x1b, \text{ or } w = x1bc, \ \ x \in \{0,1\}^*, \{b,c\} \subseteq \{0,1\}\}$

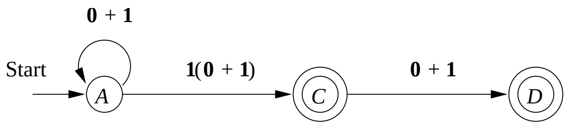


We turn this into an automaton with regex labels

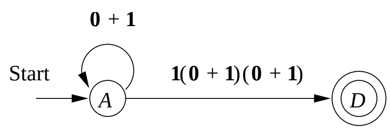




Let's eliminate state B

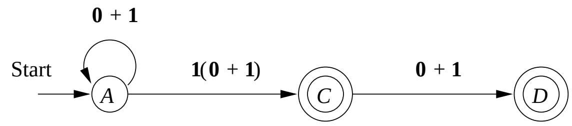


Then we eliminate state C and obtain $\mathcal{A}_D$

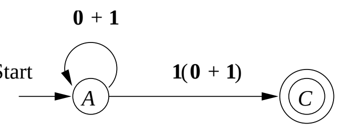


with regex $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1})$

From



we can eliminate D to obtain $\mathcal{A}_C$



with regex $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})$

- The final expression is the sum of the previous two regex's:

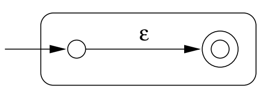
$$(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1}) + (\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})$$

From regex's to ϵ-NFA's

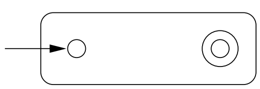
Theorem 3.7: For every regex R we can construct and ϵ-NFA A, s.t. $L(A) = L(R)$.

Proof: By structural induction:

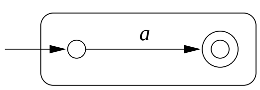
Basis: Automata for $\boldsymbol{\epsilon}$, $\emptyset$, and $\boldsymbol{a}$.



(a)

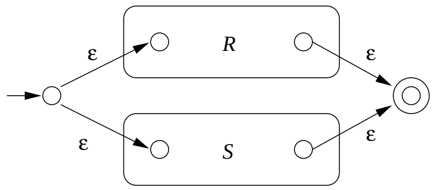


(b)

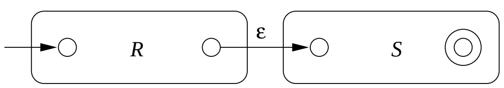


(c)

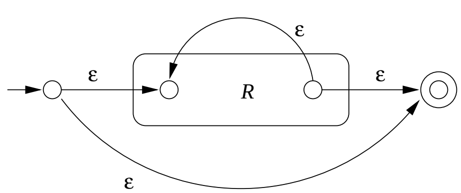
Induction: Automata for $R+S$, RS, and R^*



(a)

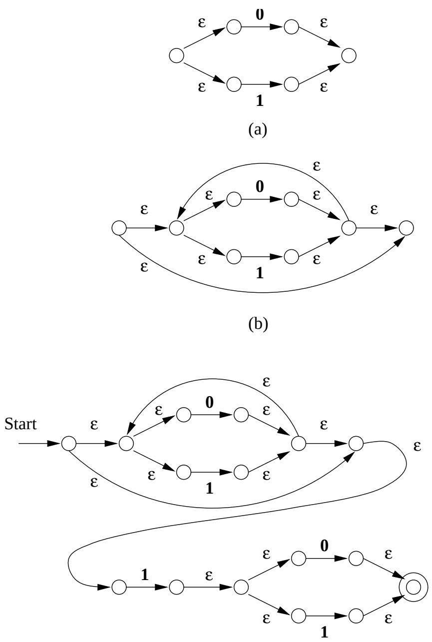


(b)



(c)

Example: We convert $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})$



(a) (b) (c)

Algebraic Laws for languages

- $L \cup M = M \cup L$.

Union is *commutative*.

- $(L \cup M) \cup N = L \cup (M \cup N)$.

Union is *associative*.

- $(LM)N = L(MN)$.

Concatenation is *associative*

Note: Concatenation is not commutative, *i.e.*, there are L and M such that $LM \neq ML$.

- $\emptyset \cup L = L \cup \emptyset = L$.

$\emptyset$ is *identity* for union.

- $\{\epsilon\}L = L\{\epsilon\} = L$.

$\{\epsilon\}$ is *left* and *right identity* for concatenation.

- $\emptyset L = L\emptyset = \emptyset$.

$\emptyset$ is *left* and *right annihilator* for concatenation.

- $L(M \cup N) = LM \cup LN$.

Concatenation is *left distributive* over union.

- $(M \cup N)L = ML \cup NL$.

Concatenation is *right distributive* over union.

- $L \cup L = L$.

Union is *idempotent*.

- $\emptyset^* = \{\epsilon\}$, $\{\epsilon\}^* = \{\epsilon\}$.

- $L^+ = LL^* = L^*L$, $L^* = L^+ \cup \{\epsilon\}$

• $(L^*)^* = L^*$. Closure is *idempotent*

Proof:

$$\begin{aligned} w \in (L^*)^* &\iff w \in \bigcup_{i=0}^{\infty} \left(\bigcup_{j=0}^{\infty} L^j \right)^i \\ &\iff \exists k, m \in \mathbb{N} : w \in (L^m)^k \\ &\iff \exists p \in \mathbb{N} : w \in L^p \\ &\iff w \in \bigcup_{i=0}^{\infty} L^i \\ &\iff w \in L^* \qquad \square \end{aligned}$$

Algebraic Laws for regex's

Evidently e.g. $L((\mathbf{0}+\mathbf{1})\mathbf{1}) = L(\mathbf{01}+\mathbf{11})$

Also e.g. $L((\mathbf{00}+\mathbf{101})\mathbf{11}) = L(\mathbf{0011}+\mathbf{10111})$.

More generally

$$L((E+F)G) = L(EG+FG)$$

for any regex's E, F, and G.

• How do we verify that a general identity like above is true?

1. Prove it by hand.

2. Let the computer prove it.

In Chapter 4 we will learn how to test automatically if $E = F$, for any *concrete* regex's E and F.

We want to test *general* identities, such as $\mathcal{E}+\mathcal{F} = \mathcal{F}+\mathcal{E}$, for *any* regex's $\mathcal{E}$ and $\mathcal{F}$.

Method:

1. "Freeze" $\mathcal{E}$ to $\boldsymbol{a}_1$, and $\mathcal{F}$ to $\boldsymbol{a}_2$

2. Test automatically if the frozen identity is true, e.g. if $L(\boldsymbol{a}_1+\boldsymbol{a}_2) = L(\boldsymbol{a}_2+\boldsymbol{a}_1)$

Question: Does this always work?

Answer: Yes, as long as the identities use only plus, dot, and star.

Let's denote a generalized regex, such as $(\mathcal{E}+\mathcal{F})\mathcal{E}$ by

$$\mathsf{E}(\mathcal{E}, \mathcal{F})$$

Now we can for instance make the substitution $\mathbf{S} = \{\mathcal{E}/\mathbf{0}, \mathcal{F}/\mathbf{11}\}$ to obtain

$$\mathbf{S}\,(\mathsf{E}(\mathcal{E}, \mathcal{F})) = (\mathbf{0}+\mathbf{11})\mathbf{0}$$

Theorem 3.13: Fix a "freezing" substitution $\spadesuit = \{\mathcal{E}_1/\boldsymbol{a}_1, \mathcal{E}_2/\boldsymbol{a}_2, \ldots, \mathcal{E}_m/\boldsymbol{a}_m\}$.

Let $\mathsf{E}(\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_m)$ be a generalized regex. Then for any regex's $E_1, E_2, \ldots, E_m$,

$$w \in L(\mathsf{E}(E_1, E_2, \ldots, E_m))$$

if and only if there are strings $w_i \in L(E_i)$, s.t.

$$w = w_{j_1} w_{j_2} \cdots w_{j_k}$$

and

$$a_{j_1} a_{j_2} \cdots a_{j_k} \in L(\mathsf{E}(\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_m))$$

For example: Suppose the alphabet is $\{1, 2\}$. Let $\mathsf{E}(\mathcal{E}_1, \mathcal{E}_2)$ be $(\mathcal{E}_1 + \mathcal{E}_2)\mathcal{E}_1$, and let E_1 be $\mathbf{1}$, and E_2 be $\mathbf{2}$. Then

$$w \in L(\mathsf{E}(E_1, E_2)) = L((E_1 + E_2)E_1) =$$

$$(\{\mathbf{1}\} \cup \{\mathbf{2}\})\{\mathbf{1}\} = \{\mathbf{11}, \mathbf{21}\}$$

if and only if

$\exists w_1 \in L(E_1) = \{\mathbf{1}\}$, $\exists w_2 \in L(E_2) = \{\mathbf{2}\} :\ w = w_{j_1} w_{j_2}$

and

$a_{j_1} a_{j_2} \in L(\mathsf{E}(\boldsymbol{a}_1, \boldsymbol{a}_2))) = L((\boldsymbol{a}_1 + \boldsymbol{a}_2)\boldsymbol{a}_1) = \{a_1 a_1, a_2 a_1\}$

if and only if

$j_1 = j_2 = 1$, or $j_1 = 1$, and $j_2 = 2$

Proof of Theorem 3.13: We do a structural induction of E.

Basis: If $\mathsf{E} = \boldsymbol{\epsilon}$, the frozen expression is also $\boldsymbol{\epsilon}$.

If $\mathsf{E} = \emptyset$, the frozen expression is also $\emptyset$.

If $\mathsf{E} = \boldsymbol{a}$, the frozen expression is also $\boldsymbol{a}$. Now $w \in L(\mathsf{E})$ if and only if there is $u \in L(\boldsymbol{a})$, s.t. $w = u$ and u is in the language of the frozen expression, i.e. $u \in \{\boldsymbol{a}\}$.

Induction:

Case 1: $\mathsf{E} = \mathsf{F} + \mathsf{G}$.

Then $\spadesuit(\mathsf{E}) = \spadesuit(\mathsf{F}) + \spadesuit(\mathsf{G})$, and $L(\spadesuit(\mathsf{E})) = L(\spadesuit(\mathsf{F})) \cup L(\spadesuit(\mathsf{G}))$

Let E and and F be regex's. Then $w \in L(E + F)$ if and only if $w \in L(E)$ or $w \in L(F)$, if and only if $\boldsymbol{a}_1 \in L(\spadesuit(\mathsf{F}))$ or $\boldsymbol{a}_2 \in L(\spadesuit(\mathsf{G}))$, if and only if $\boldsymbol{a}_1 \in \spadesuit(\mathsf{E})$, or $\boldsymbol{a}_2 \in \spadesuit(\mathsf{E})$.

Case 2: $\mathsf{E} = \mathsf{F}.\mathsf{G}$.

Then $\spadesuit(\mathsf{E}) = \spadesuit(\mathsf{F}).\spadesuit(\mathsf{G})$, and $L(\spadesuit(\mathsf{E})) = L(\spadesuit(\mathsf{F})).L(\spadesuit(\mathsf{G}))$

Let E and and F be regex's. Then $w \in L(E.F)$ if and only if $w = w_1 w_2$, $w_1 \in L(E)$ and $w_2 \in L(F)$, and $\boldsymbol{a}_1 \boldsymbol{a}_2 \in L(\spadesuit(\mathsf{F})).L(\spadesuit(\mathsf{G})) = \spadesuit(\mathsf{E})$

Case 3: $\mathsf{E} = \mathsf{F}^*$.

Prove this case at home.

Examples:

To prove $(\mathcal{L}+\mathcal{M})^* = (\mathcal{L}^*\mathcal{M}^*)^*$ it is enough to determine if $(\boldsymbol{a}_1+\boldsymbol{a}_2)^*$ is equivalent to $(\boldsymbol{a}_1^*\boldsymbol{a}_2^*)^*$

To verify $\mathcal{L}^* = \mathcal{L}^*\mathcal{L}^*$ test if $\boldsymbol{a}_1^*$ is equivalent to $\boldsymbol{a}_1^*\boldsymbol{a}_1^*$.

Question: Does $\mathcal{L}+\mathcal{M}\mathcal{L} = (\mathcal{L}+\mathcal{M})\mathcal{L}$ hold?

Theorem 3.14: $\mathsf{E}(\mathcal{E}_1,\ldots,\mathcal{E}_m) = \mathsf{F}(\mathcal{E}_1,\ldots,\mathcal{E}_m) \Leftrightarrow L(\spadesuit(\mathsf{E})) = L(\spadesuit(\mathsf{F}))$

Proof:

(Only if direction) $\mathsf{E}(\mathcal{E}_1,\ldots,\mathcal{E}_m) = \mathsf{F}(\mathcal{E}_1,\ldots,\mathcal{E}_m)$ means that $L(\mathsf{E}(E_1,\ldots,E_m)) = L(\mathsf{F}(E_1,\ldots,E_m))$ for any concrete regex's $E_1,\ldots,E_m$. In particular then $L(\spadesuit(\mathsf{E})) = L(\spadesuit(\mathsf{F}))$

(If direction) Let $E_1,\ldots,E_m$ be concrete regex's. Suppose $L(\spadesuit(\mathsf{E})) = L(\spadesuit(\mathsf{F}))$. Then by Theorem 3.13,

$$w \in L(\mathsf{E}(E_1,\ldots E_m)) \Leftrightarrow$$

$$\exists w_i \in L(E_i), w = w_{j_1}\cdots w_{j_m}, a_{j_1}\cdots a_{j_m} \in L(\spadesuit(\mathsf{E})) \Leftrightarrow$$

$$\exists w_i \in L(E_i), w = w_{j_1}\cdots w_{j_m}, a_{j_1}\cdots a_{j_m} \in L(\spadesuit(\mathsf{F})) \Leftrightarrow$$

$$w \in L(\mathsf{F}(E_1,\ldots E_m))$$

Examples:

To prove $(\mathcal{L}+\mathcal{M})^* = (\mathcal{L}^*\mathcal{M}^*)^*$ it is enough to determine if $(\boldsymbol{a}_1+\boldsymbol{a}_2)^*$ is equivalent to $(\boldsymbol{a}_1^*\boldsymbol{a}_2^*)^*$

To verify $\mathcal{L}^* = \mathcal{L}^*\mathcal{L}^*$ test if $\boldsymbol{a}_1^*$ is equivalent to $\boldsymbol{a}_1^*\boldsymbol{a}_1^*$.

Question: Does $\mathcal{L}+\mathcal{M}\mathcal{L} = (\mathcal{L}+\mathcal{M})\mathcal{L}$ hold?

Theorem 3.14: $\mathsf{E}(\mathcal{E}_1,\ldots,\mathcal{E}_m) = \mathsf{F}(\mathcal{E}_1,\ldots,\mathcal{E}_m) \Leftrightarrow L(\spadesuit(\mathsf{E})) = L(\spadesuit(\mathsf{F}))$

Proof:

(Only if direction) $\mathsf{E}(\mathcal{E}_1,\ldots,\mathcal{E}_m) = \mathsf{F}(\mathcal{E}_1,\ldots,\mathcal{E}_m)$ means that $L(\mathsf{E}(E_1,\ldots,E_m)) = L(\mathsf{F}(E_1,\ldots,E_m))$ for any concrete regex's $E_1,\ldots,E_m$. In particular then $L(\spadesuit(\mathsf{E})) = L(\spadesuit(\mathsf{F}))$

(If direction) Let $E_1,\ldots,E_m$ be concrete regex's. Suppose $L(\spadesuit(\mathsf{E})) = L(\spadesuit(\mathsf{F}))$. Then by Theorem 3.13,

$$w \in L(\mathsf{E}(E_1,\ldots E_m)) \Leftrightarrow$$

$$\exists w_i \in L(E_i), w = w_{j_1}\cdots w_{j_m}, a_{j_1}\cdots a_{j_m} \in L(\spadesuit(\mathsf{E})) \Leftrightarrow$$

$$\exists w_i \in L(E_i), w = w_{j_1}\cdots w_{j_m}, a_{j_1}\cdots a_{j_m} \in L(\spadesuit(\mathsf{F})) \Leftrightarrow$$

$$w \in L(\mathsf{F}(E_1,\ldots E_m))$$

Properties of Regular Languages

- *Pumping Lemma.* Every regular language satisfies the pumping lemma. If somebody presents you with fake regular language, use the pumping lemma to show a contradiction.

- *Closure properties.* Building automata from components through operations, e.g. given L and M we can build an automaton for $L \cap M$.

- *Decision properties.* Computational analysis of automata, e.g. are two automata equivalent.

- *Minimization techniques.* We can save money since we can build smaller machines.

The Pumping Lemma Informally

Suppose $L_{01} = \{0^n 1^n : n \geq 1\}$ were regular.

Then it would be recognized by some DFA A, with, say, k states.

Let A read 0^k. On the way it will travel as follows:

$$\begin{array}{ll} \epsilon & p_0 \\ 0 & p_1 \\ 00 & p_2 \\ \ldots & \ldots \\ 0^k & p_k \end{array}$$

$\Rightarrow \exists i < j : p_i = p_j$ Call this state q.

$\Rightarrow \widehat{\delta}(p_0, 0^i) = \widehat{\delta}(p_0, 0^j) = q$

Now you can fool A:

If $\widehat{\delta}(q, 1^i) \in F$ the machine will foolishly accept $0^j 1^i$.

If $\widehat{\delta}(q, 1^i) \notin F$ the machine will foolishly reject $0^i 1^i$.

Therefore L_{01} cannot be regular.

- Let's generalize the above reasoning.

Theorem 4.1.

The Pumping Lemma for Regular Languages.

Let L be regular.

Then $\exists n, \forall w \in L : |w| \geq n \Rightarrow w = xyz$ such that

1. $y \neq \epsilon$
2. $|xy| \leq n$
3. $\forall k \geq 0,\ xy^k z \in L$

Proof: Suppose L is regular

The L is recognized by some DFA A with, say, n states.

Let $w = a_1 a_2 \ldots a_m \in L,\ m > n$.

Let $p_i = \hat{\delta}(q_0, a_1 a_2 \ldots a_i)$.

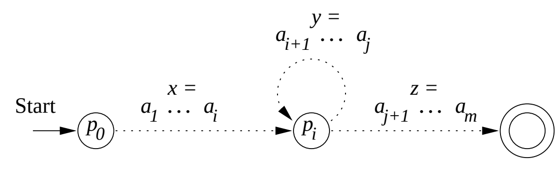
$\Rightarrow \exists i < j : p_i = p_j$

Now $w = xyz$, where

1. $x = a_1 a_2 \ldots a_i$

2. $y = a_{i+1} a_{i+2} \ldots a_j$

3. $z = a_{j+1} a_{j+2} \ldots a_m$



Evidently $xy^k z \in L$, for any $k \geq 0$. *Q.E.D.*

Example: Let L_{eq} be the language of strings with equal number of zero's and one's.

Suppose L_{eq} is regular. Then $L_{eq} = L(A)$, for some DFA A with, say, n states, and $w = 0^n 1^n \in L(A)$.

By the pumping lemma $w = xyz$, $|xy| \leq n$, $y \neq \epsilon$ and $xy^k z \in L(A)$

$$w = \underbrace{000\ldots}_{x}\underbrace{\ldots 0}_{y}\underbrace{0111\ldots 11}_{z}$$

In particular, $xz \in L(A)$, but xz has fewer 0's than 1's. $\Rightarrow L(A) \neq L_{eq}$.

Suppose $L_{pr} = \{1^p : p \text{ is prime }\}$ were regular.

Then $L_{pr} = L(A)$, for some DFA A with, say n, states.

Choose a prime $p \geq n + 2$.

$$w = \overbrace{\underbrace{111\cdots}_{x}\underbrace{\cdots 1}_{\substack{y \\ |y|=m}}\underbrace{1111\cdots 11}_{z}}^{p}$$

Now $xy^{p-m}z \in L(A)$

$|xy^{p-m}z| = |xz| + (p-m)|y| =$
$p - m + (p-m)m = (1+m)(p-m)$
which is not prime unless one of the factors is 1.

- $y \neq \epsilon \Rightarrow 1 + m > 1$
- $m = |y| \leq |xy| \leq n, \ \ p \geq n+2$
  $\Rightarrow p - m \geq n + 2 - n = 2$.

$\Rightarrow L(A) \neq L_{pr}$.

Closure Properties of Regular Languages

Let L and M be regular languages. Then the following languages are all regular:

- *Union:* $L \cup M$
- *Intersection:* $L \cap M$
- *Complement:* $\overline{N}$
- *Difference:* $L \setminus M$
- *Reversal:* $L^R = \{w^R : w \in L\}$
- *Closure:* L^*.
- *Concatenation:* $L.M$
- *Homomorphism:*
$h(L) = \{h(w) : w \in L, h$ is a homom. $\}$
- *Inverse homomorphism:*
$h^{-1}(L) = \{w \in \Sigma : h(w) \in L, h : \Sigma \rightarrow \Delta$ is a homom.$\}$

Theorem 4.4. For any regular L and M, $L \cup M$ is regular.

Proof. Let $L = L(E)$ and $M = L(F)$. Then $L(E + F) = L \cup M$ by definition.

Theorem 4.5. If L is a regular language over Σ, then so is $\overline{L} = \Sigma^* \setminus L$.

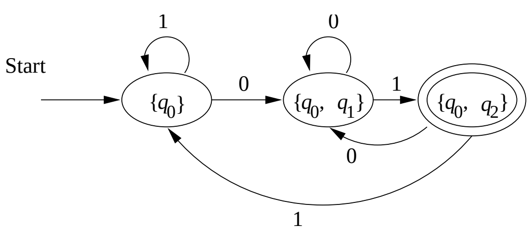
Proof. Let L be recognized by a DFA

$$A = (Q, \Sigma, \delta, q_0, F).$$

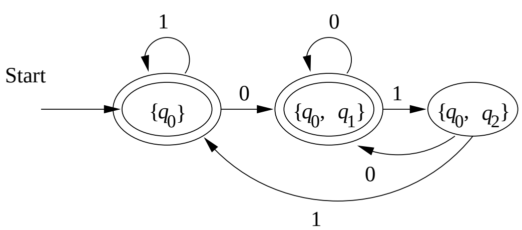
Let $B = (Q, \Sigma, \delta, q_0, Q \setminus F)$. Now $L(B) = \overline{L}$.

Example:

Let L be recognized by the DFA below



Then $\overline{L}$ is recognized by



Question: What are the regex's for L and $\overline{L}$

Theorem 4.8. If L and M are regular, then so is $L \cap M$.

Proof. By DeMorgan's law $L \cap M = \overline{\overline{L} \cup \overline{M}}$. We already that regular languages are closed under complement and union.

We shall shall also give a nice direct proof, the *Cartesian* construction from the e-commerce example.

Theorem 4.8. If L and M are regular, then so in $L \cap M$.

Proof. Let L be the language of

$$A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$$

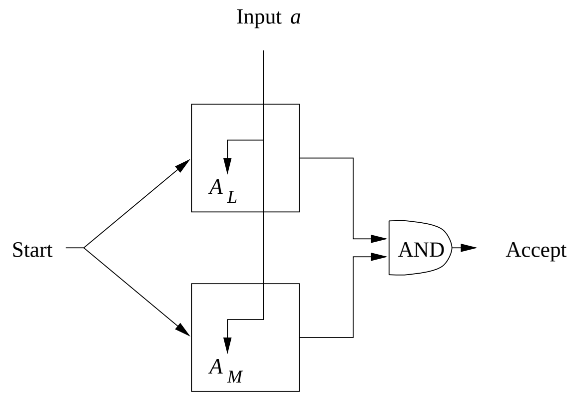
and M be the language of

$$A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$$

We assume w.l.o.g. that both automata are deterministic.

We shall construct an automaton that simulates A_L and A_M in parallel, and accepts if and only if both A_L and A_M accept.

If A_L goes from state p to state s on reading a, and A_M goes from state q to state t on reading a, then $A_{L\cap M}$ will go from state (p,q) to state (s,t) on reading a.



Formally

$$A_{L\cap M} = (Q_L \times Q_M, \Sigma, \delta_{L\cap M}, (q_L, q_M), F_L \times F_M),$$

where

$$\delta_{L\cap M}((p,q),a) = (\delta_L(p,a), \delta_M(q,a))$$

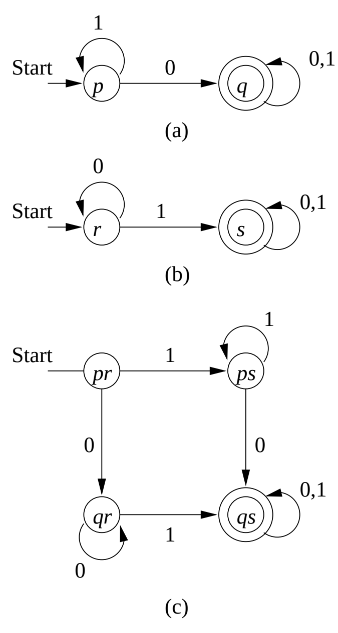
It will be shown in the tutorial by and induction on $|w|$ that

$$\hat{\delta}_{L\cap M}((q_L,q_M),w) = \left(\hat{\delta}_L(q_L,w), \hat{\delta}_M(q_M,w)\right)$$

The claim then follows.

Question: Why?

Example: $(c) = (a) \times (b)$



(a) (b) (c)

Theorem 4.10. If L and M are regular languages, then so in $L \setminus M$.

Proof. Observe that $L \setminus M = L \cap \overline{M}$. We already know that regular languages are closed under complement and intersection.

Theorem 4.11. If L is a regular language, then so is L^R.

Proof 1: Let L be recognized by an FA A. Turn A into an FA for L^R, by

1. Reversing all arcs.

2. Make the old start state the new sole accepting state.

3. Create a new start state p_0, with $\delta(p_0, \epsilon) = F$ (the old accepting states).

Theorem 4.11. If L is a regular language, then so is L^R.

Proof 2: Let L be described by a regex E. We shall construct a regex E^R, such that $L(E^R) = (L(E))^R$.

We proceed by a structural induction on E.

Basis: If E is $\boldsymbol{\epsilon}$, $\boldsymbol{\emptyset}$, or $\boldsymbol{a}$, then $E^R = E$.

Induction:

1. $E = F + G$. Then $E^R = F^R + G^R$

2. $E = F.G$. Then $E^R = G^R.F^R$

3. $E = F^*$. Then $E^R = (F^R)^*$

We will show by structural induction on E on blackboard in class that

$$L(E^R) = (L(E))^R$$

Homomorphisms

A *homomorphism* on Σ is a function $h : \Sigma \to \Theta^*$, where Σ and Θ are alphabets.

Let $w = a_1 a_2 \cdots a_n \in \Sigma^*$. Then

$$h(w) = h(a_1)h(a_2)\cdots h(a_n)$$

and

$$h(L) = \{h(w) : w \in L\}$$

Example: Let $h : \{0, 1\} \to \{a, b\}^*$ be defined by $h(0) = ab$, and $h(1) = \epsilon$. Now $h(0011) = abab$.

Example: $h(L(\mathbf{10^*1})) = L((\boldsymbol{ab})^*)$.

Theorem 4.14: $h(L)$ is regular, whenever L is.

Proof:

Let $L = L(E)$ for a regex E. We claim that $L(h(E)) = h(L)$.

Basis: If E is $\boldsymbol{\epsilon}$ or $\emptyset$. Then $h(E) = E$, and $L(h(E)) = L(E) = h(L(E))$.

If E is $\boldsymbol{a}$, then $L(E) = \{a\}$, $L(h(E)) = L(h(\boldsymbol{a})) = \{h(a)\} = h(L(E))$.

Induction:

Case 1: $L = E + F$. Now $L(h(E + F)) = L(h(E)+h(F)) = L(h(E)) \cup L(h(F)) = h(L(E)) \cup h(L(F)) = h(L(E) \cup L(F)) = h(L(E + F))$.

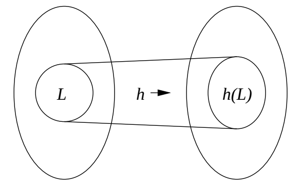
Case 2: $L = E.F$. Now $L(h(E.F)) = L(h(E)).L(h(F)) = h(L(E)).h(L(F)) = h(L(E).L(F))$

Case 3: $L = E^*$. Now $L(h(E^*)) = L(h(E)^*) = L(h(E))^* = h(L(E))^* = h(L(E^*))$.

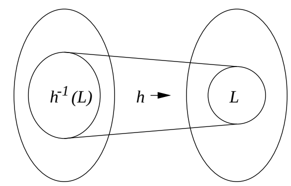
Inverse Homomorphism

Let $h : \Sigma \to \Theta^*$ be a homom. Let $L \subseteq \Theta^*$, and define

$$h^{-1}(L) = \{w \in \Sigma^* : h(w) \in L\}$$



(a)



(b)

Example: Let $h : \{a, b\} \to \{0, 1\}^*$ be defined by $h(a) = 01$, and $h(b) = 10$. If $L = L((\mathbf{00} + \mathbf{1})^*)$, then $h^{-1}(L) = L((\boldsymbol{ba})^*)$.

Claim: $h(w) \in L$ if and only if $w = (ba)^n$

Proof: Let $w = (ba)^n$. Then $h(w) = (1001)^n \in L$.

Let $h(w) \in L$, and suppose $w \notin L((\boldsymbol{ba})^*)$. There are four cases to consider.

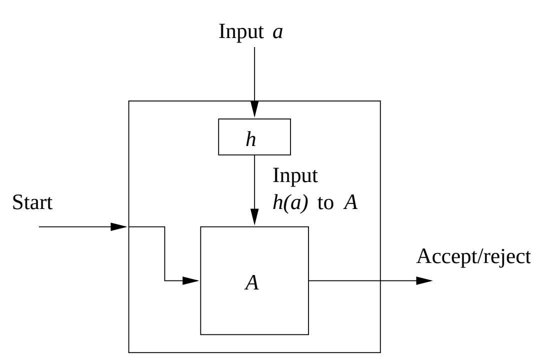
1. w begins with a. Then $h(w)$ begins with 01 and $\notin L((\mathbf{00} + \mathbf{1})^*)$.
2. w ends in b. Then $h(w)$ ends in 10 and $\notin L((\mathbf{00} + \mathbf{1})^*)$.
3. $w = xaay$. Then $h(w) = z0101v$ and $\notin L((\mathbf{00} + \mathbf{1})^*)$.
4. $w = xbby$. Then $h(w) = z1010v$ and $\notin L((\mathbf{00} + \mathbf{1})^*)$.

Theorem 4.16: Let $h : \Sigma \to \Theta^*$ be a homom., and $L \subseteq \Theta^*$ regular. Then $h^{-1}(L)$ is regular.

Proof: Let L be the language of $A = (Q, \Theta, \delta, q_0, F)$. We define $B = (Q, \Sigma, \gamma, q_0, F)$, where

$$\gamma(q, a) = \hat{\delta}(q, h(a))$$

It will be shown by induction on $|w|$ in the tutorial that $\hat{\gamma}(q_0, w) = \hat{\delta}(q_0, h(w))$



Decision Properties

We consider the following:

1. Converting among representations for regular languages.

2. Is $L = \emptyset$?

3. Is $w \in L$?

4. Do two descriptions define the same language?

From NFA's to DFA's

Suppose the ϵ-NFA has n states.

To compute ECLOSE(p) we follow at most n^2 arcs.

The DFA has 2^n states, for each state S and each $a \in \Sigma$ we compute $\delta_D(S, a)$ in n^3 steps. Grand total is $O(n^3 2^n)$ steps.

If we compute δ for reachable states only, we need to compute $\delta_D(S, a)$ only s times, where s is the number of reachable states. Grand total is $O(n^3 s)$ steps.

From DFA to NFA

All we need to do is to put set brackets around the states. Total $O(n)$ steps.

From FA to regex

We need to compute n^3 entries of size up to 4^n. Total is $O(n^3 4^n)$.

The FA is allowed to be a NFA. If we first wanted to convert the NFA to a DFA, the total time would be doubly exponential

From regex to FA's We can build an expression tree for the regex in n steps.

We can construct the automaton in n steps.

Eliminating ϵ-transitions takes $O(n^3)$ steps.

If you want a DFA, you might need an exponential number of steps.

Testing emptiness

$L(A) \neq \emptyset$ for FA A if and only if a final state is reachable from the start state in A. Total $O(n^2)$ steps.

Alternatively, we can inspect a regex E and tell if $L(E) = \emptyset$. We use the following method:

$E = F + G$. Now $L(E)$ is empty if and only if both $L(F)$ and $L(G)$ are empty.

$E = F.G$. Now $L(E)$ is empty if and only if either $L(F)$ or $L(G)$ is empty.

$E = F^*$. Now $L(E)$ is never empty, since $\epsilon \in L(E)$.

$E = \epsilon$. Now $L(E)$ is not empty.

$E = \mathbf{a}$. Now $L(E)$ is not empty.

$E = \boldsymbol{\emptyset}$. Now $L(E)$ is empty.

Testing membership

To test $w \in L(A)$ for DFA A, simulate A on w. If $|w| = n$, this takes $O(n)$ steps.

If A is an NFA and has s states, simulating A on w takes $O(ns^2)$ steps.

If A is an ϵ-NFA and has s states, simulating A on w takes $O(ns^3)$ steps.

If $L = L(E)$, for regex E of length s, we first convert E to an ϵ-NFA with $2s$ states. Then we simulate w on this machine, in $O(ns^3)$ steps.

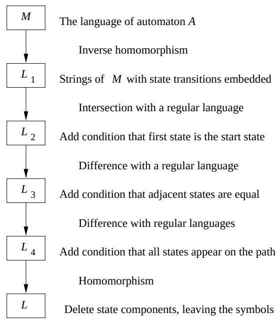
Example 4.17

Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA and $L(A) = M$.

Let $L \subseteq M$ be those words in $w \in L(A)$ for which A visits every state in Q at least once when accepting w.

We shall use closure properties of regular languages to prove that L is regular.

Plan of the proof:



- $M \rightsquigarrow L_1$

Define $T = \{[paq] : p, q \in Q, a \in \Sigma, \delta(p, a) = q\}$

Let $h : T^* \to \Sigma^*$ be the homom. defined by

$$h([paq]) = a$$

Let $L_1 = h^{-1}(M)$. Since M is regular, so is L_1

Example: Suppose A is given by

δ	0	1
$\to p$	q	p
$\star q$	q	q

Then $T = \{[p0q], [p1p], [q0q], [q1q]\}$.

For example $h^{-1}(101) =$

$$\{ [p1p][p0q][p1p], [p1p][p0q][q1q], [p1p][q0q][p1p], [p1p][q0q][q1q], [q1q][p0q][p1p], [q1q][p0q][q1q], [q1q][q0q][p1p], [q1q][q0q][q1q] \}$$

- $L_1 \rightsquigarrow L_2$

Define

$$E_1 = \bigoplus_{a \in \Sigma,\ \delta(q_0,a)=p} [q_0 a p]$$

Let

$$L_2 = L_1 \cap \Big(L(E_1).T^*\Big)$$

Now L_2 is regular and consists of those strings in L_1 starting with $[q_o \ldots][\ldots$

- $L_2 \rightsquigarrow L_3$

Define

$$E_2 = \bigoplus_{[paq]\in T^*,\ [rbs]\in T^*,\ q\neq r} [paq][rbs]$$

Let

$$L_3 = L_2 \setminus T^*.L(E_2).T^*$$

Now L_3 is regular and consists of those strings

$$[q_0 a_1 p_1][p_1 a_2 p_2] \ldots [p_{n-1} a_n p_n]$$

in T^* such that

$$a_1 a_2 \ldots a_n \in L(A)$$

$$\delta(q_o, a_1) = p_1$$

$$\delta(p_i, a_{i+1}) = p_{i+1},\ i \in \{1, 2, \ldots, n-1\}$$

$$p_n \in F$$

- $L_3 \rightsquigarrow L_4$

Define

$$E_q = \bigoplus_{[ras]\in T^*,\ r\neq q,\ s\neq q} [ras]$$

Now $L_3 \setminus L(E_q^*)$ consists of those strings in L_3 that "visit" state q at least once.

Let

$$L_4 = L_3 \setminus L\Big(\bigoplus_{q\in Q} E_q^*\Big)$$

Now L_4 is regular and consists of those strings in L_3 that "visit" *all* states $q \in Q$ at least once.

- $L_4 \rightsquigarrow L$

We only need to get rid of the state components in the words of L_4.

We can do this by letting

$$L = h(L_4)$$

Now $L =$

$$\{w : \hat{\delta}(q_0, w) \in F, \forall q \in Q\ \exists x_q, y(w = x_q y \wedge \hat{\delta}(q_0, x_q) = q)\}$$

Equivalence and Minimization of Automata

Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA, and $\{p, q\} \subseteq Q$. We define

$$p \equiv q \;\Leftrightarrow\; \forall w \in \Sigma^* \;:\; \hat{\delta}(p, w) \in F \text{ iff } \hat{\delta}(q, w) \in F$$

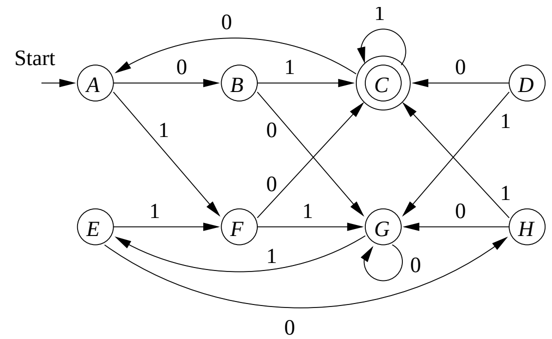
- If $p \equiv q$ we say that p and q are *equivalent*

- If $p \not\equiv q$ we say that p and q are *distinguishable*

IOW (in other words) p and q are distinguishable iff

$\exists w : \hat{\delta}(p, w) \in F$ and $\hat{\delta}(q, w) \notin F$, or vice versa

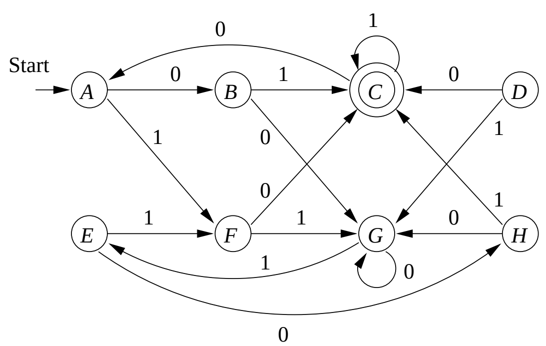
Example:



$\hat{\delta}(C, \epsilon) \in F, \;\; \hat{\delta}(G, \epsilon) \notin F \;\Rightarrow\; C \not\equiv G$

$\hat{\delta}(A, 01) = C \in F, \;\; \hat{\delta}(G, 01) = E \notin F \;\Rightarrow\; A \not\equiv G$

What about A and E?



$\hat{\delta}(A, \epsilon) = A \notin F, \;\; \hat{\delta}(E, \epsilon) = E \notin F$

$\hat{\delta}(A, 1) = F = \hat{\delta}(E, 1)$

Therefore $\hat{\delta}(A, 1x) = \hat{\delta}(E, 1x) = \hat{\delta}(F, x)$

$\hat{\delta}(A, 00) = G = \hat{\delta}(E, 00)$

$\hat{\delta}(A, 01) = C = \hat{\delta}(E, 01)$

Conclusion: $A \equiv E$.

We can compute distinguishable pairs with the following inductive *table filling algorithm*:

Basis: If $p \in F$ and $q \notin F$, then $p \not\equiv q$.

Induction: If $\exists a \in \Sigma : \delta(p, a) \not\equiv \delta(q, a)$, then $p \not\equiv q$.

Example:
Applying the table filling algo to DFA A:

B	x						
C	x	x					
D	x	x	x				
E		x	x	x			
F	x	x	x		x		
G	x	x	x	x	x	x	
H	x		x	x	x	x	x
	A	B	C	D	E	F	G

Theorem 4.20: If p and q are not distinguished by the TF-algo, then $p \equiv q$.

Proof: Suppose to the contrary that that there is a *bad pair* $\{p, q\}$, s.t.

1. $\exists w : \hat{\delta}(p, w) \in F, \hat{\delta}(q, w) \notin F$, or vice versa.

2. The TF-algo does not distinguish between p and q.

Let $w = a_1 a_2 \cdots a_n$ be the shortest string that identifies a bad pair $\{p, q\}$.

Now $w \neq \epsilon$ since otherwise the TF-algo would in the basis distinguish p from q. Thus $n \geq 1$.

Consider states $r = \delta(p, a_1)$ and $s = \delta(q, a_1)$. Now $\{r, s\}$ cannot be a bad pair since $\{r, s\}$ would be indentified by a string shorter than w. Therefore, the TF-algo must have discovered that r and s are distinguishable.

But then the TF-algo would distinguish p from q in the inductive part.

Thus there are no bad pairs and the theorem is true.

Testing Equivalence of Regular Languages

Let L and M be reg langs (each given in some form).

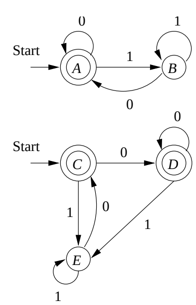
To test if $L = M$

1. Convert both L and M to DFA's.

2. Imagine the DFA that is the union of the two DFA's (never mind there are two start states)

3. If TF-algo says that the two start states are distinguishable, then $L \neq M$, otherwise $L = M$.

Example:



We can "see" that both DFA accept $L(\epsilon + (\mathbf{0}+\mathbf{1})^*\mathbf{0})$. The result of the TF-algo is

	A	B	C	D
B	x			
C		x		
D		x		
E	x		x	x

Therefore the two automata are equivalent.

Minimization of DFA's

We can use the TF-algo to minimize a DFA by merging all equivalent states. IOW, replace each state p by $p/_\equiv$.

Example: The DFA on slide 119 has equivalence classes $\{\{A, E\}, \{B, H\}, \{C\}, \{D, F\}, \{G\}\}$.

The "union" DFA on slide 125 has equivalence classes $\{\{A, C, D\}, \{B, E\}\}$.

Note: In order for $p/_\equiv$ to be an *equivalence class*, the relation $\equiv$ has to be an *equivalence relation* (reflexive, symmetric, and transitive).

Theorem 4.23: If $p \equiv q$ and $q \equiv r$, then $p \equiv r$.

Proof: Suppose to the contrary that $p \not\equiv r$. Then $\exists w$ such that $\hat{\delta}(p, w) \in F$ and $\hat{\delta}(r, w) \notin F$, or vice versa.

OTH, $\hat{\delta}(q, w)$ is either accpeting or not.

Case 1: $\hat{\delta}(q, w)$ is accepting. Then $q \not\equiv r$.

Case 1: $\hat{\delta}(q, w)$ is not accepting. Then $p \not\equiv q$.

The vice versa case is proved symmetrically

Therefore it must be that $p \equiv r$.

To minimize a DFA $A = (Q, \Sigma, \delta, q_0, F)$ construct a DFA $B = (Q/_\equiv, \Sigma, \gamma, q_0/_\equiv, F/_\equiv)$, where

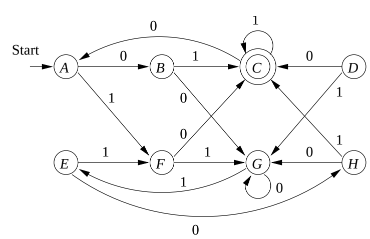
$$\gamma(p/_\equiv, a) = \delta(p, a)/_\equiv$$

In order for B to be well defined we have to show that

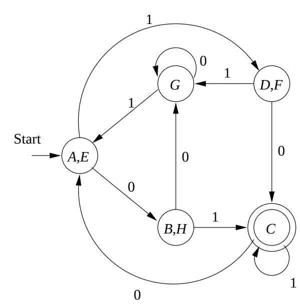
$$\text{If } p \equiv q \text{ then } \delta(p, a) \equiv \delta(q, a)$$

If $\delta(p, a) \not\equiv \delta(q, a)$, then the TF-algo would conclude $p \not\equiv q$, so B is indeed well defined. Note also that $F/_\equiv$ contains all and only the accepting states of A.

Example: We can minimize

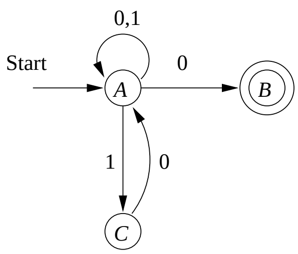


to obtain



NOTE: We cannot apply the TF-algo to NFA's.

For example, to minimize



we simply remove state C.

However, $A \not\equiv C$.

Why the Minimized DFA Can't Be Beaten

Let B be the minimized DFA obtained by applying the TF-algo to DFA A.

We already know that $L(A) = L(B)$.

What if there existed a DFA C, with $L(C) = L(B)$ and fewer states than B?

Then run the TF-algo on B "union" C.

Since $L(B) = L(C)$ we have $q_0^B \equiv q_0^C$.

Also, $\delta(q_0^B, a) \equiv \delta(q_0^C, a)$, for any a.

Claim: For each state p in B there is at least one state q in C, s.t. $p \equiv q$.

Proof of claim: There are no inaccessible states, so $p = \widehat{\delta}(q_0^B, a_1a_2 \cdots a_k)$, for some string $a_1a_2 \cdots a_k$. Now $q = \widehat{\delta}(q_0^C, a_1a_2 \cdots a_k)$, and $p \equiv q$.

Since C has fewer states than B, there must be two states r and s of B such that $r \equiv t \equiv s$, for some state t of C. But then $r \equiv s$ (why?) which is a contradiction, since B was constructed by the TF-algo.

Context-Free Grammars and Languages

- We have seen that many languages cannot be regular. Thus we need to consider larger classes of langs.

- *Contex-Free Languages* (CFL's) played a central role natural languages since the 1950's, and in compilers since the 1960's.

- *Context-Free Grammars* (CFG's) are the basis of BNF-syntax.

- Today CFL's are increasingly important for XML and their DTD's.

We'll look at: CFG's, the languages they generate, parse trees, pushdown automata, and closure properties of CFL's.

Informal example of CFG's

Consider $L_{pal} = \{w \in \Sigma^* : w = w^R\}$

For example `otto` $\in L_{pal}$, `madamimadam` $\in L_{pal}$.

In Finnish language e.g. `saippuakauppias` $\in L_{pal}$ ("soap-merchant")

Let $\Sigma = \{0, 1\}$ and suppose L_{pal} were regular.

Let n be given by the pumping lemma. Then $0^n 1 0^n \in L_{pal}$. In reading 0^n the FA must make a loop. Omit the loop; contradiction.

Let's define L_{pal} inductively:

Basis: $\epsilon, 0$, and 1 are palindromes.

Induction: If w is a palindrome, so are $0w0$ and $1w1$.

Circumscription: Nothing else is a palindrome.

CFG's is a formal mechanism for definitions such as the one for L_{pal}.

1. $P \rightarrow \epsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0P0$
5. $P \rightarrow 1P1$

0 and 1 are *terminals*

P is a *variable* (or *nonterminal*, or *syntactic category*)

P is in this grammar also the *start symbol*.

1–5 are *productions* (or *rules*)

Formal definition of CFG's

A *context-free grammar* is a quadruple

$$G = (V, T, P, S)$$

where

V is a finite set of *variables*.

T is a finite set of *terminals*.

P is a finite set of *productions* of the form $A \rightarrow \alpha$, where A is a variable and $\alpha \in (V \cup T)^*$

S is a designated variable called the *start symbol*.

Example: $G_{pal} = (\{P\}, \{0, 1\}, A, P)$, where $A = \{P \rightarrow \epsilon, P \rightarrow 0, P \rightarrow 1, P \rightarrow 0P0, P \rightarrow 1P1\}$.

Sometimes we group productions with the same head, e.g. $A = \{P \rightarrow \epsilon|0|1|0P0|1P1\}$.

Example: Regular expressions over $\{0, 1\}$ can be defined by the grammar

$$G_{regex} = (\{E\}, \{0, 1\}, A, E)$$

where $A =$

$\{E \rightarrow \mathbf{0}, E \rightarrow \mathbf{1}, E \rightarrow E.E, E \rightarrow E{+}E, E \rightarrow E^\star, E \rightarrow (E)\}$

Example: (simple) expressions in a typical prog lang. Operators are $+$ and $*$, and arguments are identfiers, i.e. strings in $L((\boldsymbol{a}+\boldsymbol{b})(\boldsymbol{a}+\boldsymbol{b}+\boldsymbol{0}+\boldsymbol{1})^*)$

The expressions are defined by the grammar

$$G = (\{E, I\}, T, P, E)$$

where $T = \{+, *, (,), a, b, 0, 1\}$ and P is the following set of productions:

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

Derivations using grammars

- *Recursive inference,* using productions from body to head
- *Derivations,* using productions from head to body.

Example of recursive inference:

	String	Lang	Prod	String(s) used
(i)	a	I	5	-
(ii)	b	I	6	-
(iii)	$b0$	I	9	*(ii)*
(iv)	$b00$	I	9	*(iii)*
(v)	a	E	1	*(i)*
(vi)	$b00$	E	1	*(iv)*
(vii)	$a + b00$	E	2	*(v), (vi)*
(viii)	$(a + b00)$	E	4	*(vii)*
(ix)	$a * (a + b00)$	E	3	*(v), (viii)*

- Derivations

Let $G = (V, T, P, S)$ be a CFG, $A \in V$, $\{\alpha, \beta\} \subset (V \cup T)^*$, and $A \rightarrow \gamma \in P$.

Then we write

$$\alpha A \beta \underset{G}{\Rightarrow} \alpha \gamma \beta$$

or, if G is understood

$$\alpha A \beta \Rightarrow \alpha \gamma \beta$$

and say that $\alpha A \beta$ *derives* $\alpha \gamma \beta$.

We define $\overset{*}{\Rightarrow}$ to be the reflexive and transitive closure of $\Rightarrow$, IOW:

Basis: Let $\alpha \in (V \cup T)^*$. Then $\alpha \overset{*}{\Rightarrow} \alpha$.

Induction: If $\alpha \overset{*}{\Rightarrow} \beta$, and $\beta \Rightarrow \gamma$, then $\alpha \overset{*}{\Rightarrow} \gamma$.

Example: Derivation of $a * (a + b00)$ from E in the grammar of slide 138:

$$E \Rightarrow E * E \Rightarrow I * E \Rightarrow a * E \Rightarrow a * (E) \Rightarrow$$

$$a*(E+E) \Rightarrow a*(I+E) \Rightarrow a*(a+E) \Rightarrow a*(a+I) \Rightarrow$$

$$a * (a + I0) \Rightarrow a * (a + I00) \Rightarrow a * (a + b00)$$

Note: At each step we might have several rules to choose from, e.g.
$I * E \Rightarrow a * E \Rightarrow a * (E)$, versus
$I * E \Rightarrow I * (E) \Rightarrow a * (E)$.

Note: Not all choices lead to successful derivations of a particular string, for instance

$$E \Rightarrow E + E$$

won't lead to a derivation of $a * (a + b00)$.

Leftmost and Rightmost Derivations

Leftmost derivation $\underset{lm}{\Rightarrow}$ Always replace the leftmost variable by one of its rule-bodies.

Rightmost derivation $\underset{rm}{\Rightarrow}$ Always replace the rightmost variable by one of its rule-bodies.

Leftmost: The derivation on the previous slide.

Rightmost:

$$E \underset{rm}{\Rightarrow} E * E \underset{rm}{\Rightarrow}$$

$$E*(E) \underset{rm}{\Rightarrow} E*(E+E) \underset{rm}{\Rightarrow} E*(E+I) \underset{rm}{\Rightarrow} E*(E+I0)$$

$$\underset{rm}{\Rightarrow} E*(E+I00) \underset{rm}{\Rightarrow} E*(E+b00) \underset{rm}{\Rightarrow} E*(I+b00)$$

$$\underset{rm}{\Rightarrow} E*(a+b00) \underset{rm}{\Rightarrow} I*(a+b00) \underset{rm}{\Rightarrow} a*(a+b00)$$

We can conclude that $E \overset{*}{\underset{rm}{\Rightarrow}} a*(a+b00)$

The Language of a Grammar

If $G(V,T,P,S)$ is a CFG, then the *language of* G is

$$L(G) = \{w \in T^* : S \overset{*}{\underset{G}{\Rightarrow}} w\}$$

i.e. the set of strings over T^* derivable from the start symbol.

If G is a CFG, we call $L(G)$ a *context-free language*.

Example: $L(G_{pal})$ is a context-free language.

Theorem 5.7:

$$L(G_{pal}) = \{w \in \{0,1\}^* : w = w^R\}$$

Proof: ($\supseteq$-direction.) Suppose $w = w^R$. We show by induction on $|w|$ that $w \in L(G_{pal})$

Basis: $|w| = 0$, or $|w| = 1$. Then w is $\epsilon, 0$, or 1. Since $P \to \epsilon, P \to 0$, and $P \to 1$ are productions, we conclude that $P \overset{*}{\underset{G}{\Rightarrow}} w$ in all base cases.

Induction: Suppose $|w| \geq 2$. Since $w = w^R$, we have $w = 0x0$, or $w = 1x1$, and $x = x^R$.

If $w = 0x0$ we know from the IH that $P \overset{*}{\Rightarrow} x$. Then

$$P \Rightarrow 0P0 \overset{*}{\Rightarrow} 0x0 = w$$

Thus $w \in L(G_{pal})$.

The case for $w = 1x1$ is similar.

($\subseteq$-direction.) We assume that $w \in L(G_{pal})$ and must show that $w = w^R$.

Since $w \in L(G_{pal})$, we have $P \overset{*}{\Rightarrow} w$.

We do an induction of the length of $\overset{*}{\Rightarrow}$.

Basis: The derivation $P \overset{*}{\Rightarrow} w$ is done in one step.

Then w must be $\epsilon, 0$, or 1, all palindromes.

Induction: Let $n \geq 1$, and suppose the derivation takes $n+1$ steps. Then we must have

$$w = 0x0 \overset{*}{\Leftarrow} 0P0 \Leftarrow P$$

or

$$w = 1x1 \overset{*}{\Leftarrow} 1P1 \Leftarrow P$$

where the second derivation is done in n steps.

By the IH x is a palindrome, and the inductive proof is complete.

Sentential Forms

Let $G = (V, T, P, S)$ be a CFG, and $\alpha \in (V \cup T)^*$. If

$$S \overset{*}{\Rightarrow} \alpha$$

we say that α is a *sentential form*.

If $S \underset{lm}{\Rightarrow} \alpha$ we say that α is a *left-sentential form*, and if $S \underset{rm}{\Rightarrow} \alpha$ we say that α is a *right-sentential form*

Note: $L(G)$ consists of those sentential forms that are in T^*.

Example: Take G from slide 138. Then $E * (I + E)$ is a sentential form since

$$E \Rightarrow E*E \Rightarrow E*(E) \Rightarrow E*(E+E) \Rightarrow E*(I+E)$$

This derivation is neither leftmost, nor rightmost

Example: $a * E$ is a left-sentential form, since

$$E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E$$

Example: $E*(E+E)$ is a right-sentential form, since

$$E \underset{rm}{\Rightarrow} E * E \underset{rm}{\Rightarrow} E * (E) \underset{rm}{\Rightarrow} E * (E + E)$$

Parse Trees

- If $w \in L(G)$, for some CFG, then w has a *parse tree*, which tells us the (syntactic) structure of w

- w could be a program, a SQL-query, an XML-document, etc.

- Parse trees are an alternative representation to derivations and recursive inferences.

- There can be several parse trees for the same string

- Ideally there should be only one parse tree (the "true" structure) for each string, i.e. the language should be *unambiguous*.

- Unfortunately, we cannot always remove the ambiguity.

Constructing Parse Trees

Let $G = (V, T, P, S)$ be a CFG. A tree is a *parse tree* for G if:

1. Each interior node is labelled by a variable in V.

2. Each leaf is labelled by a symbol in $V \cup T \cup \{\epsilon\}$. Any ϵ-labelled leaf is the only child of its parent.

3. If an interior node is lablelled A, and its children (from left to right) labelled

$$X_1, X_2, \ldots, X_k,$$

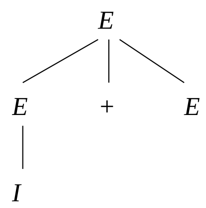
then $A \rightarrow X_1 X_2 \ldots X_k \in P$.

Example: In the grammar

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$

$\vdots$

the following is a parse tree:

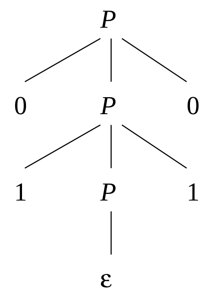


This parse tree shows the derivation $E \stackrel{*}{\Rightarrow} I + E$

Example: In the grammar

1. $P \rightarrow \epsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0P0$
5. $P \rightarrow 1P1$

the following is a parse tree:



It shows the derivation of $P \stackrel{*}{\Rightarrow} 0110$.

The Yield of a Parse Tree

The *yield* of a parse tree is the string of leaves from left to right.

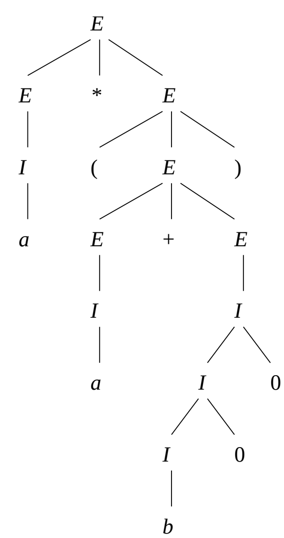
Important are those parse trees where:

1. The yield is a terminal string.

2. The root is labelled by the start symbol

We shall see the the set of yields of these important parse trees is the language of the grammar.

Example: Below is an important parse tree



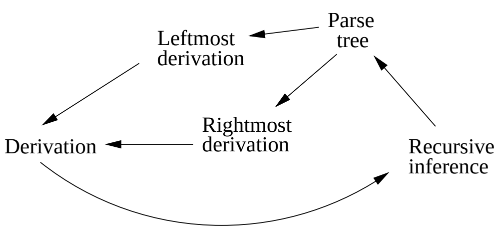
The yield is $a * (a + b00)$.

Compare the parse tree with the derivation on slide 141.

Let $G = (V, T, P, S)$ be a CFG, and $A \in V$. We are going to show that the following are equivalent:

1. We can determine by recursive inference that w is in the language of A
2. $A \stackrel{*}{\Rightarrow} w$
3. $A \underset{lm}{\stackrel{*}{\Rightarrow}} w$, and $A \underset{rm}{\stackrel{*}{\Rightarrow}} w$
4. There is a parse tree of G with root A and yield w.

To prove the equivalences, we use the following plan.

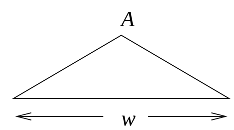


From Inferences to Trees

Theorem 5.12: Let $G = (V, T, P, S)$ be a CFG, and suppose we can show w to be in the language of a variable A. Then there is a parse tree for G with root A and yield w.

Proof: We do an induction of the length of the inference.

Basis: One step. Then we must have used a production $A \to w$. The desired parse tree is then



Induction: w is inferred in $n+1$ steps. Suppose the last step was based on a production

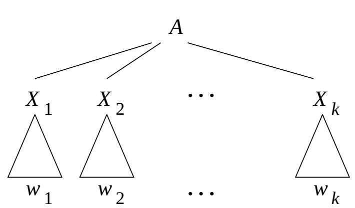
$$A \to X_1 X_2 \cdots X_k,$$

where $X_i \in V \cup T$. We break w up as

$$w_1 w_2 \cdots w_k,$$

where $w_i = X_i$, when $X_i \in T$, and when $X_i \in V$, then w_i was previously inferred being in X_i, in at most n steps.

By the IH there are parse trees i with root X_i and yield w_i. Then the following is a parse tree for G with root A and yield w:



From trees to derivations

We'll show how to construct a leftmost derivation from a parse tree.

Example: In the grammar of slide 6 there clearly is a derivation

$$E \Rightarrow I \Rightarrow Ib \Rightarrow ab.$$

Then, for any α and β there is a derivation

$$\alpha E \beta \Rightarrow \alpha I \beta \Rightarrow \alpha Ib \beta \Rightarrow \alpha ab \beta.$$

For example, suppose we have a derivation

$$E \Rightarrow E + E \Rightarrow E + (E).$$

The we can choose $\alpha = E + ($ and $\beta =)$ and continue the derivation as

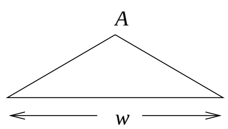
$$E + (E) \Rightarrow E + (I) \Rightarrow E + (Ib) \Rightarrow E + (ab).$$

This is why CFG's are called context-free.

Theorem 5.14: Let $G = (V, T, P, S)$ be a CFG, and suppose there is a parse tree with root labelled A and yield w. Then $A \underset{lm}{\overset{*}{\Rightarrow}} w$ in G.

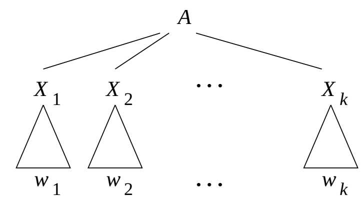
Proof: We do an induction on the height of the parse tree.

Basis: Height is 1. The tree must look like



Consequently $A \to w \in P$, and $A \underset{lm}{\Rightarrow} w$.

Induction: Height is $n + 1$. The tree must look like



Then $w = w_1 w_2 \cdots w_k$, where

1. If $X_i \in T$, then $w_i = X_i$.

2. If $X_i \in V$, then $X_i \underset{lm}{\overset{*}{\Rightarrow}} w_i$ in G by the IH.

Now we construct $A \underset{lm}{\overset{*}{\Rightarrow}} w$ by an (inner) induction by showing that

$$\forall i : A \underset{lm}{\overset{*}{\Rightarrow}} w_1 w_2 \cdots w_i X_{i+1} X_{i+2} \cdots X_k.$$

Basis: Let $i = 0$. We already know that $A \underset{lm}{\Rightarrow} X_1 X_{i+2} \cdots X_k$.

Induction: Make the IH that

$$A \underset{lm}{\overset{*}{\Rightarrow}} w_1 w_2 \cdots w_{i-1} X_i X_{i+1} \cdots X_k.$$

(*Case 1:*) $X_i \in T$. Do nothing, since $X_i = w_i$ gives us

$$A \underset{lm}{\overset{*}{\Rightarrow}} w_1 w_2 \cdots w_i X_{i+1} \cdots X_k.$$

(*Case 2:*) $X_i \in V$. By the IH there is a derivation $X_i \underset{lm}{\Rightarrow} \alpha_1 \underset{lm}{\Rightarrow} \alpha_2 \underset{lm}{\Rightarrow} \cdots \underset{lm}{\Rightarrow} w_i$. By the contexfree property of derivations we can proceed with

$$A \underset{lm}{\overset{*}{\Rightarrow}}$$

$$w_1 w_2 \cdots w_{i-1} X_i X_{i+1} \cdots X_k \underset{lm}{\Rightarrow}$$

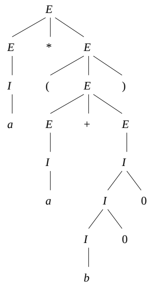
$$w_1 w_2 \cdots w_{i-1} \alpha_1 X_{i+1} \cdots X_k \underset{lm}{\Rightarrow}$$

$$w_1 w_2 \cdots w_{i-1} \alpha_2 X_{i+1} \cdots X_k \underset{lm}{\Rightarrow}$$

$$\cdots$$

$$w_1 w_2 \cdots w_{i-1} w_i X_{i+1} \cdots X_k$$

Example: Let's construct the leftmost derivation for the tree



Suppose we have inductively constructed the leftmost derivation

$$E \underset{lm}{\Rightarrow} I \underset{lm}{\Rightarrow} a$$

corresponding to the leftmost subtree, and the leftmost derivation

$$E \underset{lm}{\Rightarrow} (E) \underset{lm}{\Rightarrow} (E + E) \underset{lm}{\Rightarrow} (I + E) \underset{lm}{\Rightarrow} (a + E) \underset{lm}{\Rightarrow}$$
$$(a + I) \underset{lm}{\Rightarrow} (a + I0) \underset{lm}{\Rightarrow} (a + I00) \underset{lm}{\Rightarrow} (a + b00)$$

corresponding to the righmost subtree.

For the derivation corresponding to the whole tree we start with $E \underset{lm}{\Rightarrow} E * E$ and expand the first E with the first derivation and the second E with the second derivation:

$$\begin{aligned}
&E \underset{lm}{\Rightarrow}\\
&E * E \underset{lm}{\Rightarrow}\\
&I * E \underset{lm}{\Rightarrow}\\
&a * E \underset{lm}{\Rightarrow}\\
&a * (E) \underset{lm}{\Rightarrow}\\
&a * (E + E) \underset{lm}{\Rightarrow}\\
&a * (I + E) \underset{lm}{\Rightarrow}\\
&a * (a + E) \underset{lm}{\Rightarrow}\\
&a * (a + I) \underset{lm}{\Rightarrow}\\
&a * (a + I0) \underset{lm}{\Rightarrow}\\
&a * (a + I00) \underset{lm}{\Rightarrow}\\
&a * (a + b00)
\end{aligned}$$

From Derivations to Recursive Inferences

Observation: Suppose that $A \Rightarrow X_1 X_2 \cdots X_k \stackrel{*}{\Rightarrow} w$. Then $w = w_1 w_2 \cdots w_k$, where $X_i \stackrel{*}{\Rightarrow} w_i$

The factor w_i can be extracted from $A \stackrel{*}{\Rightarrow} w$ by looking at the expansion of X_i only.

Example: $E \Rightarrow a * b + a$, and

$$E \Rightarrow \underbrace{E}_{X_1} \underbrace{*}_{X_2} \underbrace{E}_{X_3} \underbrace{+}_{X_4} \underbrace{E}_{X_5}$$

We have

$$E \Rightarrow E * E \Rightarrow E * E + E \Rightarrow I * E + E \Rightarrow I * I + E \Rightarrow$$
$$I * I + I \Rightarrow a * I + I \Rightarrow a * b + I \Rightarrow a * b + a$$

By looking at the expansion of $X_3 = E$ only, we can extract

$$E \Rightarrow I \Rightarrow b.$$

Theorem 5.18: Let $G = (V, T, P, S)$ be a CFG. Suppose $A \underset{G}{\stackrel{*}{\Rightarrow}} w$, and that w is a string of terminals. Then we can infer that w is in the language of variable A.

Proof: We do an induction on the length of the derivation $A \underset{G}{\stackrel{*}{\Rightarrow}} w$.

Basis: One step. If $A \underset{G}{\Rightarrow} w$ there must be a production $A \to w$ in P. The we can infer that w is in the language of A.

Induction: Suppose $A \overset{*}{\underset{G}{\Rightarrow}} w$ in $n+1$ steps. Write the derivation as

$$A \underset{G}{\Rightarrow} X_1 X_2 \cdots X_k \overset{*}{\underset{G}{\Rightarrow}} w$$

The as noted on the previous slide we can break w as $w_1 w_2 \cdots w_k$ where $X_i \overset{*}{\underset{G}{\Rightarrow}} w_i$. Furthermore, $X_i \overset{*}{\underset{G}{\Rightarrow}} w_i$ can use at most n steps.

Now we have a production $A \to X_1 X_2 \cdots X_k$, and we know by the IH that we can infer w_i to be in the language of X_i.

Therefore we can infer $w_1 w_2 \cdots w_k$ to be in the language of A.

Ambiguity in Grammars and Languages

In the grammar

1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$

$\cdots$

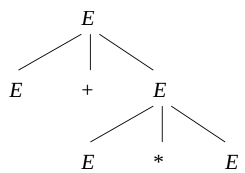
the sentential form $E + E * E$ has two derivations:

$$E \Rightarrow E + E \Rightarrow E + E * E$$

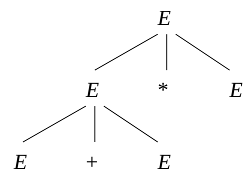
and

$$E \Rightarrow E * E \Rightarrow E + E * E$$

This gives us two parse trees:



(a)



(b)

The mere existence of several *derivations* is not dangerous, it is the existence of several parse trees that ruins a grammar.

Example: In the same grammar

5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

the string $a + b$ has several derivations, e.g.

$$E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b$$

and

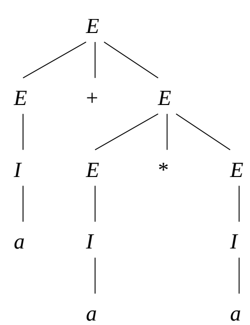
$$E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b$$

However, their parse trees are the same, and the structure of $a + b$ is unambiguous.

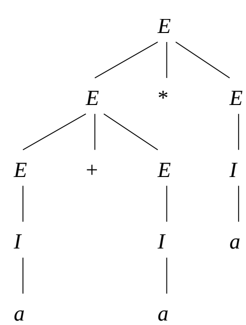
Definition: Let $G = (V, T, P, S)$ be a CFG. We say that G is *ambiguous* is there is a string in T^* that has more than one parse tree.

If every string in $L(G)$ has at most one parse tree, G is said to be *unambiguous*.

Example: The terminal string $a + a * a$ has two parse trees:



(a)



(b)

Removing Ambiguity From Grammars

Good news: Sometimes we can remove ambiguity "by hand"

Bad news: There is no algorithm to do it

More bad news: Some CFL's have only ambiguous CFG's

We are studying the grammar

$$E \to I \mid E + E \mid E * E \mid (E)$$
$$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

There are two problems:

1. There is no precedence between * and +

2. There is no grouping of sequences of operators, e.g. is $E + E + E$ meant to be $E + (E + E)$ or $(E + E) + E$.

Solution: We introduce more variables, each representing expressions of same "binding strength."

1. A *factor* is an expresson that cannot be broken apart by an adjacent * or +. Our factors are

   (a) Identifiers

   (b) A parenthesized expression.

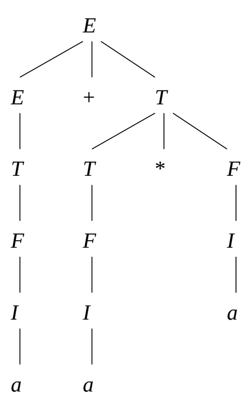
2. A *term* is an expresson that cannot be broken by +. For instance $a * b$ can be broken by $a1*$ or $*a1$. It cannot be broken by +, since e.g. $a1 + a * b$ is (by precedence rules) same as $a1 + (a * b)$, and $a * b + a1$ is same as $(a * b) + a1$.

3. The rest are *expressions*, i.e. they can be broken apart with * or +.

We'll let F stand for factors, T for terms, and E for expressions. Consider the following grammar:

1. $I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
2. $F \to I \mid (E)$
3. $T \to F \mid T * F$
4. $E \to T \mid E + T$

Now the only parse tree for $a + a * a$ will be



Why is the new grammar unambiguous?

Intuitive explanation:

- A factor is either an identifier or (E), for some expression E.

- The only parse tree for a sequence

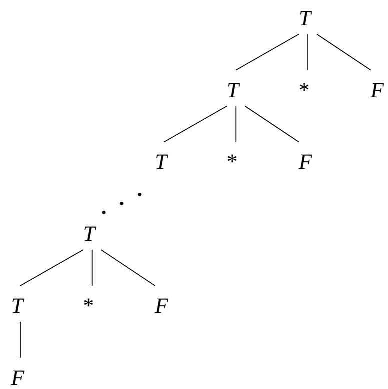
$$f_1 * f_2 * \cdots * f_{n-1} * f_n$$

of factors is the one that gives $f_1 * f_2 * \cdots * f_{n-1}$ as a term and f_n as a factor, as in the parse tree on the next slide.

- An expression is a sequence

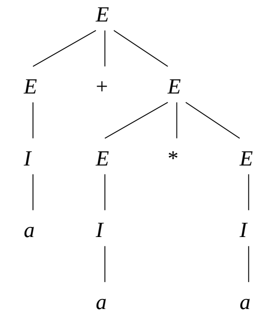
$$t_1 + t_2 + \cdots + t_{n-1} + t_n$$

of terms t_i. It can only be parsed with $t_1 + t_2 + \cdots + t_{n-1}$ as an expression and t_n as a term.

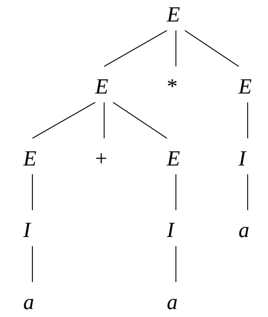


Leftmost derivations and Ambiguity

The two parse trees for $a + a * a$



(a)



(b)

give rise to two derivations:

$$E \underset{lm}{\Rightarrow} E + E \underset{lm}{\Rightarrow} I + E \underset{lm}{\Rightarrow} a + E \underset{lm}{\Rightarrow} a + E * E$$
$$\underset{lm}{\Rightarrow} a + I * E \underset{lm}{\Rightarrow} a + a * E \underset{lm}{\Rightarrow} a + a * I \underset{lm}{\Rightarrow} a + a * a$$

and

$$E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} E + E * E \underset{lm}{\Rightarrow} I + E * E \underset{lm}{\Rightarrow} a + E * E$$
$$\underset{lm}{\Rightarrow} a + I * E \underset{lm}{\Rightarrow} a + a * E \underset{lm}{\Rightarrow} a + a * I \underset{lm}{\Rightarrow} a + a * a$$

In General:

- One parse tree, but many derivations

- Many *leftmost* derivation implies many parse trees.

- Many *rightmost* derivation implies many parse trees.

Theorem 5.29: For any CFG G, a terminal string w has two distinct parse trees if and only if w has two distinct leftmost derivations from the start symbol.

Sketch of Proof: *(Only If.)* If the two parse trees differ, they have a node a which different productions, say $A \rightarrow X_1X_2 \cdots X_k$ and $B \rightarrow Y_1Y_2 \cdots Y_m$. The corresponding leftmost derivations will use derivations based on these two different productions and will thus be distinct.

(If.) Let's look at how we construct a parse tree from a leftmost derivation. It should now be clear that two distinct derivations gives rise to two different parse trees.

Inherent Ambiguity

A CFL L is *inherently ambiguous* if *all* grammars for L are ambiguous.

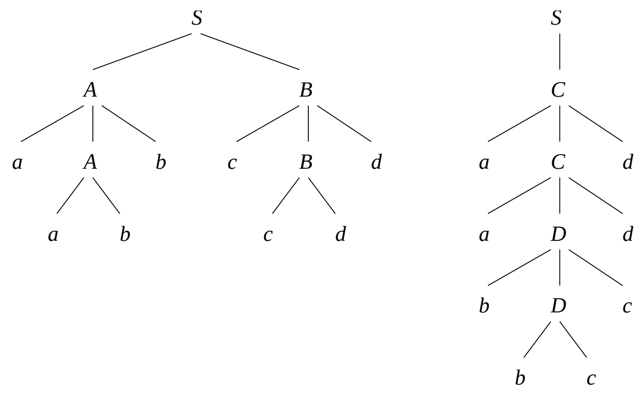
Example: Consider $L =$

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

A grammar for L is

$$
\begin{aligned}
S &\rightarrow AB \mid C \\
A &\rightarrow aAb \mid ab \\
B &\rightarrow cBd \mid cd \\
C &\rightarrow aCd \mid aDd \\
D &\rightarrow bDc \mid bc
\end{aligned}
$$

Let's look at parsing the string $aabbccdd$.



(a) (b)

From this we see that there are two leftmost derivations:

$$S \underset{lm}{\Rightarrow} AB \underset{lm}{\Rightarrow} aAbB \underset{lm}{\Rightarrow} aabbB \underset{lm}{\Rightarrow} aabbcBd \underset{lm}{\Rightarrow} aabbccdd$$

and

$$S \underset{lm}{\Rightarrow} C \underset{lm}{\Rightarrow} aCd \underset{lm}{\Rightarrow} aaDdd \underset{lm}{\Rightarrow} aabDcdd \underset{lm}{\Rightarrow} aabbccdd$$

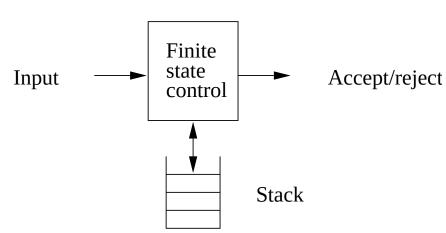
It can be shown that *every* grammar for L behaves like the one above. The language L is inherently ambiguous.

Pushdown Automata

A pushdown automata (PDA) is essentially an ϵ-NFA with a stack.

On a transition the PDA:

1. Consumes an input symbol.
2. Goes to a new state (or stays in the old).
3. Replaces the top of the stack by any string (does nothing, pops the stack, or pushes a string onto the stack)



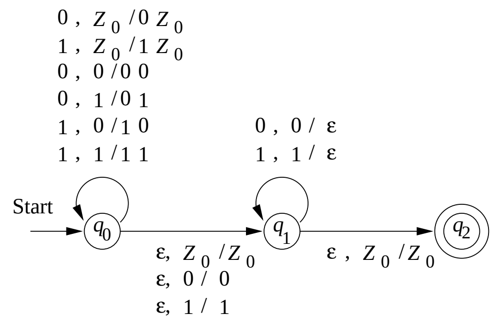
Example: Let's consider

$$L_{wwr} = \{ww^R : w \in \{0,1\}^*\},$$

with "grammar" $P \to 0P0,\ P \to 1P1,\ P \to \epsilon$. A PDA for L_{wwr} has tree states, and operates as follows:

1. Guess that you are reading w. Stay in state 0, and push the input symbol onto the stack.
2. Guess that you're in the middle of ww^R. Go spontanteously to state 1.
3. You're now reading the head of w^R. Compare it to the top of the stack. If they match, pop the stack, and remain in state 1. If they don't match, go to sleep.
4. If the stack is empty, go to state 2 and accept.

The PDA for L_{wwr} as a transition diagram:



PDA formally

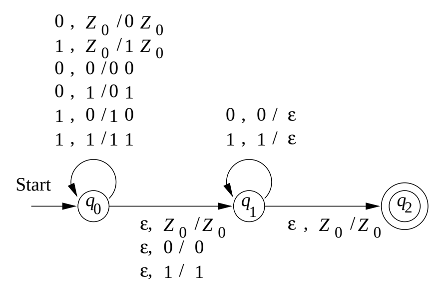
A PDA is a seven-tuple:

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

where

- Q is a finite set of states,
- Σ is a finite *input alphabet,*
- Γ is a finite *stack alphabet,*
- $\delta : Q \times \Sigma \cup \{\epsilon\} \times \Gamma \to 2^{Q \times \Gamma^*}$ is the *transition function,*
- q_0 is the *start state,*
- $Z_0 \in \Gamma$ is the *start symbol* for the stack, and
- $F \subseteq Q$ is the set of *accepting states.*

Example: The PDA



is actually the seven-tuple

$$P = (\{q_0, q_1, q_2\}, \{0,1\}, \{0,1,Z_0\}, \delta, q_0, Z_0, \{q_2\}),$$

where δ is given by the following table (set brackets missing):

	$0, Z_0$	$1, Z_0$	0,0	0,1	1,0	1,1	ϵ, Z_0	$\epsilon, 0$	$\epsilon, 1$
$\to q_0$	$q_0, 0Z_0$	$q_0, 1Z_0$	$q_0, 00$	$q_0, 01$	$q_0, 10$	$q_0, 11$	q_1, Z_0	$q_1, 0$	$q_1, 1$
q_1			q_1, ϵ			q_1, ϵ	q_2, Z_0		
$\star q_2$									

Instantaneous Descriptions

A PDA goes from configuration to configuration when consuming input.

To reason about PDA computation, we use *instantaneous descriptions* of the PDA. An ID is a triple

$$(q, w, \gamma)$$

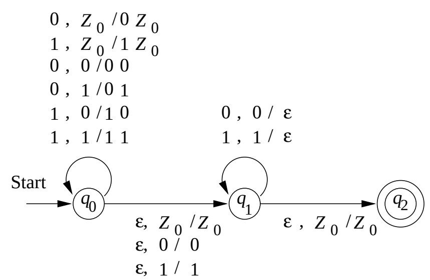
where q is the state, w the remaining input, and γ the stack contents.

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. Then $\forall w \in \Sigma^*, \beta \in \Gamma^*$:

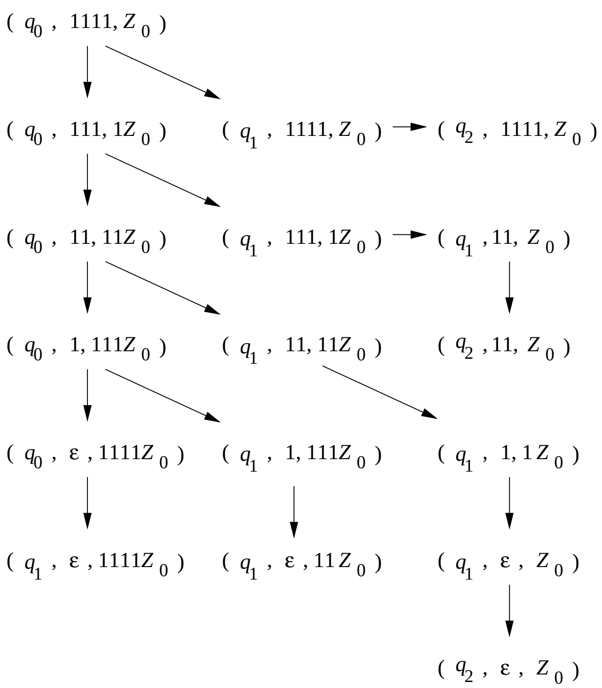
$$(p, \alpha) \in \delta(q, a, X) \Rightarrow (q, aw, X\beta) \vdash (p, w, \alpha\beta).$$

We define $\overset{*}{\vdash}$ to be the reflexive-transitive closure of $\vdash$.

Example: On input 1111 the PDA



has the following computation sequences:



The following properties hold:

1. If an ID sequence is a legal computation for a PDA, then so is the sequence obtained by adding an additional string at the end of component number two.
2. If an ID sequence is a legal computation for a PDA, then so is the sequence obtained by adding an additional string at the bottom of component number three.
3. If an ID sequence is a legal computation for a PDA, and some tail of the input is not consumed, then removing this tail from all ID's result in a legal computation sequence.

Theorem 6.5: $\forall w \in \Sigma^*, \beta \in \Gamma^*$:

$$(q, x, \alpha) \overset{*}{\vdash} (p, y, \beta) \Rightarrow (q, xw, \alpha\gamma) \overset{*}{\vdash} (p, yw, \beta\gamma).$$

Proof: Induction on the length of the sequence to the left.

Note: If $\gamma = \epsilon$ we have proerty 1, and if $w = \epsilon$ we have property 2.

Note2: The reverse of the theorem is false.

For property 3 we have

Theorem 6.6:

$$(q, xw, \alpha) \overset{*}{\vdash} (p, yw, \beta) \Rightarrow (q, x, \alpha) \overset{*}{\vdash} (p, y, \beta).$$

Acceptance by final state

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

$$L(P) = \{w : (q_0, w, Z_0) \overset{*}{\vdash} (q, \epsilon, \alpha),\ q \in F\}.$$

Example: The PDA on slide 183 accepts exactly L_{wwr}.

Let P be the machine. We prove that $L(P) = L_{wwr}$.

($\supseteq$-*direction.*) Let $x \in L_{wwr}$. Then $x = ww^R$, and the following is a legal computation sequence

$$(q_0, ww^R, Z_0) \overset{*}{\vdash} (q_0, w^R, w^R Z_0) \vdash (q_1, w^R, w^R Z_0) \overset{*}{\vdash}$$

$$(q_1, \epsilon, Z_0) \vdash (q_2, \epsilon, Z_0).$$

($\subseteq$-*direction.*)

Observe that the only way the PDA can enter q_2 is if it is in state q_1 with an empty stack.

Thus it is sufficient to show that if $(q_0, x, Z_0) \overset{*}{\vdash} (q_1, \epsilon, Z_0)$ then $x = ww^R$, for some word w.

We'll show by induction on $|x|$ that

$$(q_0, x, \alpha) \overset{*}{\vdash} (q_1, \epsilon, \alpha) \quad \Rightarrow \quad x = ww^R.$$

Basis: If $x = \epsilon$ then x is a palindrome.

Induction: Suppose $x = a_1 a_2 \ldots a_n$, where $n > 0$, and the IH holds for shorter strings.

Ther are two moves for the PDA from ID (q_0, x, α):

Move 1: The spontaneous $(q_0, x, \alpha) \vdash (q_1, x, \alpha)$. Now $(q_1, x, \alpha) \overset{*}{\vdash} (q_1, \epsilon, \beta)$ implies that $|\beta| < |\alpha|$, which implies $\beta \neq \alpha$.

Move 2: Loop and push $(q_0, a_1 a_2 \ldots a_n, \alpha) \vdash (q_0, a_2 \ldots a_n, a_1\alpha)$.

In this case there is a sequence

$(q_0, a_1 a_2 \ldots a_n, \alpha) \vdash (q_0, a_2 \ldots a_n, a_1\alpha) \vdash \ldots \vdash (q_1, a_n, a_1\alpha) \vdash (q_1, \epsilon, \alpha)$.

Thus $a_1 = a_n$ and

$$(q_0, a_2 \ldots a_n, a_1\alpha) \overset{*}{\vdash} (q_1, a_n, a_1\alpha).$$

By Theorem 6.6 we can remove a_n. Therefore

$$(q_0, a_2 \ldots a_{n-1}, a_1\alpha) \overset{*}{\vdash} (q_1, \epsilon, a_1\alpha).$$

Then, by the IH $a_2 \ldots a_{n-1} = yy^R$. Then $x = a_1 yy^R a_n$ is a palindrome.

Acceptance by Empty Stack

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

$$N(P) = \{w : (q_0, w, Z_0) \overset{*}{\vdash} (q, \epsilon, \epsilon)\}.$$

Note: q can be any state.

Question: How to modify the palindrome-PDA to accept by empty stack?

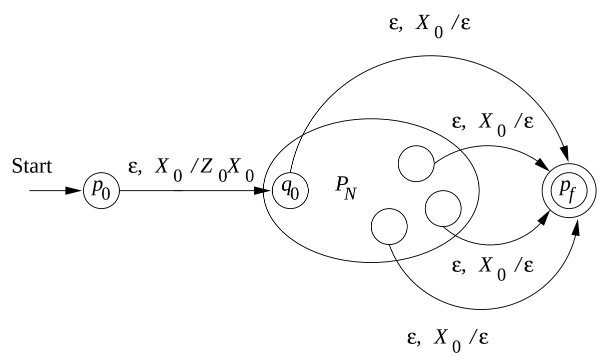
From Empty Stack to Final State

Theorem 6.9: If $L = N(P_N)$ for some PDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$, then $\exists$ PDA P_F, such that $L = L(P_F)$.

Proof: Let

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

where $\delta_F(p_0, \epsilon, X_0) = \{(q_0, Z_0 X_0)\}$, and for all $q \in Q, a \in \Sigma \cup \{\epsilon\}, Y \in \Gamma : \delta_F(q, a, Y) = \delta_N(q, a, Y)$, and in addition $(p_f, \epsilon) \in \delta_F(q, \epsilon, X_0)$.



We have to show that $L(P_F) = N(P_N)$.

($\supseteq$*direction.*) Let $w \in N(P_N)$. Then

$$(q_0, w, Z_0) \overset{*}{\underset{N}{\vdash}} (q, \epsilon, \epsilon),$$

for some q. From Theorem 6.5 we get

$$(q_0, w, Z_0 X_0) \overset{*}{\underset{N}{\vdash}} (q, \epsilon, X_0).$$

Since $\delta_N \subset \delta_F$ we have

$$(q_0, w, Z_0 X_0) \overset{*}{\underset{F}{\vdash}} (q, \epsilon, X_0).$$

We conclude that

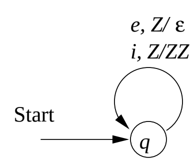
$$(p_0, w, X_0) \underset{F}{\vdash} (q_0, w, Z_0 X_0) \overset{*}{\underset{F}{\vdash}} (q, \epsilon, X_0) \underset{F}{\vdash} (p_f, \epsilon, \epsilon).$$

($\subseteq$*direction.*) By inspecting the diagram.

Let's design P_N for for cathing errors in strings meant to be in the *if-else*-grammar G

$$S \to \epsilon | SS | iS | iSe.$$

Here e.g. $\{ieie, iie, iiee\} \subseteq G$, and e.g. $\{ei, ieeii\} \cap G = \emptyset$. The diagram for P_N is



Formally,

$$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z),$$

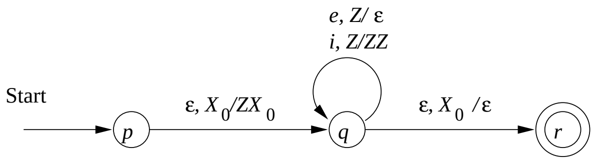
where $\delta_N(q, i, Z) = \{(q, ZZ)\}$,
and $\delta_N(q, e, Z) = \{(q, \epsilon)\}$.

From P_N we can construct

$$P_F = (\{p, q, r\}, \{i, e\}, \{Z, X_0\}, \delta_F, p, X_0, \{r\}),$$

where
$\delta_F(p, \epsilon, X_0) = \{(q, ZX_0)\}$,
$\delta_F(q, i, Z) = \delta_N(q, i, Z) = \{(q, ZZ)\}$,
$\delta_F(q, e, Z) = \delta_N(q, e, Z) = \{(q, \epsilon)\}$, and
$\delta_F(q, \epsilon, X_0) = \{(r, \epsilon)\}$

The diagram for P_F is



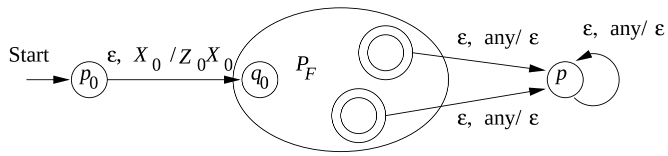
From Final State to Empty Stack

Theorem 6.11: Let $L = L(P_F)$, for some PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$. Then $\exists$ PDA P_N, such that $L = N(P_N)$.

Proof: Let

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

where $\delta_N(p_0, \epsilon, X_0) = \{(q_0, Z_0X_0)\}$, $\delta_N(p, \epsilon, Y) = \{(p, \epsilon)\}$, for $Y \in \Gamma \cup \{X_0\}$, and for all $q \in Q$, $a \in \Sigma \cup \{\epsilon\}, Y \in \Gamma$: $\delta_N(q, a, Y) = \delta_F(q, a, Y)$, and in addition $\forall q \in F$, and $Y \in \Gamma \cup \{X_0\}$: $(p, \epsilon) \in \delta_N(q, \epsilon, Y)$.



We have to show that $N(P_N) = L(P_F)$.

($\subseteq$-*direction*.) By inspecting the diagram.

($\supseteq$-*direction*.) Let $w \in L(P_F)$. Then

$$(q_0, w, Z_0) \overset{*}{\underset{F}{\vdash}} (q, \epsilon, \alpha),$$

for some $q \in F, \alpha \in \Gamma^*$. Since $\delta_F \subset \delta_N$, and Theorem 6.5 says that X_0 can be slid under the stack, we get

$$(q_0, w, Z_0X_0) \overset{*}{\underset{N}{\vdash}} (q, \epsilon, \alpha X_0).$$

Then P_N can compute:

$$(p_0, w, X_0) \underset{N}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{N}{\vdash}} (q, \epsilon, \alpha X_0) \overset{*}{\underset{N}{\vdash}} (p, \epsilon, \epsilon).$$

Equivalence of PDA's and CFG's

A language is

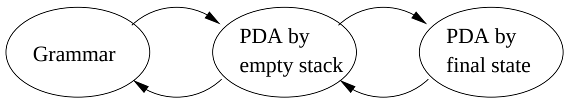
generated by a CFG

if and only if it is

accepted by a PDA by empty stack

if and only if it is

accepted by a PDA by final state



We already know how to go between null stack and final state.

From CFG's to PDA's

Given G, we construct a PDA that simulates $\overset{*}{\underset{lm}{\Rightarrow}}$.

We write left-sentential forms as

$$xA\alpha$$

where A is the leftmost variable in the form. For instance,

$$\underbrace{(a+}_{x}\underbrace{\underbrace{E}_{A}\underbrace{)}_{\alpha}}_{\text{tail}}$$

Let $xA\alpha \underset{lm}{\Rightarrow} x\beta\alpha$. This corresponds to the PDA first having consumed x and having $A\alpha$ on the stack, and then on ϵ it pops A and pushes β.

More fomally, let y, s.t. $w = xy$. Then the PDA goes non-deterministically from configuration $(q, y, A\alpha)$ to configuration $(q, y, \beta\alpha)$.

At $(q, y, \beta\alpha)$ the PDA behaves as before, unless there are terminals in the prefix of β. In that case, the PDA pops them, provided it can consume matching input.

If all guesses are right, the PDA ends up with empty stack and input.

Formally, let $G = (V, T, Q, S)$ be a CFG. Define P_G as

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

where

$$\delta(q, \epsilon, A) = \{(q, \beta) : A \to \beta \ \in \ Q\},$$

for $A \in V$, and

$$\delta(q, a, a) = \{(q, \epsilon)\},$$

for $a \in T$.

Example: On blackboard in class.

Theorem 6.13: $N(P_G) = L(G)$.

Proof:

($\supseteq$-*direction*.) Let $w \in L(G)$. Then

$$S = \gamma_1 \underset{lm}{\Rightarrow} \gamma_2 \underset{lm}{\Rightarrow} \cdots \underset{lm}{\Rightarrow} \gamma_n = w$$

Let $\gamma_i = x_i\alpha_i$. We show by induction on i that if

$$S \overset{*}{\underset{lm}{\Rightarrow}} \gamma_i,$$

then

$$(q, w, S) \overset{*}{\vdash} (q, y_i, \alpha_i),$$

where $w = x_i y_i$.

Basis: For $i = 1, \gamma_1 = S$. Thus $x_1 = \epsilon$, and $y_1 = w$. Clearly $(q, w, S) \overset{*}{\vdash} (q, w, S)$.

Induction: IH is $(q, w, S) \overset{*}{\vdash} (q, y_i, \alpha_i)$. We have to show that

$$(q, y_i, \alpha_i) \vdash (q, y_{i+1}, \alpha_{i+1})$$

Now α_i begins with a variable A, and we have the form

$$\underbrace{x_i A\chi}_{\gamma_i} \underset{lm}{\Rightarrow} \underbrace{x_{i+1}\beta\chi}_{\gamma_{i+1}}$$

By IH $A\chi$ is on the stack, and y_i is unconsumed. From the construction of P_G is follows that we can make the move

$$(q, y_i, \chi) \vdash (q, y_i, \beta\chi).$$

If β has a prefix of terminals, we can pop them with matching terminals in a prefix of y_i, ending up in configuration $(q, y_{i+1}, \alpha_{i+1})$, where $\alpha_{i+1} = \beta\chi$, which is the tail of the sentential $x_i\beta\chi = \gamma_{i+1}$.

Finally, since $\gamma_n = w$, we have $\alpha_n = \epsilon$, and $y_n = \epsilon$, and thus $(q, w, S) \overset{*}{\vdash} (q, \epsilon, \epsilon)$, i.e. $w \in N(P_G)$

($\subseteq$-*direction*.) We shall show by an induction on the length of $\overset{*}{\vdash}$, that

($\clubsuit$) If $(q, x, A) \overset{*}{\vdash} (q, \epsilon, \epsilon)$, then $A \overset{*}{\Rightarrow} x$.

Basis: Length 1. Then it must be that $A \to \epsilon$ is in G, and we have $(q, \epsilon) \in \delta(q, \epsilon, A)$. Thus $A \overset{*}{\Rightarrow} \epsilon$.

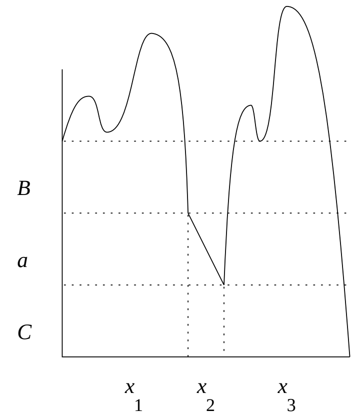
Induction: Length is $n > 1$, and the IH holds for lengths $< n$.

Since A is a variable, we must have

$$(q, x, A) \vdash (q, x, Y_1Y_2 \cdots Y_k) \vdash \cdots \vdash (q, \epsilon, \epsilon)$$

where $A \to Y_1Y_2 \cdots Y_k$ is in G.

We can now write x as $x_1x_2 \cdots x_n$, according to the figure below, where $Y_1 = B, Y_2 = a$, and $Y_3 = C$.



Now we can conclude that

$$(q, x_ix_{i+1} \cdots x_k, Y_i) \overset{*}{\vdash} (q, x_{i+1} \cdots x_k, \epsilon)$$

is less than n steps, for all $i \in \{1, \ldots, k\}$. If Y_i is a variable we have by the IH and Theorem 6.6 that

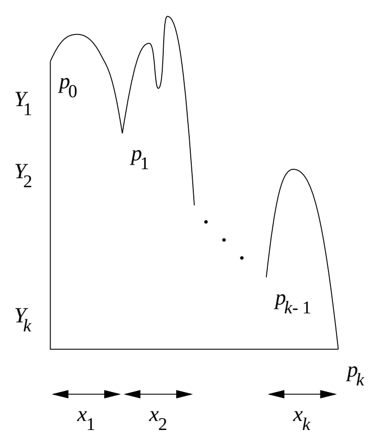
$$Y_i \overset{*}{\Rightarrow} x_i$$

If Y_i is a terminal, we have $|x_i| = 1$, and $Y_i = x_i$. Thus $Y_i \overset{*}{\Rightarrow} x_i$ by the reflexivity of $\overset{*}{\Rightarrow}$.

The claim of the theorem now follows by choosing $A = S$, and $x = w$. Suppose $w \in N(P)$. Then $(q, w, S) \overset{*}{\vdash} (q, \epsilon, \epsilon)$, and by ($\clubsuit$), we have $S \overset{*}{\Rightarrow} w$, meaning $w \in L(G)$.

From PDA's to CFG's

Let's look at how a PDA can consume $x = x_1x_2 \ldots x_k$ and empty the stack.

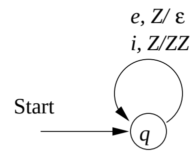


We shall define a grammar with variables of the form $[p_{i-1}Y_ip_i]$ representing going from p_{i-1} to p_i with net effect of popping Y_i.

Formally, let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ be a PDA. Define $G = (V, \Sigma, R, S)$, where

$$
\begin{aligned}
V &= \{[pXq] : \{p, q\} \subseteq Q, X \in \Gamma\} \cup \{S\} \\
R &= \{S \to [q_0 Z_0 p] : p \in Q\} \cup \\
&\quad \{[\mathbf{q} X r_k] \to a[\mathbf{r} Y_1 r_1] \cdots [r_{k-1} Y_k r_k] : \\
&\qquad\qquad a \in \Sigma \cup \{\epsilon\}, \\
&\qquad\qquad \{r_1, \ldots, r_k\} \subseteq Q, \\
&\qquad\qquad (\mathbf{r}, Y_1 Y_2 \cdots Y_k) \in \delta(\mathbf{q}, a, X)\}
\end{aligned}
$$

Example: Let's convert



$$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z),$$

where $\delta_N(q, i, Z) = \{(q, ZZ)\}$,
and $\delta_N(q, e, Z) = \{(q, \epsilon)\}$ to a grammar

$$G = (V, \{i, e\}, R, S),$$

where $V = \{[qZq], S\}$, and
$R = \{S \to [qZq], [qZq] \to i[qZq][qZq], [qZq] \to e\}$.

If we replace $[qZq]$ by A we get the productions $S \to A$ and $A \to iAA|e$.

Example: Let $P = (\{p, q\}, \{0, 1\}, \{X, Z_0\}, \delta, q, Z_0)$, where δ is given by

1. $\delta(q, 1, Z_0) = \{(q, XZ_0)\}$
2. $\delta(q, 1, X) = \{(q, XX)\}$
3. $\delta(q, 0, X) = \{(p, X)\}$
4. $\delta(q, \epsilon, X) = \{(q, \epsilon)\}$
5. $\delta(p, 1, X) = \{(p, \epsilon)\}$
6. $\delta(p, 0, Z_0) = \{(q, Z_0)\}$

to a CFG.

We get $G = (V, \{0, 1\}, R, S)$, where

$$V = \{[pXp], [pXq], [pZ_0p], [pZ_0q], S\}$$

and the productions in R are

$S \to [qZ_0q] | [qZ_0p]$

From rule (1):

$[qZ_0q] \to 1[qXq][qZ_0q]$
$[qZ_0q] \to 1[qXp][pZ_0q]$
$[qZ_0p] \to 1[qXq][qZ_0p]$
$[qZ_0p] \to 1[qXp][pZ_0p]$

From rule (2):

$[qXq] \to 1[qXq][qXq]$
$[qXq] \to 1[qXp][pXq]$
$[qXp] \to 1[qXq][qXp]$
$[qXp] \to 1[qXp][pXp]$

From rule (3):

$[qXq] \to 0[pXq]$
$[qXp] \to 0[pXp]$

From rule (4):

$[qXq] \to \epsilon$

From rule (5):

$[pXp] \to 1$

From rule (6):

$[pZ_0q] \to 0[qZ_0q]$
$[pZ_0p] \to 0[qZ_0p]$

Theorem 6.14: Let G be constructed from a PDA P as above. Then $L(G) = N(P)$

Proof:

($\supseteq$-*direction.*) We shall show by an induction on the length of the sequence $\overset{*}{\vdash}$ that

(♠) If $(q, w, X) \overset{*}{\vdash} (p, \epsilon, \epsilon)$ then $[qXp] \overset{*}{\Rightarrow} w$.

Basis: Length 1. Then w is an a or ϵ, and $(p, \epsilon) \in \delta(q, w, X)$. By the construction of G we have $[qXp] \to w$ and thus $[qXp] \overset{*}{\Rightarrow} w$.

Induction: Length is $n > 1$, and ♠ holds for lengths $< n$. We must have

$$(q, w, X) \vdash (r_0, x, Y_1Y_2 \cdots Y_k) \vdash \cdots \vdash (p, \epsilon, \epsilon),$$

where $w = ax$ or $w = \epsilon x$. It follows that $(r_0, Y_1Y_2 \cdots Y_k) \in \delta(q, a, X)$. Then we have a production

$$[qXr_k] \to a[r_0Y_1r_1] \cdots [r_{k-1}Y_kr_k],$$

for all $\{r_1, \ldots, r_k\} \subset Q$.

We may now choose r_i to be the state in the sequence $\overset{*}{\vdash}$ when Y_i is popped. Let $w = w_1w_2 \cdots w_k$, where w_i is consumed while Y_i is popped. Then

$$(r_{i-1}, w_i, Y_i) \overset{*}{\vdash} (r_i, \epsilon, \epsilon).$$

By the IH we get

$$[r_{i-1}, Y, r_i] \overset{*}{\Rightarrow} w_i$$

We then get the following derivation sequence:

$$[qXr_k] \Rightarrow a[r_0Y_1r_1] \cdots [r_{k-1}Y_kr_k] \overset{*}{\Rightarrow}$$

$$aw_1[r_1Y_2r_2][r_2Y_3r_3] \cdots [r_{k-1}Y_kr_k] \overset{*}{\Rightarrow}$$

$$aw_1w_2[r_2Y_3r_3] \cdots [r_{k-1}Y_kr_k] \overset{*}{\Rightarrow}$$

$$\cdots$$

$$aw_1w_2 \cdots w_k = w$$

($\supseteq$-*direction.*) We shall show by an induction on the length of the derivation $\stackrel{*}{\Rightarrow}$ that

($\heartsuit$) If $[qXp] \stackrel{*}{\Rightarrow} w$ then $(q, w, X) \stackrel{*}{\vdash} (p, \epsilon, \epsilon)$

Basis: One step. Then we have a production $[qXp] \rightarrow w$. From the construction of G it follows that $(p, \epsilon) \in \delta(q, a, X)$, where $w = a$. But then $(q, w, X) \stackrel{*}{\vdash} (p, \epsilon, \epsilon)$.

Induction: Length of $\stackrel{*}{\Rightarrow}$ is $n > 1$, and $\heartsuit$ holds for lengths $< n$. Then we must have

$$[qXr_k] \Rightarrow a[r_0 Y_1 r_1][r_1 Y_2 r_2] \cdots [r_{k-1} Y_k r_k] \stackrel{*}{\Rightarrow} w$$

We can break w into $aw_2 \cdots w_k$ such that $[r_{i-1} Y_i r_i] \stackrel{*}{\Rightarrow} w_i$. From the IH we get

$$(r_{i-1}, w_i, Y_i) \stackrel{*}{\vdash} (r_i, \epsilon, \epsilon)$$

From Theorem 6.5 we get
$(r_{i-1}, w_i w_{i+1} \cdots w_k, Y_i Y_{i+1} \cdots Y_k) \stackrel{*}{\vdash}$
$(r_i, w_{i+1} \cdots w_k, Y_{i+1} \cdots Y_k)$

Since this holds for all $i \in \{1, \ldots, k\}$, we get
$(q, aw_1 w_2 \cdots w_k, X) \vdash$
$(r_0, w_1 w_2 \cdots w_k, Y_1 Y_2 \cdots Y_k) \stackrel{*}{\vdash}$
$(r_1, w_2 \cdots w_k, Y_2 \cdots Y_k) \stackrel{*}{\vdash}$
$(r_2, w_3 \cdots w_k, Y_3 \cdots Y_k) \stackrel{*}{\vdash}$
(p, ϵ, ϵ).

Deterministic PDA's

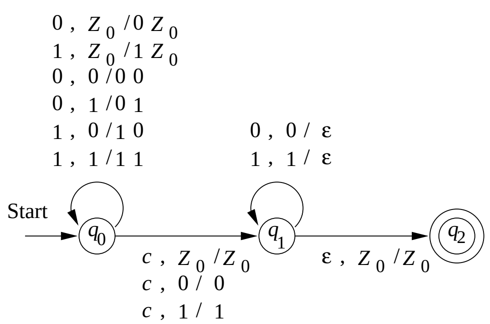
A PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is *deterministic* iff

1. $\delta(q, a, X)$ is always empty or a singleton.
2. If $\delta(q, a, X)$ is nonempty, then $\delta(q, \epsilon, X)$ must be empty.

Example: Let us define

$$L_{wcwr} = \{wcw^R : w \in \{0, 1\}^*\}$$

Then L_{wcwr} is recognized by the following DPDA



We'll show that Regular$\subset L$(DPDA) $\subset$ CFL

Theorem 6.17: If L is regular, then $L = L(P)$ for some DPDA P.

Proof: Since L is regular there is a DFA A s.t. $L = L(A)$. Let

$$A = (Q, \Sigma, \delta_A, q_0, F)$$

We define the DPDA

$$P = (Q, \Sigma, \{Z_0\}, \delta_P, q_0, Z_0, F),$$

where

$$\delta_P(q, a, Z_0) = \{(\delta_A(q, a), Z_0)\},$$

for all $p, q \in Q$, and $a \in \Sigma$.

An easy induction (do it!) on $|w|$ gives

$$(q_0, w, Z_0) \stackrel{*}{\vdash} (p, \epsilon, Z_0) \Leftrightarrow \hat{\delta_A}(q_0, w) = p$$

The theorem then follows (why?)

What about DPDA's that accept by null stack?

They can recognize only CFL's with the prefix property.

A language L has the *prefix property* if there are no two distinct strings in L, such that one is a prefix of the other.

Example: L_{wcwr} has the prefix property.

Example: $\{0\}^*$ does not have the prefix property.

Theorem 6.19: L is $N(P)$ for some DPDA P if and only if L has the prefix property and L is $L(P')$ for some DPDA P'.

Proof: Homework

- We have seen that Regular$\subseteq L$(DPDA).
- $L_{wcwr} \in L$(DPDA)$\setminus$ Regular
- Are there languages in CFL$\setminus L$(DPDA).

    Yes, for example L_{wwr}.

- What about DPDA's and Ambiguous Grammars?

    L_{wwr} has unamb. grammar $S \to 0S0|1S1|\epsilon$ but is not L(DPDA).

For the converse we have

Theorem 6.20: If $L = N(P)$ for some DPDA P, then L has an unambiguous CFG.

Proof: By inspecting the proof of Theorem 6.14 we see that if the construction is applied to a DPDA the result is a CFG with unique leftmost derivations.

Theorem 6.20 can actually be strengthen as follows

Theorem 6.21: If $L = L(P)$ for some DPDA P, then L has an unambiguous CFG.

Proof: Let \$ be a symbol outside the alphabet of L, and let $L' = L\$$.
It is easy to see that L' has the prefix property.
By Theorem 6.19 we have $L' = N(P')$ for some DPDA P'.
By Theorem 6.20 $N(P')$ can be generated by an unambiguous CFG G'
Modify G' into G, s.t. $L(G) = L$, by adding the production

$$\$ \to \epsilon$$

Since G' has unique leftmost derivations, G' also has unique lm's, since the only new thing we're doing is adding derivations

$$w\$ \underset{lm}{\Rightarrow} w$$

to the end.

Properties of CFL's

- *Simplification* of CFG's. This makes life easier, since we can claim that if a language is CF, then it has a grammar of a special form.

- *Pumping Lemma for CFL's.* Similar to the regular case.

- *Closure properties.* Some, but not all, of the closure properties of regular languages carry over to CFL's.

- *Decision properties.* We can test for membership and emptiness, but for instance, equivalence of CFL's is undecidable.

Chomsky Normal Form

We want to show that every CFL (without ϵ) is generated by a CFG where all productions are of the form

$$A \to BC, \text{ or } A \to a$$

where A, B, and C are variables, and a is a terminal. This is called CNF, and to get there we have to

1. Eliminate *useless symbols*, those that do not appear in any derivation $S \stackrel{*}{\Rightarrow} w$, for start symbol S and terminal w.

2. Eliminate *ϵ-productions*, that is, productions of the form $A \to \epsilon$.

3. Eliminate *unit productions*, that is, productions of the form $A \to B$, where A and B are variables.

Eliminating Useless Symbols

- A symbol X is *useful* for a grammar $G = (V, T, P, S)$, if there is a derivation

$$S \underset{G}{\stackrel{*}{\Rightarrow}} \alpha X \beta \underset{G}{\stackrel{*}{\Rightarrow}} w$$

for a teminal string w. Symbols that are not useful are called *useless*.

- A symbol X is *generating* if $X \underset{G}{\stackrel{*}{\Rightarrow}} w$, for some $w \in T^*$

- A symbol X is *reachable* if $S \underset{G}{\stackrel{*}{\Rightarrow}} \alpha X \beta$, for some $\{\alpha, \beta\} \subseteq (V \cup T)^*$

It turns out that if we eliminate non-generating symbols first, and then non-reachable ones, we will be left with only useful symbols.

Example: Let G be

$$S \to AB|a, \ A \to b$$

S and A are generating, B is not. If we eliminate B we have to eliminate $S \to AB$, leaving the grammar

$$S \to a, \ A \to b$$

Now only S is reachable. Eliminating A and b leaves us with

$$S \to a$$

with language $\{a\}$.

OTH, if we eliminate non-reachable symbols first, we find that all symbols are reachable. From

$$S \to AB|a, \ A \to b$$

we then eliminate B as non-generating, and are left with

$$S \to a, \ A \to b$$

that still contains useless symbols

Theorem 7.2: Let $G = (V, T, P, S)$ be a CFG such that $L(G) \neq \emptyset$. Let $G_1 = (V_1, T_1, P_1, S)$ be the grammar obtained by

1. Eliminating all nongenerating symbols and the productions they occur in. Let the new grammar be $G_2 = (V_2, T_2, P_2, S)$.

2. Eliminate from G_2 all nonreachable symbols and the productions they occur in.

The G_1 has no useless symbols, and $L(G_1) = L(G)$.

Proof: We first prove that G_1 has no useless symbols:

Let X remain in $V_1 \cup T_1$. Thus $X \overset{*}{\Rightarrow} w$ in G_1, for some $w \in T^*$. Moreover, every symbol used in this derivation is also generating. Thus $X \overset{*}{\Rightarrow} w$ in G_2 also.

Since X was not eliminated in step 2, there are α and β, such that $S \overset{*}{\Rightarrow} \alpha X \beta$ in G_2. Furthermore, every symbol used in this derivation is also reachable, so $S \overset{*}{\Rightarrow} \alpha X \beta$ in G_1.

Now every symbol in $\alpha X \beta$ is reachable and in $V_2 \cup T_2 \supseteq V_1 \cup T_1$, so each of them is generating in G_2.

The terminal derivation $\alpha X \beta \overset{*}{\Rightarrow} xwy$ in G_2 involves only symbols that are reachable from S, because they are reached by symbols in $\alpha X \beta$. Thus the terminal derivation is also a derviation of G_1, i.e.,

$$S \overset{*}{\Rightarrow} \alpha X \beta \overset{*}{\Rightarrow} xwy$$

in G_1.

We then show that $L(G_1) = L(G)$.

Since $P_1 \subseteq P$, we have $L(G_1) \subseteq L(G)$.

Then, let $w \in L(G)$. Thus $S \overset{*}{\underset{G}{\Rightarrow}} w$. Each symbol is this derivation is evidently both reachable and generating, so this is also a derivation of G_1.

Thus $w \in L(G_1)$.

We have to give algorithms to compute the generating and reachable symbols of $G = (V, T, P, S)$.

The generating symbols $g(G)$ are computed by the following closure algorithm:

Basis: $g(G) == T$

Induction: If $\alpha \in g(G)$ and $X \to \alpha \in P$, then $g(G) == g(G) \cup \{X\}$.

Example: Let G be $S \to AB|a,\ A \to b$

Then first $g(G) == \{a, b\}$.

Since $S \to a$ we put S in $g(G)$, and because $A \to b$ we add A also, and that's it.

Theorem 7.4: At saturation, $g(G)$ contains all and only the generating symbols of G.

Proof:

We'll show in class on an induction on the stage in which a symbol X is added to $g(G)$ that X is indeed generating.

Then, suppose that X is generating. Thus $X \overset{*}{\underset{G}{\Rightarrow}} w$, for some $w \in T^*$. We prove by induction on this derivation that $X \in g(G)$.

Basis: Zero Steps. Then X is added in the basis of the closure algo.

Induction: The derivation takes $n > 0$ steps. Let the first production used be $X \to \alpha$. Then

$$X \Rightarrow \alpha \overset{*}{\Rightarrow} w$$

and $\alpha \overset{*}{\Rightarrow} w$ in less than n steps and by the IH $\alpha \in g(G)$. From the inductive part of the algo it follows that $X \in g(G)$.

The set of reachable symbols $r(G)$ of $G = (V, T, P, S)$ is computed by the following closure algorithm:

Basis: $r(G) == \{S\}$.

Induction: If variable $A \in r(G)$ and $A \to \alpha \in P$ then add all symbols in α to $r(G)$

Example: Let G be $S \to AB|a,\ A \to b$

Then first $r(G) == \{S\}$.

Based on the first production we add $\{A, B, a\}$ to $r(G)$.

Based on the second production we add $\{b\}$ to $r(G)$ and that's it.

Theorem 7.6: At saturation, $r(G)$ contains all and only the reachable symbols of G.

Proof: Homework.

Eliminating ϵ-Productions

We shall prove that if L is CF, then $L \setminus \{\epsilon\}$ has a grammar without ϵ-productions.

Variable A is said to be *nullable* if $A \overset{*}{\Rightarrow} \epsilon$.

Let A be nullable. We'll then replace a rule like

$$A \to BAD$$

with

$$A \to BAD,\ A \to BD$$

and delete any rules with body ϵ.

We'll compute $n(G)$, the set of nullable symbols of a grammar $G = (V, T, P, S)$ as follows:

Basis: $n(G) == \{A : A \to \epsilon \in P\}$

Induction: If $\{C_1 C_2 \cdots C_k\} \subseteq n(G)$ and $A \to C_1 C_2 \cdots C_k \in P$, then $n(G) == n(G) \cup \{A\}$.

Theorem 7.7: At saturation, $n(G)$ contains all and only the nullable symbols of G.

Proof: Easy induction in both directions.

Once we know the nullable symbols, we can transform G into G_1 as follows:

- For each $A \to X_1 X_2 \cdots X_k \in P$ with $m \leq k$ nullable symbols, replace it by 2^m rules, one with each sublist of the nullable symbols absent.

Exeption: If $m = k$ we don't delete all m nullable symbols.

- Delete all rules of the form $A \to \epsilon$.

Example: Let G be

$$S \to AB,\ A \to aAA|\epsilon,\ B \to bBB|\epsilon$$

Now $n(G) = \{A, B, S\}$. The first rule will become

$$S \to AB|A|B$$

the second

$$A \to aAA|aA|aA|a$$

the third

$$B \to bBB|bB|bB|b$$

We then delete rules with ϵ-bodies, and end up with grammar G_1 :

$$S \to AB|A|B,\ A \to aAA|aA|a,\ B \to bBB|bB|b$$

Theorem 7.9: $L(G_1) = L(G) \setminus \{\epsilon\}$.

Proof: We'll prove the stronger statement:

($\sharp$) $A \overset{*}{\Rightarrow} w$ in G_1 if and only if $w \neq \epsilon$ and $A \overset{*}{\Rightarrow} w$ in G.

$\subseteq$-*direction:* Suppose $A \overset{*}{\Rightarrow} w$ in G_1. Then clearly $w \neq \epsilon$ (Why?). We'll show by and induction on the length of the derivation that $A \overset{*}{\Rightarrow} w$ in G also.

Basis: One step. Then there exists $A \to w$ in G_1. Form the construction of G_1 it follows that there exists $A \to \alpha$ in G, where α is w plus some nullable variables interspersed. Then

$$A \Rightarrow \alpha \overset{*}{\Rightarrow} w$$

in G.

Induction: Derivation takes $n > 1$ steps. Then

$$A \Rightarrow X_1 X_2 \cdots X_k \overset{*}{\Rightarrow} w \text{ in } G_1$$

and the first derivation is based on a production

$$A \to Y_1 Y_2 \cdots Y_m$$

where $m \geq k$, some Y_i's are X_j's and the other are nullable symbols of G.

Furhtermore, $w = w_1 w_2 \cdots w_k$, and $X_i \overset{*}{\Rightarrow} w_i$ in G_1 in less than n steps. By the IH we have $X_i \overset{*}{\Rightarrow} w_i$ in G. Now we get

$$A \underset{G}{\Rightarrow} Y_1 Y_2 \cdots Y_m \underset{G}{\overset{*}{\Rightarrow}} X_1 X_2 \cdots X_k \underset{G}{\overset{*}{\Rightarrow}} w_1 w_2 \cdots w_k = w$$

$\supseteq$-*direction:* Let $A \underset{G}{\overset{*}{\Rightarrow}} w$, and $w \neq \epsilon$. We'll show by induction of the length of the derivation that $A \overset{*}{\Rightarrow} w$ in G_1.

Basis: Length is one. Then $A \to w$ is in G, and since $w \neq \epsilon$ the rule is in G_1 also.

Induction: Derivation takes $n > 1$ steps. Then it looks like

$$A \underset{G}{\Rightarrow} Y_1 Y_2 \cdots Y_m \underset{G}{\overset{*}{\Rightarrow}} w$$

Now $w = w_1 w_2 \cdots w_m$, and $Y_i \underset{G}{\overset{*}{\Rightarrow}} w_i$ in less than n steps.

Let $X_1 X_2 \cdots X_k$ be those Y_j's in order, such that $w_j \neq \epsilon$. Then $A \to X_1 X_2 \cdots X_k$ is a rule in G_1.

Now $X_1 X_2 \cdots X_k \underset{G}{\overset{*}{\Rightarrow}} w$ (Why?)

Each $X_j / Y_j \underset{G}{\overset{*}{\Rightarrow}} w_j$ in less than n steps, so by IH we have that if $w \neq \epsilon$ then $Y_j \overset{*}{\Rightarrow} w_j$ in G_1. Thus

$$A \Rightarrow X_1 X_2 \cdots X_k \overset{*}{\Rightarrow} w \text{ in } G_1$$

The claim of the theorem now follows from statement ($\sharp$) on slide 238 by choosing $A = S$.

Eliminating Unit Productions

$$A \to B$$

is a *unit* production, whenever A and B are variables.

Unit productions can be eliminated.

Let's look at grammar

$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
$F \to I \mid (E)$
$T \to F \mid T * F$
$E \to T \mid E + T$

It has unit productions $E \to T$, $T \to F$, and $F \to I$

We'll expand rule $E \to T$ and get rules

$$E \to F,\ E \to T * F$$

We then expand $E \to F$ and get

$$E \to I|(E)|T * F$$

Finally we expand $E \to I$ and get

$$E \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \mid (E) \mid T * F$$

The expansion method works as long as there are no cycles in the rules, as e.g. in

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

The following method based on *unit pairs* will work for all grammars.

(A, B) is a *unit pair* if $A \stackrel{*}{\Rightarrow} B$ using unit productions only.

Note: In $A \to BC$, $C \to \epsilon$ we have $A \stackrel{*}{\Rightarrow} B$, but not using unit productions only.

To compute $u(G)$, the set of all unit pairs of $G = (V, T, P, S)$ we use the following closure algorithm

Basis: $u(G) == \{(A, A) : A \in V\}$

Induction: If $(A, B) \in u(G)$ and $B \to C \in P$ then add (A, C) to $u(G)$.

Theorem: At saturation, $u(G)$ contains all and only the unit pair of G.

Proof: Easy.

Given $G = (V, T, P, S)$ we can construct $G_1 = (V, T, P_1, S)$ that doesn't have unit productions, and such that $L(G_1) = L(G)$ by setting

$$P_1 = \{A \to \alpha : \alpha \notin V, B \to \alpha \in P, (A, B) \in u(G)\}$$

Example: Form the grammar of slide 242 we get

Pair	Productions
(E, E)	$E \to E + T$
(E, T)	$E \to T * F$
(E, F)	$E \to (E)$
(E, I)	$E \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(T, T)	$T \to T * F$
(T, F)	$T \to (E)$
(T, I)	$T \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(F, F)	$F \to (E)$
(F, I)	$F \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(I, I)	$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

The resulting grammar is equivalent to the original one (proof omitted).

Summary

To "clean up" a grammar we can

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

in this order.

Chomsky Normal Form, CNF

We shall show that every nonempty CFL without ϵ has a grammar G without useless symbols, and such that every production is of the form

- $A \to BC$, where $\{A, B, C\} \subseteq T$, or
- $A \to \alpha$, where $A \in V$, and $\alpha \in T$.

To achieve this, start with any grammar for the CFL, and

1. "Clean up" the grammar.
2. Arrange that all bodies of length 2 or more consists of only variables.
3. Break bodies of length 3 or more into a cascade of two-variable-bodied productions.

- For step 2, for every terminal a that appears in a body of length ≥ 2, create a new variable, say A, and replace a by A in all bodies.

Then add a new rule $A \to a$.

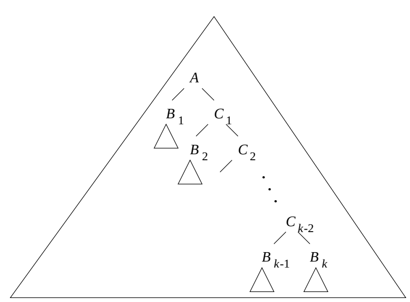
- For step 3, for each rule of the form

$$A \to B_1 B_2 \cdots B_k,$$

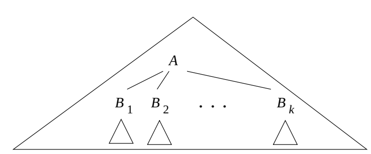
$k \geq 3$, introduce new variables $C_1, C_2, \ldots C_{k-2}$, and replace the rule with

$$\begin{array}{rcl} A & \to & B_1 C_1 \\ C_1 & \to & B_2 C_2 \\ & \cdots & \\ C_{k-3} & \to & B_{k-2} C_{k-2} \\ C_{k-2} & \to & B_{k-1} B_k \end{array}$$

Illustration of the effect of step 3



(a)



(b)

Example of CNF conversion

Let's start with the grammar (step 1 already done)

$E \to E + T \mid T * F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
$T \to T * F \mid (E)a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
$F \to (E) \ \ a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

For step 2, we need the rules
$A \to a, B \to b, Z \to 0, O \to 1$
$P \to +, M \to *, L \to (, R \to)$
and by replacing we get the grammar

$E \to EPT \mid TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$
$T \to TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$
$F \to LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$
$I \to a \mid b \mid IA \mid IB \mid IZ \mid IO$
$A \to a, B \to b, Z \to 0, O \to 1$
$P \to +, M \to *, L \to (, R \to)$

For step 3, we replace

$E \to EPT$ by $E \to EC_1, C_1 \to PT$

$E \to TMF, T \to TMF$ by
$E \to TC_2, T \to TC_2, C_2 \to MF$

$E \to LER, T \to LER, F \to LER$ by
$E \to LC_3, T \to LC_3, F \to LC_3, C_3 \to ER$

The final CNF grammar is

$E \to EC_1 \mid TC_2 \mid LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$
$T \to TC_2 \mid LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$
$F \to LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$
$I \to a \mid b \mid IA \mid IB \mid IZ \mid IO$
$C_1 \to PT, C_2 \to MF, C_3 \to ER$
$A \to a, B \to b, Z \to 0, O \to 1$
$P \to +, M \to *, L \to (, R \to)$

The size of parse trees

Theorem: Suppose we have a parse tree according to a CFG G in CNF, and let w be the yield of the tree. If the longest path (no. of edges) in the tree is n, then $|w| \le 2^{n-1}$.

Proof: Induction on n.

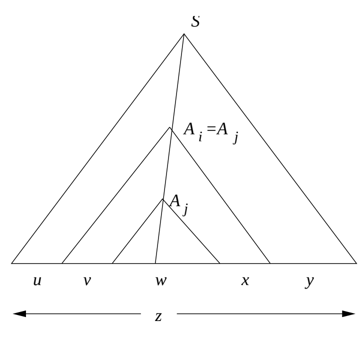
Basis: $n = 1$. Then the tree consists of a root and a leaf, and the production must be of the form $S \to a$. Thus $|w| = |a| = 1 = 2^0 = 2^{n-1}$.

Induction: Let the longest path be n. Then the root must use a production of the form $S \to AB$. No path in the subtrees rooted at A and B can have a path longer than $n-1$. Thus the IH applies, and $S \Rightarrow AB \stackrel{*}{\Rightarrow} w = uv$, where where $A \stackrel{*}{\Rightarrow} u$ and $B \stackrel{*}{\Rightarrow} v$. By the IH we have $|u| \le 2^{n-2}$ and $|v| \le 2^{n-2}$. Consequently $|w| = |u| + |v| \le 2^{n-2} + 2^{n-2} = 2^{n-1}$.

The Pumping Lemma for CFL's

Theorem: Let L be a CFL. Then there exists a constant n such that for any $z \in L$, if $|z| \ge n$, then z can be written as $uvwxy$, where

1. $|vwx| \le n$.
2. $vx \neq \epsilon$
3. $uv^iwx^iy \in L$, for all $i \ge 0$.

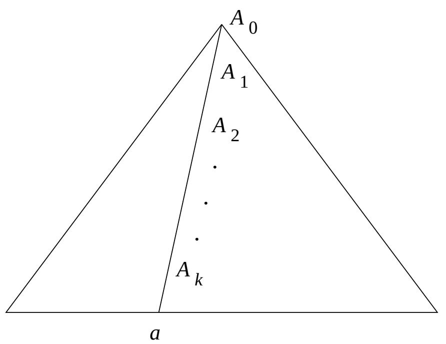


Proof:

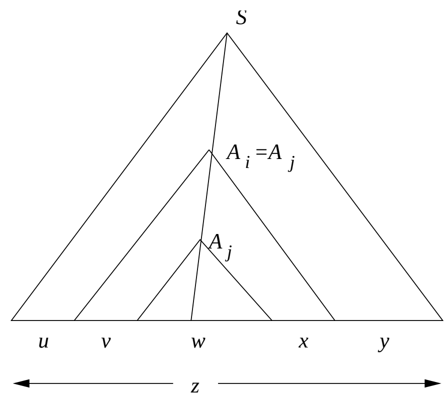
Let G be a CFG in CNF, such that $L(G) = L \setminus \{\epsilon\}$, and let m be the number of variables in G.

Choose $n = 2^m$. Let w be a yield of a parse three where the longest path is at most m. By the previous theorem $|w| \le 2^{m-1} = n/2$.

Since $|z| \ge n$ the parse tree for z must have a path of length $k \ge m+1$.

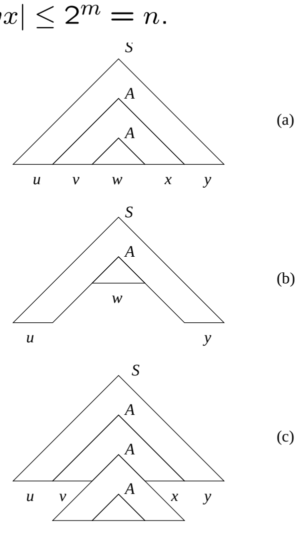


Since G has only m variables, at least one variable has to be repeated. Suppose $A_i = A_j$, where $k - m \le i < j \le k$ (choose A_i as close to the bottom as possible).



Then we can pump the tree in (a) as uv^0wx^0y (tree (b)) or uv^2wx^2 (tree (c)), and in general as uv^iwx^iy, $i \ge 0$.

Since the longest path in the subtree rooted at A_i is at most $m+1$, the previous theorem gives us $|vwx| \le 2^m = n$.



Closure Properties of CFL's

Consider a mapping

$$s : \Sigma \to 2^{\Delta^*}$$

where Σ and Δ are finite alphabets. Let $w \in \Sigma^*$, where $w = a_1 a_2 \ldots a_n$, and define

$$s(w) = s(a_1).s(a_2).\cdots.s(a_n)$$

and, for $L \subseteq \Sigma^*$,

$$s(L) = \bigcup_{w \in L} s(w).$$

Such a mapping s is called a *substitution*.

Example: $\Sigma = \{0, 1\}, \Delta = \{a, b\}$,
$s(0) = \{a^n b^n : n \geq 1\}$, $s(1) = \{aa, bb\}$.

Let $w = 01$. Then $s(w) = s(0).s(1) =$

$$\{a^n b^n aa : n \geq 1\} \cup \{a^n b^{n+2} : n \geq 1\}.$$

Let $L = \{0\}^*$. Then $s(L) = (s(0))^* =$

$$\{a^{n_1} b^{n_1} a^{n_2} b^{n_2} \cdots a^{n_k} b^{n_k} : k \geq 0, n_i \geq 1\}.$$

Theorem 7.23: Let L be a CFL over Σ, and s a substitution, such that $s(a)$ is a CFL, $\forall a \in \Sigma$. Then $s(L)$ is a CFL.

Proof: We start with grammars

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

for L, and

$$G_a = (V_a, T_a, P_a, S_a)$$

for each $s(a)$. We then construct

$$G' = (V', T', P', S)$$

where

$V' = (\bigcup_{a \in \Sigma} V_a) \cup V$

$T' = \bigcup_{a \in \Sigma} T_a$

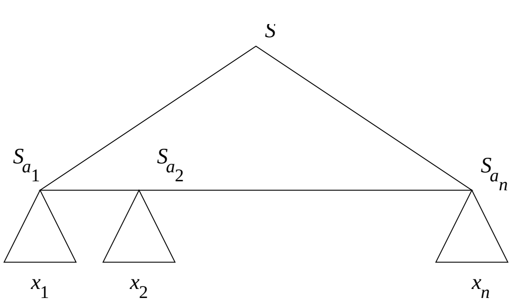
$P' = \bigcup_{a \in \Sigma} P_a$ plus the productions of P with each a in a body replaced with symbol S_a.

Now we have to show that

- $L(G') = s(L)$.

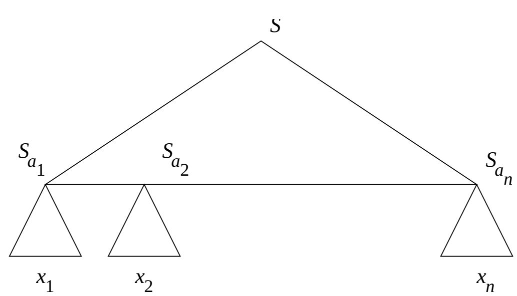
Let $w \in s(L)$. Then $\exists x = a_1 a_2 \ldots a_n$ in L, and $\exists x_i \in s(a_i)$, such that $w = x_1 x_2 \ldots x_n$.

A derivation tree in G' will look like



Thus we can generate $S_{a_1} S_{a_2} \ldots S_{a_n}$ in G' and form there we generate $x_1 x_2 \ldots x_n = w$. Thus $w \in L(G')$.

Then let $w \in L(G')$. Then the parse tree for w must again look like



Now delete the dangling subtrees. Then you have yield

$$S_{a_1} S_{a_2} \ldots S_{a_n}$$

where $a_1 a_2 \ldots a_n \in L(G)$. Now w is also equal to $s(a_1 a_2 \ldots a_n)$, which is in $s(L)$.

Applications of the Substitution Theorem

Theorem 7.24: The CFL's are closed under (i) : union, (ii) : concatenation, (iii) : Kleene closure and positive closure $+$, and (iv) : homomorphism.

Proof: (i): Let L_1 and L_2 be CFL's, let $L = \{1,2\}$, and $s(1) = L_1, s(2) = L_2$. Then $L_1 \cup L_2 = s(L)$.

(ii) : Here we choose $L = \{12\}$ and s as before. Then $L_1.L_2 = s(L)$.

(iii) : Suppose L_1 is CF. Let $L = \{1\}^*, s(1) = L_1$. Now $L_1^* = s(L)$. Similar proof for $+$.

(iv) : Let L_1 be a CFL over Σ, and h a homomorphism on Σ. Define s by

$$a \mapsto \{h(a)\}$$

Then $h(L) = s(L)$.

Theorem: If L is CF, then so in L^R.

Proof: Suppose L is generated b $G = (V, T, P, S)$. Construct $G^R = (V, T, P^R, S)$, where

$$P^R = \{A \to \alpha^R : A \to \alpha \in P\}$$

Show at home by inductions on the lengths of the derivations in G (for one direction) and in G^R (for the other direction) that $(L(G))^R = L(G^R)$.

CFL's are <u>not</u> closed under $\cap$

Let $L_1 = \{0^n 1^n 2^i : n \geq 1, i \geq 1\}$. The L_1 is CF with grammar

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

Also, $L_2 = \{0^i 1^n 2^n : n \geq 1, i \geq 1\}$ is CF with grammar

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

However, $L_1 \cap L_2 = \{0^n 1^n 2^n : n \geq 1\}$ which is not CF, as we have proved using the pumping lemma for CFL's.

CF $\cap$ Regular $=$ CF

Theorem 7.27: If L is CR, and R regular, then $L \cap R$ is CF.

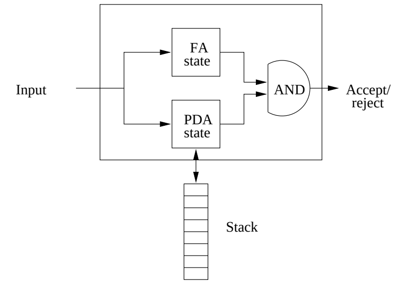
Proof: Let L be accepted by PDA

$$P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$$

by final state, and let R be accepted by DFA

$$A = (Q_A, \Sigma, \delta_A, q_A, F_A)$$

We'll construct a PDA for $L \cap R$ according to the picture



Formally, define

$$P' = (Q_P \times Q_A, , \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A)$$

where

$$\delta\Big((q,p), a, X\Big) = \Big\{\Big((r, \widehat{\delta}_A(p,a)), \gamma\Big) : (r,\gamma) \in \delta_P(q,a,X)\Big\}$$

Prove at home by an induction $\overset{*}{\vdash}$, both for P and for P' that

$$(q_P, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \gamma), \text{ and } \widehat{\delta}(q_A, w) \in F_A$$

if and only if

$$\Big((q_P, q_A), w, Z_0\Big) \Big((q, \widehat{\delta}(p_A, w)), \epsilon, \gamma\Big)$$

The claim then follows (Why?)

Theorem 7.29: Let L, L_1, L_2 be CFL's and R regular. Then

1. $L \setminus R$ is CF
2. $\bar{L}$ is not necessarily CF
3. $L_1 \setminus L_2$ is not necessarily CF

Proof:

1. $\bar{R}$ is regular, $L \cap \bar{R}$ is regular, and $L \cap \bar{R} = L \setminus R$.
2. If $\bar{L}$ always were CF, it would follow that
$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$
always would be CF.
3. Note that Σ^* is CF, so if $L_1 \setminus L_2$ were always CF, then so would $\Sigma^* \setminus L = \bar{L}$.

Inverse homomorphism

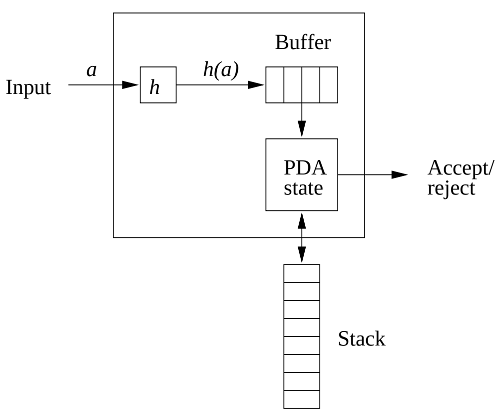
Let $h : \Sigma \to \Theta^*$ be a homom. Let $L \subseteq \Theta^*$, and define

$$h^{-1}(L) = \{w \in \Sigma^* : h(w) \in L\}$$

Now we have

Theorem 7.30: Let L be a CFL, and h a homomorphism. Then $h^{-1}(L)$ is a CFL.

Proof: The plan of the proof is



Let L be accepted by PDA

$$P = (Q, \Theta, \Gamma, \delta, q_0, Z_0, F)$$

We construct a new PDA

$$P' = (Q', \Sigma, \Gamma, \delta', (q_0, \epsilon), Z_0, F \times \{\epsilon\})$$

where

- $Q' = \{(q, x) : q \in Q, x \in \textit{suffix}(h(a)),\ a \in \Sigma\}$
- $\delta'((q,\epsilon), a, X) =$
  $\{((q, h(a)), X) : \epsilon \neq a \in \Sigma,\ q \in Q,\ X \in \Gamma\}$
- $\delta'((q, bx), \epsilon, X) = \{((p, x), \gamma) :$
  $(p, \gamma) \in \delta(q, b, X),\ b \in T \cup \{\epsilon\},\ q \in Q, X \in \Gamma\}$

Show at home by suitable inductions that

- $(q_0, h(w), Z_0) \overset{*}{\vdash} (p, \epsilon, \gamma)$ in P if and only if $((q_0, \epsilon), w, Z_0) \overset{*}{\vdash} ((p, \epsilon), \epsilon, \gamma)$ in P'.

Decision Properties of CFL's

We'll look at the following:

- Complexity of converting among CFA's and PDAQ's
- Converting a CFG to CNF
- Testing $L(G) \neq \emptyset$, for a given G
- Testing $w \in L(G)$, for a given w and fixed G.
- Preview of undecidable CFL problems

Converting between CFA's and PDA's

- Input size is n.
- n is the *total* size of the input CFG or PDA.

The following work in time $O(n)$

1. Converting a CFG to a PDA (slide 203)
2. Converting a "final state" PDA to a "null stack" PDA (slide 199)
3. Converting a "null stack" PDA to a "final state" PDA (slide 195)

Avoidable exponential blow-up

For converting a PDA to a CFG we have

(slide 210)

At most n^3 variables of the form $[pXq]$

If $(\mathbf{r}, Y_1 Y_2 \cdots Y_k) \in \delta(\mathbf{q}, a, X)\}$, we'll have $O(n^n)$ rules of the form

$$[\mathbf{q}X r_k] \rightarrow a[\mathbf{r}Y_1 r_1] \cdots [r_{k-1} Y_k r_k]$$

- By introducing $k-2$ new states we can modify the PDA to push at most *one* symbol per transition. Illustration on blackboard in class.

- Now, k will be ≤ 2 for all rules.
- Total length of all transitions is still $O(n)$.
- Now, each transition generates at most n^2 productions
- Total size (and time to calculate) the grammar is therefore $O(n^3)$.

Converting into CNF

Good news:

1. Computing $r(G)$ and $g(G)$ and eliminating useless symbols takes time $O(n)$. This will be shown shortly

   (slides 229,232,234)

2. Size of $u(G)$ and the resulting grammar with productions P_1 is $O(n^2)$

   (slides 244,245)

3. Arranging that bodies consist of only variables is $O(n)$

   (slide 248)

4. Breaking of bodies is $O(n)$ (slide 248)

Bad news:

- Eliminating the nullable symbols can make the new grammar have size $O(2^n)$

(slide 236)

The bad news are avoidable:

Break bodies first before eliminating nullable symbols

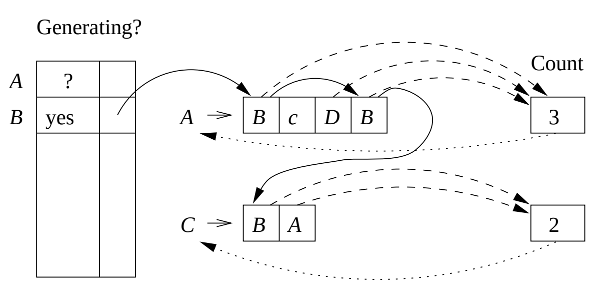
- Conversion into CNF is $O(n^2)$

Testing emptiness of CFL's

$L(G)$ is non-empty if the start symbol S is generating.

A naive implementation on $g(G)$ takes time $O(n^2)$.

$g(G)$ can be computed in time $O(n)$ as follows:



Creation and initialzation of the array is $O(n)$

Creation and initialzation of the links and counts is $O(n)$

When a count goes to zero, we have to

1. Finding the head variable A, checkin if it already is "yes" in the array, and if not, queueing it is $O(1)$ per production. Total $O(n)$

2. Following links for A, and decreasing the counters. Takes time $O(n)$.

Total time is $O(n)$.

$w \in L(G)$**?**

Inefficient way:

Suppose G is CNF, test string is w, with $|w| = n$. Since the parse tree is binary, there are $2n-1$ internal nodes.

Generate *all* binary parse trees of G with $2n-1$ internal nodes.

Check if any parse tree generates w

CYK-algo for membership testing

The grammar G is fixed

Input is $w = a_1 a_2 \cdots a_n$

We construct a triangular table, where X_{ij} contains all variables A, such that

$$A \overset{*}{\underset{G}{\Rightarrow}} a_i a_{i+1} \cdots a_j$$

X_{15}				
X_{14}	X_{25}			
X_{13}	X_{24}	X_{35}		
X_{12}	X_{23}	X_{34}	X_{45}	
X_{11}	X_{22}	X_{33}	X_{44}	X_{55}
a_1	a_2	a_3	a_4	a_5

To fill the table we work row-by-row, upwards

The first row is computed in the basis, the subsequent ones in the induction.

Basis: $X_{ii} == \{A : A \to a_i \text{ is in } G\}$

Induction:

We wish to compute X_{ij}, which is in row $j-i+1$.

$A \in X_{ij}$, if
$A \overset{*}{\Rightarrow} a_i a_i + 1 \cdots a_j$, if
for some $k < j$, and $A \to BC$, we have
$B \overset{*}{\Rightarrow} a_i a_{i+1} \cdots a_k$, and $C \overset{*}{\Rightarrow} a_{k+1} a_{k+2} \cdots a_j$, if
$B \in X_{ik}$, and $C \in X_{kj}$

Example:

G has productions

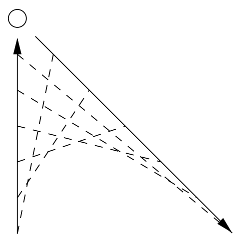
$$\begin{array}{rcl} S & \to & AB|BC \\ A & \to & BA|a \\ B & \to & CC|b \\ C & \to & AB|a \end{array}$$

{S,A,C}				
-	{S,A,C}			
-	{B}	{B}		
{S,A}	{B}	{S,C}	{S,A}	
{B}	{A,C}	{A,C}	{B}	{A,C}
b	a	a	b	a

To compute X_{ij} we need to compare at most n pairs of previously computed sets:

$(X_{ii}, X_{i=1,j}), (X_{i,i+1}, X_{i+2,j}), \ldots, (X_{i,j-1}, X_{jj})$

as suggested below



For $w = a_1 \cdots a_n$, there are $O(n^2)$ entries X_{ij} to compute.

For each X_{ij} we need to compare at most n pairs $(X_{ik}, X_{k+1,j})$.

Total work is $O(n^3)$.

Preview of undecidable CFL problems

The following are undecidable:

1. Is a given CFG G ambiguous?

2. Is a given CFL inherently ambiguous?

3. Is the intersection of two CFL's empty?

4. Are two CFL's the same?

5. Is a given CFL universal (equal to Σ^*)?

∞

Problems that computers cannot solve

Evidently, it is important to know that programs do what they are supposed to, IOW, we would like to make sure that programs are *correct*.

It is easy to see that the program

```
main()
{
       printf(‘‘hello, world\n’’);
}
```

indeed prints `hello, world`.

What about the program in Fig. 8.2 in the textbook?

It will print **hello, world**, for input n if and only if the equation

$$x^n + y^n = z^n$$

has a solution where x, y, and z are integers.

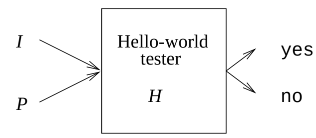
We now know that it will print **hello, world**, for input $n = 2$, and loop forever for inputs $n > 2$.

It took humanity 300+ years to prove this.

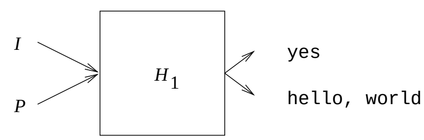
Can we hope to have a program that proves the correctness of programs?

The hypothetical "Hello, world" tester H

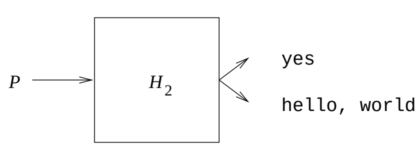
Suppose the following program H exists.



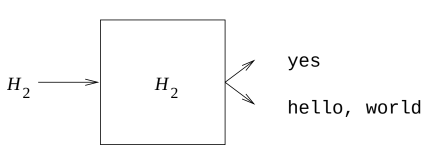
Modify the **no** print statement of H to **hello, world**. We get program H_1



Modify H_1 so that it takes only P as input, stores P and uses it both as P and I. We get program H_2.

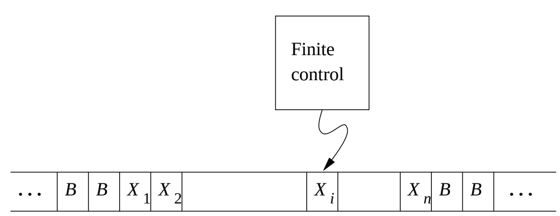


Give H_2 as input to H_2.



- If H_2 prints **yes** it should have printed **hello, world**.
- If H_2 prints **hello, world** it should have printed **yes**.
- Thus H_2 cannot exist.
- Consequently H cannot exist either.

The Turing Machine (1936)



A TM makes a *move* depending on its state, and the symbol under the tape head.

In a move, a TM will

1. Change state

2. Write a tape symbol in the cell scanned

3. Move the tape head one cell left or right

A (deterministic) *Turing Machine* is a 7-tuple

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

where

- Q is a finite set of *states*,
- Σ is a finite set of *input symbols*,
- Γ is a finite set of *tape symbols*, $\Gamma \supset \Sigma$
- δ is a *transition function* from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$,
- q_0 is the *start state*,
- $B \in \Gamma \setminus \Sigma$ is the *blank symbol*, and
- $F \subseteq Q$ is the set of final states.

Instantaneous Description

A TM changes *configuration* after each move.

We use Instantaneous Descriptions, ID's, to describe configurations.

An ID is a string of the form

$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n$$

where

1. q is the state of the TM.
2. $X_1 X_2 \cdots X_n$ is the non-blank portion of the tape
3. The tape head is scanning the ith symbol

The moves and the language of a TM

We'll use $\underset{M}{\vdash}$ to indicate a move by M from a configuration to another.

- Suppose $\delta(q, X_i) = (p, Y, L)$. Then

$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n \underset{M}{\vdash} X_1 X_2 \cdots p X_{i-1} Y X_{i+1} \cdots X_n$$

- If $\delta(q, X_i) = (p, Y, R)$, we have

$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n \underset{M}{\vdash} X_1 X_2 \cdots X_{i-1} Y p X_{i+1} \cdots X_n$$

We denote the reflexive-transitive closure of $\underset{M}{\vdash}$ with $\underset{M}{\overset{*}{\vdash}}$.

- A TM $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ *accepts* the language

$$L(M) = \{w \in \Sigma^* : q_0 w \underset{M}{\overset{*}{\vdash}} \alpha p \beta, p \in F, \alpha, \beta \in \Gamma^*\}$$

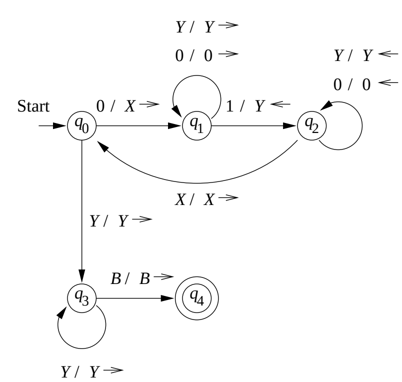
A TM for $\{0^n 1^n : n \geq 1\}$

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

where δ is given by the following table

	0	1	X	Y	B
$\rightarrow q_0$	(q_1, X, R)			(q_3, Y, R)	
q_1	$(q_1, 0, R)$	(q_2, Y, L)		(q_1, Y, R)	
q_2	$(q_2, 0, L)$		(q_0, X, R)	(q_2, Y, L)	
q_3				(q_3, Y, R)	(q_4, B, R)
$\star q_4$					

We can represent M by the following *transition diagram*



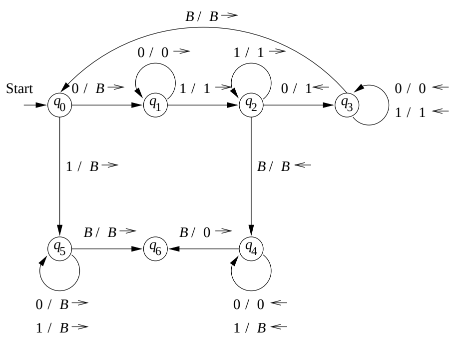
A TM with "output"

The following TM computes

$$m \dot{-} n = max(m - n, 0)$$

	0	1	B
$\rightarrow q_0$	(q_1, B, R)	(q_5, B, R)	
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	
q_2	$(q_1, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
$\star q_6$			

The transition diagram is



Programming Techniques for TM's

Although TM's seem very simple, they are as powerful as any computer.

Lots of "features" can be simulated with a "standard" machine.

- Storage in State

A TM M that "remembers" the first symbol.

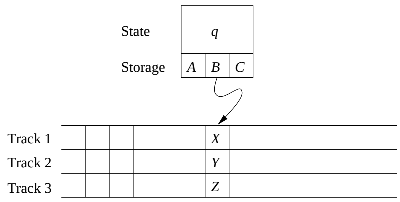
$$M = \Big(Q, \{0, 1\}, \{0, 1, B\}, \delta, [q_0, B], B, \{[q_1, B]\}\Big)$$

where $Q = \{q_0, q_1\} \times \{0, 1, B\}$.

δ	0	1	B
$\rightarrow [q_0, B]$	$([q_1, 0], 0, R)$	$([q_1, 1], 1, R)$	
$[q_1, 0]$		$([q_1, 0], 1, R)$	$([q_1, B], B, R)$
$[q_1, 1]$	$([q_1, 0], 1, R)$		$([q_1, B], B, R)$
$\star[q_1, B]$			

$$L(M) = L(01^* + 10^*).$$

- Multiple Tracks for $\{wcw : w \in \{0, 1\}^*\}$



$$M = (Q, \Sigma, \Gamma, \delta, [q_1, B], [B, B], \{[q_0, B]\})$$

where

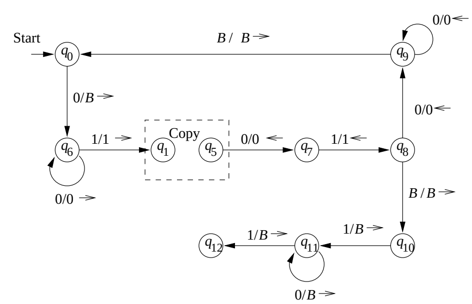
$$Q = \{q_1, q_2, \ldots, q_9\} \times \{0, 1, B\}$$

$$\Sigma = \{[B, 0], [B, 1], [B, c]\}$$

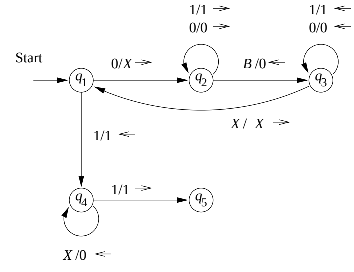
$$\Gamma = \{B, *\} \times \{0, 1, c, B\}$$

- Subroutines

The TM computes $0^m 1 0^n 1 \mapsto 0^{m \cdot n}$

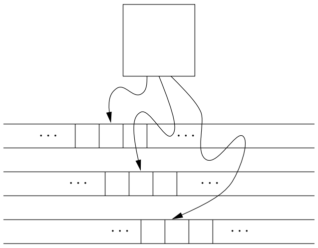


Here is the "Copy" TM



Variations of the basic TM

- Multitape TM's. Input on first tape.

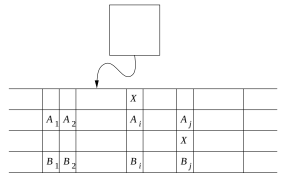


In one move the TM

1. Remains in the same state or enters a new state.
2. For each tape, writes a new symbol in the current cell.
3. Independently moves each head left or right.

Theorem: Every language accepted by a multi-tape TM M is RE

Proof idea: Simulate M by multitrack TM N



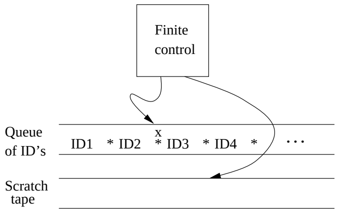
1. $2k$ tracks for k tape simulation. Even tracks = tape content. Odd tracks = head position.
2. N has to visit k head markers to simulate a move of M. Store the number of heads visited in state.
3. For each head N does what M does on the corresponding tape.
4. N changes state acording to M.

- Nondeterministic TM's

Theorem: For every nondeterministic TM M_N there is a deterministic TM M_D such that

$$L(M_N) = L(M_D).$$

Proof idea: Suppose at each state M_N has k choiches for each symbol.



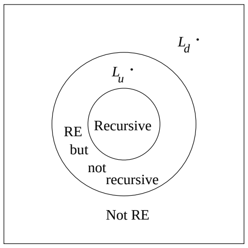
1. M_D has transitions $\delta(q, a, k)$.
2. For each ID copy it to scratch tape. For each k create a new ID at the end of the queue.
3. Unmark the current ID and go to next ID.

Undecidability

We want to prove undecidable L_u which is the language of pairs (M, w) such that:

1. M is a TM (encoded in binary) with input alphabet $\{0, 1\}$.
2. $w \in \{0, 1\}^*$.
3. M accepts w.

The landscape of languages (problems)



1. The recursive languages L:

   There is a TM M that always halts and such that $L(M) = L$. IOW there is an *algorithm* that on input w answers "yes" if $w \in L$, and answers "no" if $w \notin L$.
2. The recursively enumerable languages L:

   There is a TM M that halts and accepts if $w \in L$, and might run forever otherwise.
3. The non-RE languages L:

   There is no TM whatsoever for L.

Encoding TM's

- Enumerate strings: $w \in \{0,1\}^*$ is treated as binary integer $1w$. The ith string is denoted w_i.

$\epsilon = w_1, 0 = w_2, 1 = w_3, 00 = w_4, 01 = w_5, \ldots$

To enccode $M = (Q, \{0,1\}, \Gamma, \delta, q_1, B, F)$

- Assume $Q = \{q_1, q_2, \ldots, q_r\}, F = \{q_2\}$.
- Assume $\Gamma = \{X_1, X_2, \ldots, X_s\}$, where $X_1 = 0, X_2 = 1$, and $X_3 = B$.
- Encode L as D_1 and R as D_2.
- Encode $\delta(q_i, X_j) = (q_k, X_\ell, D_m)$ as
  $$0^i 1 0^j 1 0^k 1 0^\ell 1 0^m$$
- Encode the entire TM as
  $$C_1 11 C_2 11 \cdots 11 c_{n-1} 11 C_n$$
  where each C_i is the code of one transition.

Example:

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

where

δ	0	1	B
$\rightarrow q_1$		$(q_3, 0, R)$	
$\star q_2$			
q_3	$(q_1, 1, R)$	$(q_2, 0, R)$	$(q_3, 1, L)$

The code for transitions C_1, C_2, C_3, C_4:

0100100010100
0001010100100
00010010010100
0001000100010010

The code for the entire M:

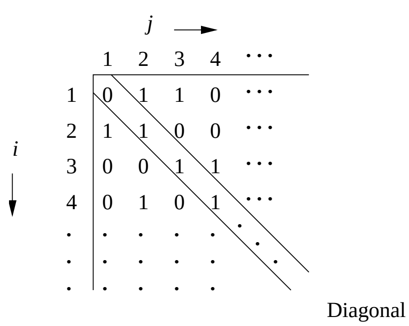
01001000101001100010101001001100010010010100110001000100010010

The diagonalization language L_d

The ith TM M_i: The TM whose code is w_i.

Write down a matrix where

$$(i,j) = \begin{cases} 1 & \text{if } w_j \in L(M_i), \\ 0 & \text{otherwise} \end{cases}$$

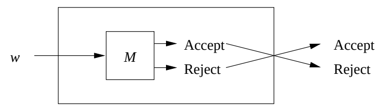


Define $L_d = \{w_i : w_i \notin L(M_i)\}$

If there is a TM for L_d it must be M_i for some i.
If $w_i \in L_d$ then $w_i \in L(M_i)$ and thus $w_i \notin L_d$
If $w_i \notin L_d$ then $w_i \notin L(M_i)$ and thus $w_i \in L_d$

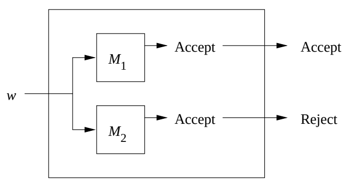
Theorem: If L is recursive, so is $\bar{L}$.

Proof:



Theorem: If L is RE and $\overline{L}$ also is RE, then L is recursive.

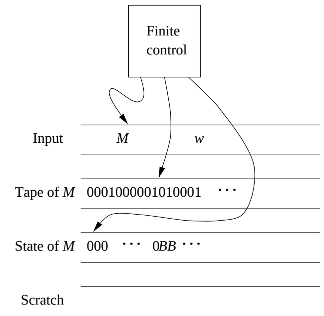
Proof:



The universal language L_u

$$L_u = \{(\text{enc}(M), \text{enc}(w)) : w \in L(M)\}$$

TM U where $L(U) = L_u$



Operation of U:

1. If enc(M) is not legitimate halt and reject

2. Write enc(w) on tape 2. Use the blank of U for 1000

3. Write the start state 0 on tape 3. Place head of tape 2 on first simulated cell

4. Search tape 1 for $0^i10^j10^k10^\ell10^m$, where

   (a) 0^i is the state on tape 3

   (b) 0^j is tape symbol of M that begins under the head on 2

5. Make the move

   (a) Change tape 3 to 0^k

   (b) Replace 0^j on tape 2 by 0^ℓ

   (c) Move head 2 left (if $m = 1$) or right (if $m = 2$) to next 1

   (d) If no $0^i10^j1 \cdots 1 \cdots$ is not found on tape 1, then halt and reject

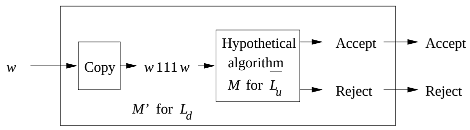
   (e) If M enters its accepting state then accept and halt

Theorem: L_u is RE but not recursive.

Proof: L_u is RE since $L(U) = L_u$.

Suppose L_u were recursive. Then $\overline{L_u}$ is also recursive. Let M be an always halting TM with $L(M) = \overline{L_u}$.

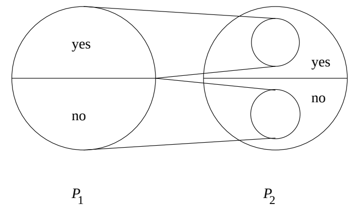
We modify M to M', such that $L(M') = L_d$



- $w_i \in L(M') \Rightarrow w_i 111 w_i \in \overline{L_u} \Rightarrow w_i \notin L(M_i) \Rightarrow w_i \in L_d$
- $w_i \notin L(M') \Rightarrow w_i 111 w_i \in L_u \Rightarrow w_i \in L(M_i) \Rightarrow w_i \notin L_d$

Reductions for proving lower bounds

Find an algorithm that reduces a known hard problem P_1 to P_2.



Theorem: If there is a reduction from P_1 to P_2, then

1. If P_1 is undecidable, then so is P_2
2. If P_1 is non-RE, then so is P_2.

Proof: by contradiction. If there were an algorithm for P_2 you could also solve P_1 by first reducing P_1 to P_2 and then running the algorithm for P_2.

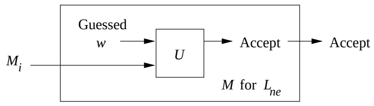
A non-recursive and a non-RE language

$$L_e = \{\text{enc}(M) : L(M) = \emptyset\}$$

$$L_{ne} = \{\text{enc}(M) : L(M) \neq \emptyset\}$$

Theorem: L_{ne} is recursively enumerable.

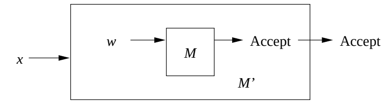
Proof: Non-deterministic TM for L_{ne}



Theorem: $L_{ne} = \{\text{enc}(M) : L(M) \neq \emptyset\}$ is not recursive.

Proof: by contradiction. Suppose $\exists$ TM M, such that $L(M) = L_{ne}$. Transform instance (M, w) of L_u into M' such that

$$w \in L(M) \Leftrightarrow L(M') \neq \emptyset$$



We have reduced L_u to L_{ne}.

Suppose there is an algorithm for L_{ne}.

Run the algorithm to see if $L(M') \neq \emptyset$.

Since L_u is not recursive, L_{ne} cannot be recursive either.

Theorem: $L_e = \{\text{enc}(M) : L(M) = \emptyset\}$ is not RE

Proof: If L_e were RE then L_{ne} would be recursive, since $\overline{L_e} = L_{ne}$.

Other undecidable properties of TM's

1. $L_{fin} = \{\text{enc}(M) : L(M) \text{ is finite}\}$

2. $L_{reg} = \{\text{enc}(M) : L(M) \text{ is regular}\}$

3. $L_{cfl} = \{\text{enc}(M) : L(M) \text{ is a CFL}\}$

These follow from Rice's Theorem.

Properties of the RE languages

Every nontrivial property of the RE languages is undecidable.

Property of RE languages (example):
"the language is CF"

Formally: A *nontrivial property* is a nonempty strict subset of all RE languages.

Let $\mathcal{P}$ be a nontrivial property of the RE languages.

$L_{\mathcal{P}} = \{\text{enc}(M) : L(M) \in \mathcal{P}\}$.

Rice's Theorem: $L_{\mathcal{P}}$ is not recursive.

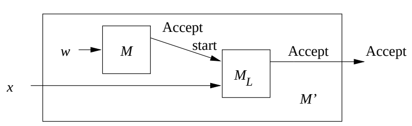
Proof of Rice's Theorem:

Suppose first $\emptyset \notin \mathcal{P}$.

Let $L \in \mathcal{P}$ and M_L be a TM such that $L(M_L) = L$.

Transform instance (M, w) of L_u into TM M' such that

$$L(M') = \begin{cases} L & \text{if } w \in L(M), \\ \emptyset & \text{if } w \notin L(M) \end{cases}$$



We have reduced L_u to $L_{\mathcal{P}}$.

Suppose there is an algorithm for $L_{\mathcal{P}}$.

Run the algorithm to see if $L(M') \neq \emptyset$.

Since L_u is not recursive, $L_{\mathcal{P}}$ cannot be recursive either.

Proof of Rice's Theorem:

Suppose then that $\emptyset \in \mathcal{P}$

Consider $\overline{\mathcal{P}}$: the set of RE languages that do not have property $\mathcal{P}$. Based on the above $\overline{\mathcal{P}}$ is undecidable.

Since every TM accepts an RE language we have

$$\overline{L_{\mathcal{P}}} = L_{\overline{\mathcal{P}}}$$

If $L_{\mathcal{P}}$ were decidable then $L_{\overline{\mathcal{P}}}$ would also be decidable.

Post's Correspondence Problem

PCP is a problem about strings that is undecidable (RE)

Let $A = w_1, w_2, \ldots, w_k$ and $B = x_1, x_2, \ldots, x_k$, where $x_i, y_i \in \Sigma^*$ for some alphabet Σ.

The instance (A, B) has a *solution* if there exists indices $i_1, i_2, \ldots, i_m$, such that

$$w_{i_1} w_{i_2} \cdots w_{i_m} = x_{i_1} x_{i_2} \cdots x_{i_m}$$

Example:

	List A	List B
i	w_i	x_i
1	1	111
2	10111	10
3	10	0

Solution: $i_1 = 2, i_2 = 1, i_3 = 1, i_4 = 3$ gives

$$w_2 w_1 w_1 w_3 = x_2 x_1 x_1 x_3 = 101111110$$

Another solution: 2,1,1,3,2,1,1,3

Example:

	List A	List B
i	w_i	x_i
1	10	101
2	011	11
3	101	011

This PCP instance has no solution.

Suppose $i_1, i_2, \ldots, i_m$ is a solution:

If $i_1 = 2$ we cannot match $\begin{matrix} w_2 \\ x_2 \end{matrix} = \begin{matrix} 011 \\ 11 \end{matrix}$

If $i_1 = 3$ we cannot match $\begin{matrix} w_3 \\ x_3 \end{matrix} = \begin{matrix} 101 \\ 011 \end{matrix}$

Therefore $i_1 = 1$ and a partial solution is $\begin{matrix} w_1 \\ x_1 \end{matrix} = \begin{matrix} 10 \\ 101 \end{matrix}$

If $i_2 = 1$ we cannot match $\begin{matrix} w_1 w_1 \\ x_1 x_1 \end{matrix} = \begin{matrix} 1010 \\ 101101 \end{matrix}$

If $i_2 = 2$ we cannot match $\begin{matrix} w_1 w_2 \\ x_1 x_2 \end{matrix} = \begin{matrix} 10011 \\ 10111 \end{matrix}$

Only $i_2 = 3$ is possible giving $\begin{matrix} w_1 w_3 \\ x_1 x_3 \end{matrix} = \begin{matrix} 10101 \\ 101011 \end{matrix}$

Now we are back to "square one:"

Only $i_3 = 3$ is possible giving $\begin{matrix} w_1 w_3 \\ x_1 x_3 \end{matrix} = \begin{matrix} 10101101 \\ 101011011 \end{matrix}$

Only $i_4 = 3$ is possible giving $\begin{matrix} w_1 w_3 w_3 \\ x_1 x_3 x_3 \end{matrix} = \begin{matrix} 10101101101 \\ 101011011011 \end{matrix}$

Conclusion: The first list can never catch up with the second

The Modified PCP:

Let $A = w_1, w_2, \ldots, w_k$ and $B = x_1, x_2, \ldots, x_k$, where $x_i, y_i \in \Sigma^*$ for some alphabet Σ.

The modifeid PCP A, B has a *solution* if there exists indices $i_1, i_2, \ldots, i_m$, such that

$$w_1 w_{i_1}, w_{i_2}, \ldots, w_{i_m} = x_1 x_{i_1}, x_{i_2}, \ldots x_{i_m}$$

Example:

	List A	List B
i	w_i	x_i
1	1	111
2	10111	10
3	10	0

Any solution would have to begin with $\begin{matrix} w_1 \\ x_1 \end{matrix} = \begin{matrix} 1 \\ 111 \end{matrix}$

If $i_2 = 2$ we cannot match $\begin{matrix} w_1 w_2 \\ x_1 x_2 \end{matrix} = \begin{matrix} 110111 \\ 11110 \end{matrix}$

If $i_2 = 3$ we cannot match $\begin{matrix} w_1 w_2 \\ x_1 x_2 \end{matrix} = \begin{matrix} 110 \\ 1110 \end{matrix}$

If $i_2 = 1$ we have to match $\begin{matrix} w_1 w_2 \\ x_1 x_2 \end{matrix} = \begin{matrix} 11 \\ 111111 \end{matrix}$

We are back to "square one."

We reduce MPCP to PCP

Let MPCP be $A = w_1, w_2, \ldots, w_k$, $B = x_1, x_2, \ldots, x_k$

We construct PCP $A' = y_0, y_1, \ldots, y_{k+1}$, $B' = z_0, z_1, \ldots, z_{k+1}$ as follows:

$y_0 = *y_1$ and $z_0 = z_1$

If $w_i = a_1 a_2 \ldots a_\ell$ then $y_i = a_1 * a_2 * \ldots a_\ell *$

If $x_i = b_1 b_2 \ldots b_p$ then $z_i = *b_1 * b_2 \ldots * b_p$

$y_{k+1} = \$$ and $z_{k+1} = *\$$

Now (A, B) has a solution iff (A', B') has one.

Example:

Let MPCP be

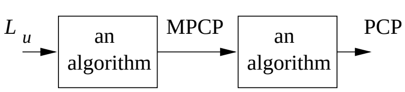
	List A	List B
i	w_i	x_i
1	1	111
2	10111	10
3	10	0

Then we construct PCP

	List A	List B
i	y_i	z_i
0	*1*	*1*1*1
1	1*	*1*1*1
2	1*0*1*1*1*	*1*0
3	1*0*	*
4	$	*$

PCP is undecidable

Given an instance (M, w) of L_u we construct instance (A, B) of MPCP such that $w \in L(M)$ iff (A, B) has a solution.



Partial solutions will consist of strings of the form

$$\#\alpha_1\#\alpha_2\#\alpha_3\cdots$$

where α_1 is the initial configuartion of M on w, and $\alpha_i \vdash \alpha_{i+1}$.

The string from list B will always be one ID ahead of A until M accepts w and A can catch up.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be the TM. WLOG assume that M never prints a blank, and never moves the head left of the initial position.

1. The initial pair

List A	List B
$\#$	$\#q_0w\#$

2. For each $X \in \Gamma$

List A	List B
X	X
$\#$	$\#$

3. $\forall q \in Q \setminus F$, $\forall p \in Q$, $\forall X, Y, Z \in \Gamma$

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

4. $\forall q \in F$, $\forall X, Y \in \Gamma$

List A	List B
XqY	q
Xq	q
qY	q

5. Final pair

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

Example: L_u instance: $(M, 01)$

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

δ	0	1	B
$\rightarrow q_1$	$(q_2, 1, R)$	$(q_2, 0, L)$	$(q_2, 1, L)$
q_2	$(q_3, 0, L)$	$(q_1, 0, R)$	$(q_2, 0, R)$
$\star q_3$	-	-	-

The corresponding MPCP is:

Rule	List A	List B	Source
(1)	#	#$q_1$01	
(2)	0	0	
	1	1	
	#	#	
(3)	$q_1$0	1q_2	from $\delta(q_1, 0) = (q_2, 1, R)$
	0$q_1$1	$q_2$00	from $\delta(q_1, 1) = (q_2, 0, L)$
	1$q_1$1	$q_2$10	from $\delta(q_1, 1) = (q_2, 0, L)$
	0q_1#	$q_2$01#	from $\delta(q_1, B) = (q_2, 1, L)$
	1q_1#	$q_2$11#	from $\delta(q_1, B) = (q_2, 1, L)$
	0$q_2$0	$q_3$00	from $\delta(q_2, 0) = (q_3, 0, L)$
	1$q_2$0	$q_3$10	from $\delta(q_2, 0) = (q_3, 0, L)$
	$q_2$1	0q_1	from $\delta(q_2, 1) = (q_1, 0, R)$
	q_2#	0q_2#	from $\delta(q_2, B) = (q_2, 0, R)$
(4)	0$q_3$0	q_3	
	0$q_3$1	q_3	
	1$q_3$0	q_3	
	1$q_3$1	q_3	
	0q_3	q_3	
	0q_3	q_3	
	1q_3	q_3	
	$q_3$0	q_3	
	$q_3$1	q_3	
(5)	q_3##	#	