BLG311E Formal Languages and Automata

Berk Canberk Tolga Ovatman

2020

Outline I

Outline II

Outline III

Section 1

Introduction, Motivation and Background

Computing Machines

Computer

A computer is a general purpose *machine* which can be programmed to carry out a finite set of arithmetic or logical operations(*computation*).

Machine

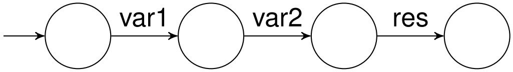
A machine is a tool consisting of one or more generally moving parts that is constructed to achieve a particular goal. However, the advent of electronics technology has led to the development of devices without moving parts that are considered machines.

Abstract Machine

An abstract machine, is a theoretical model of a computer hardware or software system.

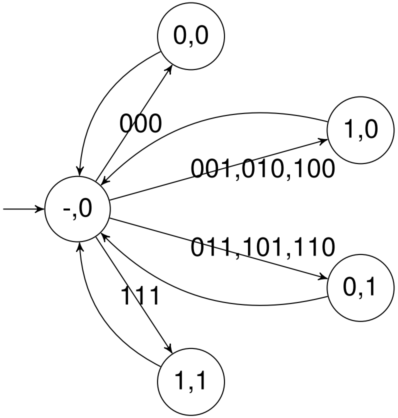
A Very Simple Program

```
 var1 = int(input())
var2 = int(input())
res = var1 + var2
```



A Very Simple Binary Adder

```
with open("input.txt") as file:
    var1 = file.readline().strip()
    var2 = file.readline().strip()
    carry = 0
    res = []
    for i in range(len(var1)-1, 0, -1):
        bit1 = int(var1[i])
        bit2 = int(var2[i])
        if bit1 + bit2 + carry == 0:
            res.insert(0, "0")
        elif bit1 + bit2 + carry == 1:
            res.insert(0, "1")
        elif bit1 + bit2 + carry == 2:
            res.insert(0, "0")
            carry = 1
        else:
            res.insert(0, "1")
            carry = 1
    if carry == 1:
            res.insert(0, "1")
```



Finite State Machine(FSM)

An FSM has a mathmematical model defined by a quintuple $(S, I, O, \delta, \omega)$, where:

- S: Set of states.
- I: Input alphabet (a finite, non-empty set of symbols)
- O: Output alphabet (a finite, non-empty set of symbols)
- δ: The transition function defined on $I \times S \to S$
- ω: The output function defined on either $S \to O$ or $I \times S \to O$

FSM properties

A finite state machine should hold the following properties

1. It has finite input and output alphabets
2. It is deterministic

Deterministic Machine

For a deterministic machine the outcome of a transition from one state to another given a certain input can be predicted for every occurrence. In a deterministic finite state machine, for each pair of state and input symbol there is one and only one transition to a next state.

FSM properties

A finite state machine should hold the following properties

3. It has transducer capability.

Transducer

A transducer machine has two alphabets: an input alphabet and an output alphabet. Transducers are said to be able to *transform* inputs to output.

Transducers

When realizing transducers with digital circuits the concept of discrete-time is used.

Discrete Time

Discrete time is the discontinuity of a function's time domain that results from sampling a variable at a finite interval. One of the fundamental concepts behind discrete time is an implied (actual or hypothetical) system clock.

Transducers

Transducers holds some certain properties in digital discrete-time systems.

- Sequence: Discretization is performed by the sequence order of the labels. n is the length of the sequence.
- $[\Lambda]$: A singleton set which only includes an *empty string*.
- I^*: Set of input sequences : $I^* = [\Lambda] \cup I \cup I^2 \cup \ldots \cup I^n \ldots$
- O^*: Set of output sequences : $O^* = [\Lambda] \cup O \cup O^2 \cup \ldots \cup O^n \ldots$

An example Machine

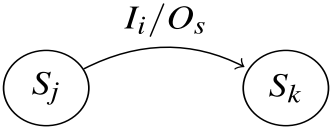
Suppose we want to design a machine that reads numerical codes input from a keypad and accepts *only one* certain key combination. For this particular case$(S, I, O, \delta, \omega)$ can be defined as:

- $I \in \{0, 1, 2, \ldots, 9\}$. We may want to restrict the number of digits to 4, for example $1234 \in I^4$
- S: At each keystroke we need to control if the correct digit is input. In our case $|S| = 5$ or $s_i \in S : 0 \leq i \leq 4$
- $O = \{open, closed\}$ is the output alphabet.
- Our machine is going to accept the code after 4 correct inputs. The final state $F = s_4$. Let's ignore any wrong combinations at this moment for the sake of simplicity.
- Since we only have one correct combination δ can be defined as follows δ :
  $(s_0, 1) \to s_1$
  $(s_1, 9) \to s_2$
  $(s_2, 0) \to s_3$
  $(s_3, 3) \to s_4$
- Our output function ω maps $s_i \to closed$ where $0 \leq i \leq 3$ and $s_4 \to open$

Modeling FSMs

Following elements can be used in modeling FSMs:
Mealy Machine

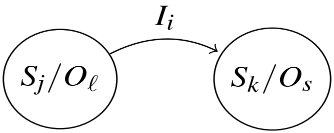
	I_1	$\dots$	I_i	$\dots$	I_m
S_1					
$\dots$					
S_j			S_k/O_s		
$\dots$					
S_n					



$$\begin{tikzcd} S_j \arrow[r, bend left, "I_i/O_s"] & S_k \end{tikzcd}$$

Moore Machine

	I_1	$\dots$	I_i	$\dots$	I_m	O
S_1						O_1
$\dots$						$\dots$
S_j			S_k			O_ℓ
$\dots$						$\dots$
S_n						O_p

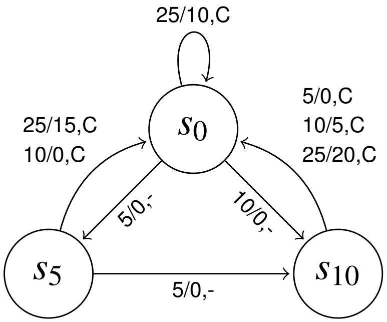


$$\begin{tikzcd} S_j/O_\ell \arrow[r, bend left, "I_i"] & S_k/O_s \end{tikzcd}$$

Example

A coffee vending machine that accepts coins of 5, 10 and 25 cents, gives coffe for 15 cents and returns the change. Let's build Mealy and Moore models of this machine.
Mealy model, $S_x/i,j$: x denotes money input so far, i change return and j holds C for coffee output and $-$ for no output.

State diagram:



State table:

	5	10	25
S_0	$S_5/0,-$	$S_{10}/0,-$	$S_0/10,C$
S_5	$S_{10}/0,-$	$S_0/0,C$	$S_0/15,C$
S_{10}	$S_0/0,C$	$S_0/5,C$	$S_0/20,C$

Example

For Moore model the number of states need to be at least the number of different State/output pairs in the Mealy model. States should not be assigned to the coins input. When no coins are input current state is preserved.

State table:

	5	10	25	Output
S_0	S_5	S_{10}	S_{25}	0,-
S_5	S_{10}	S_{15}	S_{30}	0,-
S_{10}	S_{15}	S_{20}	S_{35}	0,-
S_{15}	S_5	S_{10}	S_{25}	0,C
S_{20}	S_5	S_{10}	S_{25}	5,C
S_{25}	S_5	S_{10}	S_{25}	10,C
S_{30}	S_5	S_{10}	S_{25}	15,C
S_{35}	S_5	S_{10}	S_{25}	20,C

We can map Mealy model transitions to Moore states of this machines as follows

$S_5/0,- \to S_5$

$S_{10}/0,- \to S_{10}$

$S_0/0,C \to S_{15}$

$S_0/5,C \to S_{20}$

$S_0/10,C \to S_{25}$

$S_0/15,C \to S_{30}$

$S_0/20,C \to S_{35}$

Modeling Computation and Computer Programs

!!! SPOILER ALERT !!!

- Any kind of problem that can be handled by computers can be encoded into 0's and 1's
- Any solution that can be executed by a computer can be modeled as a machine
- Using Automata Theory, we can reason about the problems and solutions that can be handled by computers
- We can reason about the relative effectiveness of algorithms, intractability and undecidability.

Formal Languages and Automata

Main three objectives of this course are:

- Understand the concept of an artificial language, how to classify and process artificial languages, properties of different artificial languages.
- Understand the concept of a state machine, how to classify them, how to use them to process languages, properties of different state machine models
- To familiarize with the process of modeling and designing an abstract computing machine.

"A *machine* cannot be understood by stopping it. Understanding must move with the flow of the *machine*, must join it and flow with it."

— The First Law of Mentat from Frank Herbert's Dune

Proving Techniques

- Brute force
  - e.g. Every number from the set {2, 4, 6, . . . , 26} can be written as the sum of at most 3 square numbers.
- Direct proofs
  - e.g. $\forall a \in \mathbb{Z}[3|(a-2) \Rightarrow 3|(a^2-1)]$
- Indirect proofs
  - e.g. $\forall x, y \in \mathbb{N}[x \cdot y > 25 \Rightarrow (x > 5) \vee (y > 5)]$

Proving Techniques

- Proof by contradiction
  - e.g. $\sqrt{2}$ is not a rational number
- Equivalence proofs
  - to prove $P \Leftrightarrow Q$, both $P \Rightarrow Q$ and $Q \Rightarrow P$ must be proven
- Set equivalence
  - To prove that two sets A and B are equal, it is enough to show that $A \subseteq B$ and $B \subseteq A$.

Mathematical Induction

Example

Prove that $1^2 + 2^2 + 3^2 + \ldots + n^2$ is $\frac{n(n+1)(2n+1)}{6}$

- Basis step: Let P(1) be, for n=1
  $1^2 = \frac{1(1+1)(2\cdot 1+1)}{6}$
- Assumption: Let P(n) be, for $n = k$
  $1^2 + 2^2 + 3^2 + \ldots + k^2 = \frac{k(k+1)(2k+1)}{6}$ is true
- In the inductive step we will show that $P(n) \Rightarrow P(n+1)$. Since $P(1)$ is true, we will be able to conclude that $P(n)$ is true for all $n \geq 1$.

Mathematical Induction

- Inductive step: For $n = k + 1$

$$1^2 + 2^2 + 3^2 + \ldots + (k+1)^2 \overset{?}{=} \frac{(k+1)(k+1+1)(2(k+1)+1)}{6}$$

$$1^2 + 2^2 + 3^2 + \ldots + k^2 + (k+1)^2 \overset{?}{=} \frac{(k+1)(k+2)(2k+3)}{6}$$

$$\frac{k(k+1)(2k+1)}{6} + \frac{6(k+1)^2}{6} \overset{?}{=} \frac{2k^3+9k^2+13k+6}{6}$$

$$\frac{2k^3+3k^2+k}{6} + \frac{6(k+1)^2}{6} \overset{?}{=} \frac{2k^3+9k^2+13k+6}{6}$$

$$\frac{2k^3+3k^2+k}{6} + \frac{6k^2+12k+6}{6} \overset{?}{=} \frac{2k^3+9k^2+13k+6}{6}$$

$$\frac{2k^3+9k^2+13k+6}{6} = \frac{2k^3+9k^2+13k+6}{6}$$

Pigeonhole Principle

Pigeonhole Principle

If there are n + 1 pigeons and n pigeonholes, then some pigeonhole must contain at least two pigeons.

Genaralized Pigeonhole Principle

If there are kn+1 pigeons and n pigeonholes, then some pigeonhole must contain at least k + 1 pigeons.

Pigeonhole Principle

Example 1

Given twelve integers, show that two of them can be chosen whose difference is divisible by 11.

Example 2

Twenty-five crates of apples are delivered to a store. The apples are of three different sorts, and all the apples in each crate are of the same sort. Show that among these crates there are at least nine containing the same sort of apple.

Example 3

Show that in any group of five people, there are at least two who have an identical number of friends within the group.

Section 2

Formal Languages

Finite State Machine(FSM)

An FSM has a mathmematical model defined by a quintuple $(S, I, O, \delta, \omega)$, where:

- S: Set of states.
- I: Input alphabet (a finite, non-empty set of symbols)
- O: Output alphabet (a finite, non-empty set of symbols)
- δ: The transition function defined on $I \times S \rightarrow S$
- ω: The output function defined on either $S \rightarrow O$ or $I \times S \rightarrow O$

An example Machine

Suppose we want to design a machine that reads numerical codes input from a keypad and accepts *only one* certain key combination. For this particular case(S, I, O, δ, f) can be defined as:

- $I \in \{0, 1, 2, \ldots, 9\}$. We may want to restrict the number of digits to 4, for example $1234 \in I^4$

Formal Languages

Language

Language may refer either to the specifically human capacity for acquiring and using complex systems of communication, or to a specific instance of such a system of complex communication.

Formal Language

In mathematics, computer science, and linguistics, a formal language is a set of strings of symbols. The alphabet of a formal language is the set of symbols, letters, or tokens from which the strings of the language may be formed which are called words.

Inductively Defined Sets

One way to build a formal language is to inductively define the set of words using a set of symbols.

Inductive Definition

A definition of a term within which the term itself appears, and that is well-founded, avoiding an infinite regress. Also called recursive definition.

Inductively Defined Sets

In order to define a set E inductively, we need to have:

- Basis: An initial set S of elements where $S \subseteq E$. Members of S are called basis elements which are used to derive members of E. An alphabet contains basis elements for natural languages.
- Rules: Functions that are used to transform elements of E into new elements.

$$\begin{aligned} \Omega = \{f_1, f_2, \ldots, f_n\} \wedge \forall f_i \in \Omega \\ f_i : E \times E \times \ldots \times E \rightarrow E \\ \forall x_1, x_2, \ldots, x_p \in E \;\; f_i(x_1, x_2, \ldots, x_p) = X \in E \end{aligned}$$

- Closure: Performing the rules on members of E always produces a member of E. In another way $\Omega(E) \subseteq E$

Inductively Defined Sets

In order to define a set E inductively, we need to have:

- Induction:

$$
\begin{aligned}
&S_0 = S\\
&S_1 = \Omega(S_0) \cup S_0 \text{ Rules applied}\\
&S_2 = \Omega(S_1) \cup S_1\\
&\dots\\
&S_{i+1} = S_i \cup \Omega(S_i) = S_i \subseteq E \text{ (closure)}
\end{aligned}
$$

  i can be finite or infinite.
- Element height: Defined as the cardinality of the set in which an element is derived for the first time. $H(x) = min\{i|x \in S_i\}$

Inductively Defined Sets

Let's build the set of even numbers inductively:

i. $0 \in E$ basis element
ii. $n \in E \Rightarrow (n+2) \in E$
iii. No other number can be identified as even number, apart from the rule number (ii)

Section 3

Alphabets and Languages

Definitions

- Alphabet: A finite, non-empty set of symbols or characters. Denoted as Σ
- Word: A finite array or string from Σ
- Word Length: Number of symbols in a word.
- Empty string: A word of length 0. Denoted as Λ or ε.

Word concatenation

Concatenation is performed by joining two character strings end-to-end.

$$(x = a_1 a_2 \ldots a_n) \wedge (y = b_1 b_2 \ldots b_m) \Rightarrow xy = a_1 a_2 \ldots a_n b_1 b_2 \ldots b_m (x \& y)$$

Identity element of concatenation is Λ
$x = \Lambda \Rightarrow xy = y$
$y = \Lambda \Rightarrow xy = x$

We can build a monoid set of words(Σ^*) from the alphabet using concatenation operation having an empty string as an identity element and holding associativity property.

Reverse of a string

$(baba)^R = abab$
$(ana)^R = ana$

We can define the operation using induction:

1. $|w| = 0 \Rightarrow w^R = w = \Lambda$
2. Assume we have two strings w and u.
   $|u| = n$ $\qquad |w| = n+1$
   $n \in \mathbb{N}$. Following these assumptions
   $(w = ua) \wedge (a \in \Sigma) \Rightarrow w^R = au^R$

For example:

$$|w| = 1 \Rightarrow w = \Lambda a$$
$$w^R = a\Lambda = w$$
$$|w| = 2 \Rightarrow w = ua(u = b)$$
$$w^R = au^R = au = ab$$
$$|w| = 3 \Rightarrow w = ua, u = cb$$
$$w^R = au^R = abc$$

Reverse of a string

Theorem

$(wx)^R = x^R \cdot w^R \wedge |x| = n, |w| = m \wedge m, n \in \mathbb{N}$

Proof.

Basis step:

$|x| = 0 \Rightarrow x = \Lambda$

$(wx)^R = (w\Lambda)^R = w^R = \Lambda w^R = \Lambda^R w^R = x^R w^R$

Inductive step: $|x| \leq n \Rightarrow (wx)^R = x^R w^R$

For $|x| = n + 1$

$(x = ua) \wedge (|u| = n) \wedge (a \in \Sigma) \wedge (x^r = au^r)$

$(wx)^R = (w(ua))^R = ((wu)a)^R = a(u^R w^R) = au^R w^R = x^R w^R$

□

For example (snow ball)^R= (ball)^R(snow)^R

Σ^+ denotes the set of non-empty strings over the Σ alphabet

i $a \in \Sigma \Rightarrow a \in \Sigma^+$

ii $(x \in \Sigma^+ \wedge a \in \Sigma) \Rightarrow ax \in \Sigma^+$

iii Σ^+ doesn't contain any elements other than the ones that can be constructed by applying i and ii finitely.

For example:
$\Sigma = \{a, b\} \Rightarrow \Sigma^+ = \{a, b, aa, ba, ab, bb, aaa, aab, \ldots\}$.

Σ^* denotes the set of all strings over the Σ alphabet

i $\Lambda \in \Sigma^*$

ii $(x \in \Sigma^* \wedge a \in \Sigma) \Rightarrow ax \in \Sigma^*$

iii Σ^* doesn't contain any elements other than the ones that can be constructed by applying i and ii finitely.

For example:

$\Sigma = \{a, b\} \Rightarrow \Sigma^* = \{\Lambda, a, b, aa, ba, ab, bb, aaa, aab, \ldots\}$

$\Sigma = \{0, 1\} \Rightarrow \Sigma^* = \{\Lambda, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$

Assuming $(x \in \Sigma^*) \wedge (\forall n \in \mathbb{N})$, a string's nth power($x^n$) can be formally defined as :

1. $x^0 = \Lambda$
2. $x^{n+1} = x^n \cdot x = (x^n \& x)$

For example:
$\Sigma = \{a, b\} \;\; x = ab$
$x^0 = \Lambda, \;\; x^1 = ab, \;\; x^2 = abab, \;\; x^3 = ababab$

or
$\{a^n b^n | n \geq 0\}$ defines the set: $\{\Lambda, ab, aabb, aaabbb, \dots\}$

Let Σ be a finite alphabet.

A language over Σ is a subset of Σ^*.

The rules of choosing a subset from Σ^* can be performed by using a grammar.

For example:
$\{a^m b^n | m, n \in \mathbb{N}\}$ is a language defined over $\{a, b\}$.

This language cannot contain any other symbols than a and b.

In this language b always succeeds a. For instance ba doesn't belong to this language.

Multiplication of Languages

Let A and B be languages defined over Σ. Cartesian product $A \times B$ produces a new language.
$AB = \{xy | x \in A \wedge y \in B\}$
$AB = \{z | z = xy \wedge x \in A \wedge y \in B\}$

For example:
Let $\Sigma = \{a, b\}$ be the alphabet
Let $A = \{\Lambda, a, ab\}$ and $B = \{a, bb\}$ be the languages
$AB = \{a, bb, aa, abb, aba, abbb\}$
$BA = \{a, aa, aab, bb, bba, bbab\}$
$AB \neq BA$

Theorems

Let $\varnothing$ denote empty language; A,B,C,D denote different languages defined over the alphabet Σ.

1. $A\varnothing = \varnothing A = \varnothing$
2. $A\{\Lambda\} = \{\Lambda\}A = A$
3. $(AB)C = A(BC)$
4. $(A \subset B) \wedge (C \subset D) \Rightarrow AC \subset BD$
5. $A(B \cup C) = AB \cup AC$
6. $(B \cup C)A = BA \cup CA$
7. $A(B \cap C) \subset AB \cap AC$
8. $(B \cap C)A \subset BA \cap CA$

Selected Proofs.

$$A\varnothing = \varnothing A = \varnothing$$

$A\varnothing = \{xy | (x \in A) \land (y \in \varnothing)\}$ doesn't hold since $\forall y; y \notin \varnothing$.

Since there doesn't exist any x, y couples satisfying the condition $A\varnothing = \varnothing$ □

Selected Proofs.

$$(A \subset B) \wedge (C \subset D) \Rightarrow AC \subset BD$$

We are going to use a direct proof. We assume $(A \subset B) \wedge (C \subset D)$ and $z \in AC$

$$z = xy \Leftrightarrow x \in A \wedge y \in C$$

$$((A \subset B) \wedge (x \in A) \Rightarrow x \in B) \wedge ((C \subset D) \wedge (y \in C) \Rightarrow y \in D)$$

$$((x \in B) \wedge (y \in D)) \Leftrightarrow xy \in BD$$

$$(xy \in AC \Rightarrow xy \in BD) \Leftrightarrow AC \subset BD$$

$\square$

Selected Proofs.

$$A(B \cap C) \subset AB \cap AC$$

$$\begin{aligned}
A(B \cap C) &\Rightarrow \forall z((z = xy) \wedge (x \in A) \wedge (y \in B \cap C)) \\
(x \in A) \wedge (y \in B \cap C) &\Leftrightarrow ((x \in A) \wedge (y \in B) \wedge (y \in C)) \\
((x \in A) \wedge (y \in B) \wedge (y \in C)) &\Leftrightarrow ((x \in A) \wedge (y \in B) \wedge (x \in A) \wedge (y \in C)) \\
((x \in A) \wedge (y \in B) \wedge (x \in A) \wedge (y \in C)) &\Leftrightarrow (xy \in AB) \wedge (xy \in AC) \\
(xy \in AB) \wedge (xy \in AC) &\Rightarrow xy \in (AB \cap AC) \\
\forall z(z \in A(B \cap C) &\Rightarrow z \in AB \cap AC)
\end{aligned}$$

$\square$

Let's give a counter example to show that $AB \cap AC \subseteq A(B \cap C)$ might not hold all the time.

$A = \{a^n | n \in \mathbb{N}\}$
$B = \{a^n b^n | n \in \mathbb{N}\}$
$C = \{b^n | n \in \mathbb{N}\}$

Let's take $z = a^5 b^2$. $z \in AB \cap AC$
However $B \cap C = \{\Lambda\}$ therefore
$z \notin A(B \cap C)$.

$z = a^5 b^2 = a^3 a^2 b^2$
$z \in AB \cap AC$
$z \notin A(B \cap C)$ because $B \cap C = \{\Lambda\}$

A^n : Let language A be defined over Σ

i $A^0 = \{\Lambda\}$

ii $A^{n+1} = A^n A; \forall n \in \mathbb{N}$

For example:

$\Sigma = \{a, b\}$
$A = \{\Lambda, a, b\}$
$A^1 = A$
$A^2 = \{\Lambda, a, b, aa, ab, ba, bb\}$

Theorems

Let A and B denote different languages defined over the alphabet Σ.

1. $A^m A^n = A^{m+n}$
2. $(A^m)^n = A^{mn}$
3. $A \subset B \Rightarrow A^n \subset B^n \ldots$

Proof.

$$A^m A^n = A^{m+n}$$

We are going to use induction.

For the base case, we use the previous definition:
$n = 1 : A^m A^1 = A^{m+1}$

Inductive step: $A^m A^{n+1} = A^m A^n A^1$
Concat op. is associative: $A^m (A^n A^1) = (A^m A^n) A^1 = A^{m+n} A^1$ $\square$

Proof.

$$(A^m)^n = A^{mn}$$

We are going to use induction.

For the base case $n = 1 : A^m = A^m$

Inductive step: $(A^m)^{n+1} = (A^m)^n (A^m)^1 = A^{mn+m} = A^{m(n+1)}$

$\square$

Proof.

$$A \subset B \Rightarrow A^n \subset B^n \ldots$$

Let's remember $A^2 = A \times A, B^2 = B \times B$ and
$A \subset B \Rightarrow (\forall x(x \in A \rightarrow x \in B))$

For $n = 2$:
$A^2 = \{xy | x \in A \wedge y \in A\}$
$A \subset B \Rightarrow (x \in A \rightarrow x \in B)$
$A \subset B \Rightarrow (y \in A \rightarrow y \in B)$

$(\forall xy(xy \in A^2 \rightarrow xy \in B^2)) \Rightarrow A^2 \subset B^2$
Proof continues similarly for $n := n + 1$ □

Star closure (Kleene Star, Kleen Closure)

Positive Closure

$A^+ : \bigcup_{n=1}^{\infty} A^n = A \cup \ldots$

Star Closure

$A^* : \bigcup_{n=0}^{\infty} A^n = A^0 \cup A \cup \ldots$

Theorems

Let A and B denote different languages defined over the alphabet Σ and let $n \in \mathbb{N}$.

1. $A^* = \Lambda \cup A^+$. Derived from the definition
2. $A^n \subseteq A^*, n \geq 0$. Derived from the definition
3. $A^n \subseteq A^+, n \geq 1$. Derived from the definition
4. $A \subseteq AB^*$ Proof tip: $A^0 \subseteq B^* \Rightarrow A \subseteq AB^*$

Theorems

Let A and B denote different languages defined over the alphabet Σ and let $n \in \mathbb{N}$.

5. $A \subseteq B^*A$
6. $A \subseteq B \Rightarrow A^* \subseteq B^*$. Can be proven by using $A \subseteq B \Rightarrow A^n \subseteq B^n$. Once true for all n, then it is true for union of all n.
7. $A \subseteq B \Rightarrow A^+ \subseteq B^+$
8. $AA^* = A^*A = A^+$
9. $\Lambda \in A \Leftrightarrow A^+ = A^*$
10. $(A^*)^* = A^*A^* = A^*$
11. $(A^*)^+ = (A^+)^* = A^*$
12. $A^*A^+ = A^+A^* = A^+$
13. $(A^*B^*)^* = (A \cup B)^* = (A^* \cup B^*)^*$.

Proof.

Selected Proofs

$$AA^* = A^*A = A^+$$

$$AA^* = A^+ \Rightarrow A(A^0 \cup A^1 \cup \ldots) = A \cup A^2 \cup \ldots = A^+ \qquad \square$$

Selected Proofs

$$\Lambda \in A \Leftrightarrow A^{+} = A^{*}$$

First let's show $\Lambda \in A \Rightarrow A^{+} = A^{*}$

$\Lambda \in A \Rightarrow A \cup \{\Lambda\} = A$
$A \cup A^{0} = A$

$A^{+} = A \cup \{\bigcup_{n=2}^{\infty} A^{n}\} = A^{0} \cup A \cup \{\ldots\} = A^{*}$

Selected Proofs.

$$\Lambda \in A \Leftrightarrow A^+ = A^*$$

Secondly $A^+ = A^* \Rightarrow \Lambda \in A$
If $A \cup A^2 \cup \ldots = A^0 \cup A \cup \ldots$
$A^0 \subseteq A \cup A^2 \cup \ldots$
This inclusion can be interpreted in two different ways:

- i $\Lambda \in A \Rightarrow A^0 \subseteq A$
- ii $\Lambda \notin A \Rightarrow \exists i, \Lambda \in A^i, i \in (\mathbb{N}^+ - 1)$

However $\Lambda \notin A \Rightarrow \forall x \in A^i(|x| \geq i)$
on the contrary $|\Lambda| = 0 \Rightarrow \Lambda \notin A^i \wedge i \geq 2.$
We have a contradiction proving ii wrong. Therefore i must be true.

$\square$

Selected Proofs.

$$(A^*B^*)^* = (A \cup B)^*$$

We need to prove both $(A^*B^*)^* \subseteq (A \cup B)^*$ and $(A \cup B)^* \subseteq (A^*B^*)^*$

$$(A \subseteq A \cup B) \wedge (B \subseteq A \cup B)$$
$$(A^* \subseteq (A \cup B)^*) \wedge (B^* \subseteq (A \cup B)^*)$$
$$A^*B^* \subseteq (A \cup B)^*$$
$$(A^*B^*)^* \subseteq ((A \cup B)^*)^*$$
$$(A^*B^*)^* \subseteq (A \cup B)^*$$

□

Selected Proofs.

$$(A^*B^*)^* = (A \cup B)^*$$

We need to prove both $(A^*B^*)^* \subseteq (A \cup B)^*$ and $(A \cup B)^* \subseteq (A^*B^*)^*$
$A \subseteq A^* \subseteq A^*B^*$ and $B \subseteq B^* \subseteq A^*B^*$

It is possible to write statements above since A^* and B^* contains Λ
$(A \cup B \subseteq A^*B^*) \Rightarrow (A \cup B)^* \subseteq (A^*B^*)^*$
$(A^*B^*)^* \subseteq (A \cup B)^*$ and $(A \cup B)^* \subseteq (A^*B^*)^*$.

$\square$

Arden's Theorem

Let A and B denote subsets of Σ^* and let $\Lambda \notin A$.
The only solution of $X = AX \cup B$ is $X = A^*B$

$$
\begin{aligned}
&X = AX \cup B \\
&X = A(AX \cup B) \cup B \\
&X = A^2X \cup AB \cup B \\
&X = A^2(AX \cup B) \cup AB \cup B \\
&X = A^3X \cup A^2B \cup AB \cup B \\
\ldots& \\
&X = \ldots \cup A^3B \cup A^2B \cup AB \cup B \\
&X = (\ldots \cup A^3 \cup A^2 \cup A^1 \cup A^0)B \\
&X = A^*B
\end{aligned}
$$

Demonstration

$X = A^*B$ is a solution
Considering $\{\Lambda\}B = B$,

$$\begin{aligned} A^*B &= AA^*B \cup B \\ &= A^+B \cup B \\ &= (A^+ \cup \{\Lambda\})B \\ &= A^*B \end{aligned}$$

Section 4

Languages and Grammars

Formal Grammar

A formal grammar is a set of formation rules for strings in a formal language. The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax.

In our context
Σ denotes the alphabet
Σ^* is the set of all words that can be constructed using Σ
Grammar is the set of formation rules to construct a subset of Σ^*
For example:
To construct arithmetic expressions following can be used:
$\mathbb{Z}\ +\ -\ \times\ /\ (\ )$. All the well-formed arithmetic expressions are meaningful except division by zero.
$(((2-1)/3)+4\times 6)$ is a well-formed and meaningful arithmetic expression. $2+(3/(5-(10/2)))$ is well-formed as well but not meaningful because of division by zero.

The Syntax of Grammars

In the classic formalization of generative grammars first proposed by Noam Chomsky in the 1950s. A grammar G(also called a phase structure grammar) is formally defined as the tuple $(N, \Sigma, n_0, \mapsto)$:

- N is a set of *nonterminal symbols*[1].
- Σ is a set of *terminal symbols* that is disjoint from N
- n_0 is the start symbol
- $\mapsto$ is the set of production rules.
  If we call $V = N \cup \Sigma$, $\mapsto$ is a relation defined over V^*.
  It has the structure: $V^*NV^* \to V^*$
  where $*$ is a Kleene star operator.

Formal grammars are also sometimes denoted as phase structure grammars in the literature.

[1] A terminal symbol cannot be replaced with another set of symbols

Example 1

$S =$Harry,Sally,runs,swims,fast,often,long

$N =$sentence,verbal sentence,noun,verb,adverb

$n_0 =$sentence

Grammar:

sentence $\mapsto$ noun | verbal sentence

noun $\mapsto$ Harry

noun $\mapsto$ Sally

verbal sentence $\mapsto$ verb | adverb

adverb $\mapsto$ fast

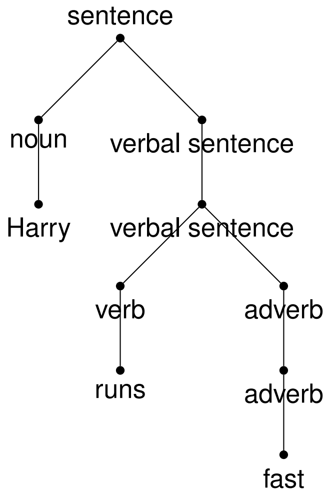
adverb $\mapsto$ often

adverb $\mapsto$ long

verb $\mapsto$ runs

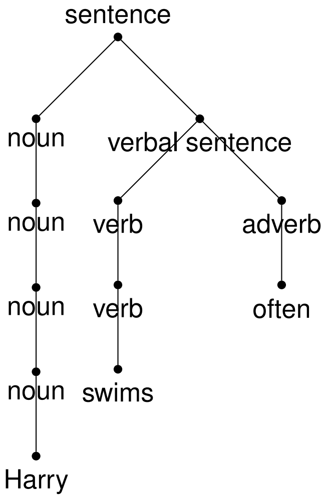
verb $\mapsto$ swims

We can apply these rules in one of two ways to derive a well-formed syntax. We can either successively expand the leftmost expression first or . . .



S =Harry,Sally,runs,swims,fast,often,long
N =sentence,verbal
sentence,noun,verb,adverb
n_0 =sentence
Grammar:
sentence $\mapsto$ noun, verbal sentence
noun $\mapsto$ Harry
noun $\mapsto$ Sally
verbal sentence $\mapsto$ verb, adverb
adverb $\mapsto$ fast
adverb $\mapsto$ often
adverb $\mapsto$ long
verb $\mapsto$ runs
verb $\mapsto$ swims

...or we can successively expand the rightmost expression first



S =Harry,Sally,runs,swims,fast,often,long

N =sentence,verbal sentence,noun,verb,adverb

n_0 =sentence

Grammar:

sentence $\mapsto$ noun, verbal sentence

noun $\mapsto$ Harry

noun $\mapsto$ Sally

verbal sentence $\mapsto$ verb, adverb

adverb $\mapsto$ fast

adverb $\mapsto$ often

adverb $\mapsto$ long

verb $\mapsto$ runs

verb $\mapsto$ swims

and derive a canonical parse tree.

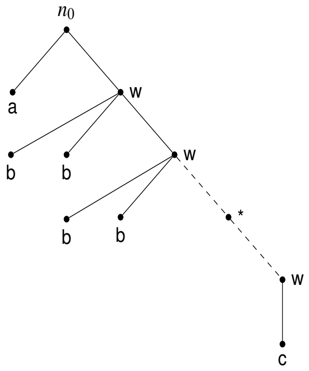
Parsing

Parsing, or, syntactic analysis, is the process of analyzing a text, made of a sequence of tokens (for example, words), to determine its grammatical structure with respect to a given formal grammar.

Example 2

$S = a, b, c$
$N = n_0, w$

$$\begin{aligned} \mapsto = \{ & n_0 \rightarrow aw \mid \\ & w \rightarrow bbw \mid \\ & w \rightarrow c \} \end{aligned}$$



Example 2

$S = a, b, c$
$N = n_0, w$

$$\begin{aligned} \mapsto = \{n_0 &\to aw \mid \\ w &\to bbw \mid \\ w &\to c\} \end{aligned}$$

$L(G) = \{a\} \cdot \{bb\}^* \cdot \{c\}$
$= a(bb)^*c$
$\{a(bb)^n c | n \in \mathbb{N}\}$

Example 3

$S = a, b, c$
$N = n_0, w$

$$\begin{aligned} \mapsto = \{ & n_0 \to a n_0 b \ \mid \\ & n_0 b \to bw \ \mid \\ & abw \to c \} \end{aligned}$$

$$\begin{aligned} n_0 &\Rightarrow a n_0 b \\ a n_0 b &\Rightarrow a(a n_0 b) b \Rightarrow a^n n_0 b^n \\ a^n (n_0 b) b^{n-1} &\Rightarrow a^n bwb^{n-1} \\ a^{n-1}(abw) b^{n-1} &\Rightarrow a^{n-1} c b^{n-1} \\ n_0 &\Rightarrow^* a^{n-1} c b^{n-1} \end{aligned}$$

Or in a general form: $L(G) = \{a^m c b^m | m \in \mathbb{N}\}$

Example 4

$S = a, b, c$

$N = n_0, A, B, C$

$\mapsto = \{$

$n_0 \rightarrow A$

$A \rightarrow aABC$

$A \rightarrow abC$

$CB \rightarrow BC$

$bB \rightarrow bb$

$bC \rightarrow bc$

$cC \rightarrow cc$

$\}$

n_0

$\underline{\mathrm{A}}$

$\mathrm{a}\underline{\mathrm{A}}\mathrm{BC}$

$\mathrm{aa}\underline{\mathrm{A}}\mathrm{BCBC}$

$\mathrm{aaab}\underline{\mathrm{CB}}\mathrm{CBC}$

$\mathrm{aaab}\underline{\mathrm{BC}}\underline{\mathrm{CB}}\mathrm{C}$

$\mathrm{aaab}\underline{\mathrm{bCB}}\mathrm{CC}$

$\mathrm{aaab}\underline{\mathrm{bB}}\mathrm{CCC}$

$\mathrm{aaabb}\underline{\mathrm{bC}}\mathrm{CC}$

$\mathrm{aaabbb}\underline{\mathrm{cC}}\mathrm{C}$

$\mathrm{aaabbbc}\underline{\mathrm{cC}}$

$\mathrm{aaabbbccc}$

$a^3b^3c^3$

$L(G) = \{a^k b^k c^k | k \in \mathbb{N}^+\}$

Section 5

Chomsky Hierarchy of Grammars

Chomsky Hierarchy

The Chomsky hierarchy is a containment hierarchy of classes of formal grammars. For a formal grammar $G = \{N, \Sigma, n_0, \mapsto\}$

Type 0 grammars

- Type 0 grammars (unrestricted grammars) include all formal grammars.
- They generate exactly all languages that can be recognized by a Turing machine.
- These languages are also known as the recursively enumerable languages.
- Example 3 conforms to a Type 0 grammar.

Chomsky Hierarchy

The Chomsky hierarchy is a containment hierarchy of classes of formal grammars. For a formal grammar $G = \{N, \Sigma, n_0, \mapsto\}$

Type 1 grammars

- Type 1 grammars (context-sensitive grammars) have transformation rules s.t. for $w_1 \rightarrow w_2$ rule $|w_1| \leq |w_2|$ should hold.
- Context sensitivity follows the rules having the form $lwr \rightarrow lw'r$
- The languages described by these grammars are exactly all languages that can be recognized by a linear bounded automaton.
- Example 4 conforms to a Type 1 grammar.

Chomsky Hierarchy

The Chomsky hierarchy is a containment hierarchy of classes of formal grammars. For a formal grammar $G = \{N, \Sigma, n_0, \mapsto\}$

Type 2 grammars

- Type 2 grammars (context-free grammars) generate the context-free languages.
- These are defined by rules of the form $A \to \gamma$ with a nonterminal(A) and a string of terminals and nonterminals (γ).
- These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton.
- Context-free languages are the theoretical basis for the syntax of most programming languages.
- Example 1 conforms to a Type 2 grammar.

Chomsky Hierarchy

Type 3 grammars

- Type 3 grammars (regular grammars) generate the regular languages.
- Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a number of terminals, possibly followed by a single nonterminal.
- These languages are exactly all languages that can be decided by a finite state automaton.
- Additionally, this family of formal languages can be obtained by regular expressions.
- Regular languages are commonly used to define search patterns.
- Example 2 conforms to a Type 3 grammar.

Only Type-2 and Type-3 grammars have syntax trees.

$n_0 \rightarrow \Lambda$ rule can be added to any type to support empty sentences

Type 3 $\subseteq$ Type 2 $\subseteq$ Type 1 $\subseteq$ Type 0

Chomsky Hierarchy

Type	Language (Grammars)	Form of Productions in Grammar	Accepting Device
0	Recursively enumerable (unrestricted)	$\alpha \to \beta$ $(\alpha, \beta \in (N \cup \Sigma)^*,$ α contains a variable$)$	Turing machine
1	Context-sensitive	$\alpha \to \beta$ $(\alpha, \beta \in (N \cup \Sigma)^*, \lvert\beta\rvert \geq \lvert\alpha\rvert,$ α contains a variable$)$	Linear-bounded automaton
2	Context-free	$A \to \alpha$ $(A \in N, \alpha \in (N \cup \Sigma)^*)$	Pushdown automaton
3	Regular	$A \to \sigma B, A \to \sigma$ $(A, B \in N, \sigma \in \Sigma^*)$	Finite automaton

Backus-Naur Form

BNF

BNF is a notation technique for context-free grammars, often used to describe the syntax of languages used in computing, such as computer programming languages, instruction sets and communication protocols. It can be applied wherever exact descriptions of languages are needed. In the syntax of BNF

- $\mapsto$ is denoted as ::=
- non-terminals denoted inside < >
- terminals denoted as they are
- repetitions are separated by a pipe |

Backus-Naur Form

A Type 0 Grammar

$$
\begin{aligned}
<n_0> &::= a <n_0> b \\
<n_0> b &::= b <w> \\
ab <w> &::= c
\end{aligned}
$$

Backus-Naur Form

A Type 1 Grammar

$$
\begin{aligned}
<n_0> &::= <A> \\
<A> &::= a<A><B><C> \mid ab<C> \\
<C><B> &::= <B><C> \\
b<B> &::= bb \\
b<C> &::= bc \\
c<C> &::= cc
\end{aligned}
$$

Backus-Naur Form

A Type 2 Grammar

$$
\begin{aligned}
< \text{sentence} > &::=< \text{noun} >< \text{verbal sentence} > \\
< \text{noun} > &::= \text{Harry} \mid \text{Sally} \\
< \text{verbal sentence} > &::=< \text{verb} >< \text{adverb} > \\
< \text{verb} > &::= \text{runs} \mid \text{swims} \\
< \text{adverb} > &::= \text{fast} \mid \text{often}|\text{long}
\end{aligned}
$$

Backus-Naur Form

A Type 3 Grammar

$$
\begin{aligned}
< n_0 > &::= a < w > \\
< w > &::= bb < w > \quad | \; c
\end{aligned}
$$

Rules of the form $w \rightarrow bbw$ are called recursive rules. If a recursive rule's non-terminal symbol is at the rightmost side it is called a normal rule.

Section 6

Regular Languages and Regular Expressions

Regular Expression

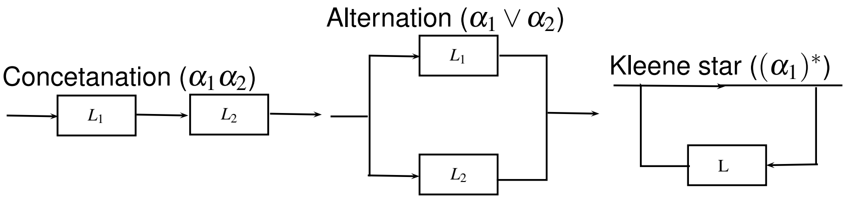
Regular expressions (abbrv: regex, regexp) provide a concise and flexible means to "match" (specify and recognize) strings of text, such as particular characters, words, or patterns of characters.
A regular expression can be defined over an alphabet Σ by induction:

1. Each and every element of Λ and Σ is a regular expression
   - $L(\Lambda) = \{\Lambda\}$
   - $L(a) = \{a\} \ \forall a \in \Sigma$
   - $L(\varnothing) = \varnothing$
2. Concetanation operation($\cdot$) on two regular expressions produce another regular expression
   $L(\alpha\beta) = L(\alpha)L(\beta)$
3. Alternation operation($\vee$) on two regular expressions produce another regular expression
   $L(\alpha \vee \beta) = L(\alpha) \cup L(\beta)$
4. Kleene star operation($*$) on a regular expression produce another regular expression.
   $L(\alpha^*) = (L(\alpha))^*$

Theorem

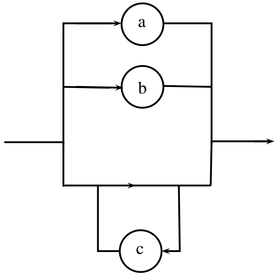
Let L be a language described over S s.t. $L \subseteq S^*$. L is called a regular language if it conforms to a regular grammar G. In other words $L = L(G)$.
Regular expressions describe regular languages in formal language theory. There exists an isomorphism between the described language and the regular expression. They have the same expressive power as regular grammars.

Syntax diagrams can be used define regular expressions. The following three diagrams describe basic properties of regular expressions.

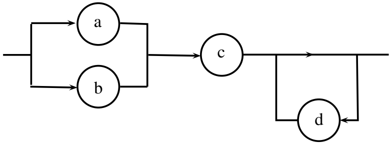

Concetanation $(\alpha_1 \alpha_2)$
L_1
L_2
Alternation $(\alpha_1 \vee \alpha_2)$
L_1
L_2
Kleene star $((\alpha_1)^*)$
L

Following are some additional examples on syntax diagrams.

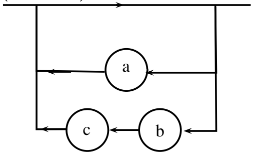
$a \vee b \vee c^*$



$(a \vee b)cd^*$



$(a \vee bc)^*$



The basis regular expression syntax only covers these operators. Operators like numbered repetitions (e.g. $a^n b^n$) can be used to extend the type of language being represented.

For further practice check out the website Debuggex[2] and Regex101[3].

[2]https://www.debuggex.com

[3]https://regex101.com/

Precedence of Regular Expressions

Precedence of operators are defined from highest to lowest as

- Kleene star (*) and positive closure ($^+$)
- Concatenation ($\cdot$), may be omitted
- Alternation ($\vee$), sometimes shown with ($+$) or ($|$)

Algebraic Laws of Regular Expressions

- Commutativeness
  - $L_1 \vee L_2 = L_2 \vee L_1$
- Association
  - $(L_1 \vee L_2) \vee L_3 = L_1 \vee (L_2 \vee L_3)$
  - $(L_1 L_2) L_3 = L_1 (L_2 L_3)$
- Identity
  - $L \vee \varnothing = L$
  - $\Lambda L = L \Lambda = L$
- Absorbing
  - $L\varnothing = \varnothing$

Algebraic Laws of Regular Expressions

- Distribution
  - $L_1(L_2 \vee L_3) = L_1L_2 \vee L_1L_3$
  - $(L_1 \vee L_2)L_3) = L_1L_3 \vee L_2L_3$
- Idempotency
  - $L_1 \vee L_1 = L_1$
- Kleene Closure
  - $(L^*)^* = L^*$
  - $(\varnothing)^* = \Lambda$
  - $\Lambda^* = \Lambda$
  - $L^+ = LL^*$
  - $L? = \Lambda \vee L$ we will prefer the latter notation

Section 7

Context Free Languages and Parsing

Context Free Languages

- A language class larger than the class of regular languages
- Generated by context free grammars
- Can you provide a Type-3 grammar to produce palindromes from $\Sigma = \{a, b\}$
- Can you provide a Type-3 grammar to produce parenthesized expressions from $\Sigma = \{(, a, b,)\}$
- Think about HTML tags. Nested scopes in programming languages. If-then-else structures.

CFG for $\{0^n 1^n | n \geq 1\}$

$$< S > ::= 0 < S > 1 \ | \ 01$$

CFG for well-formed parantheses

$$< S > ::= < S >< S > \ | \ (< S >) \ | \ ()$$

CFG for $\{0^m 1^n | m \geq n\}$

$$< S > ::= 0 < S > 1 \ | \ < A >$$
$$< A > ::= 0 < A > \ | \ \Lambda$$

CFG for simple arithmetic expressions

$$< E > ::= < E > + < E > \ | \ < E > \times < E > \ | (< E >) | \ < F >$$
$$< F > ::= x < F > \ | y < F > \ | 0 < F > \ | 1 < F >$$
$$< F > ::= x \,|\, y \,|\, 0 \,|\, 1$$

Leftmost Derivations

$$
\begin{aligned}
&<E> ::= <E> + <E> \mid <E> \times <E> \mid (<E>) \mid <F> \\
&<F> ::= x <F> \mid y <F> \mid 0 <F> \mid 1 <F> \\
&<F> ::= x \mid y \mid 0 \mid 1
\end{aligned}
$$

$$E \Rightarrow^* x \times (xy + 10)$$

$$
\begin{array}{ll}
<E> & \Rightarrow x \times (<F> + <E>) \\
\Rightarrow <E> \times <E> & \Rightarrow x \times (x <F> + <E>) \\
\Rightarrow <F> \times <E> & \Rightarrow x \times (xy + <E>) \\
\Rightarrow x \times <E> & \Rightarrow x \times (xy + <F>) \\
\Rightarrow x \times (<E>) & \Rightarrow x \times (xy + 1 <F>) \\
\Rightarrow x \times (<E> + <E>) & \Rightarrow x \times (xy + 10)
\end{array}
$$

Rightmost Derivations

$$<E> ::= <E> + <E> \mid <E> \times <E> \mid (<E>) \mid <F>$$
$$<F> ::= x<F> \mid y<F> \mid 0<F> \mid 1<F>$$
$$<F> ::= x \mid y \mid 0 \mid 1$$

$$E \Rightarrow^* x \times (xy + 10)$$

$$<E>$$
$$\Rightarrow <E> \times <E>$$
$$\Rightarrow <E> \times (<E>)$$
$$\Rightarrow <E> \times (<E> + <E>)$$
$$\Rightarrow <E> \times (<E> + <F>)$$
$$\Rightarrow <E> \times (<E> + 1<F>)$$
$$\Rightarrow <E> \times (<E> + 10)$$
$$\Rightarrow <E> \times (<F> + 10)$$
$$\Rightarrow <E> \times (x<F> + 10)$$
$$\Rightarrow <E> \times (xy + 10)$$
$$\Rightarrow <F> \times (xy + 10)$$
$$\Rightarrow x \times (xy + 10)$$

Sentential Forms

- For a grammar G, with start symbol S, any string α such that $S \Rightarrow^* \alpha$ is called a sentential form.
- If α contains only terminals then α is called a sentence in $L(G)$.
- If α contains one or more non-terminals, it is just a sentential form.
- A *left-sentential form* occurs during the leftmost derivation of a sentence
- A *right-sentential form* occurs during the rightmost derivation of a sentence

Sentential Forms

$$<E> ::= <E> + <E> \mid <E> \times <E> \mid (<E>) \mid <F>$$
$$<F> ::= x<F> \mid y<F> \mid 0<F> \mid 1<F>$$
$$<F> ::= x \mid y \mid 0 \mid 1$$

$$\begin{aligned}&<E>\\ &\Rightarrow <E> \times <E>\\ &\Rightarrow <E> \times (<E>)\\ &\Rightarrow <E> \times (<E> + <E>)\\ &\Rightarrow <E> \times (<F> + <E>)\\ &\Rightarrow <E> \times (1 + <E>)\end{aligned}$$

$<E> \times (1 + <E>)$ is a sentential form but it is neither rightmost nor leftmost

$$\begin{aligned}&<E>\\ &\Rightarrow <E> \times <E>\\ &\Rightarrow <F> \times <E>\\ &\Rightarrow x \times (<F> + <E>)\end{aligned}$$

$x \times (<F> + <E>)$ is a left sentential form

$$\begin{aligned}&<E>\\ &\Rightarrow <E> \times <E>\\ &\Rightarrow <E> \times (<E>)\\ &\Rightarrow <E> \times (<E> + <E>)\end{aligned}$$

$<E> \times (<E> + <E>)$ is a right sentential form

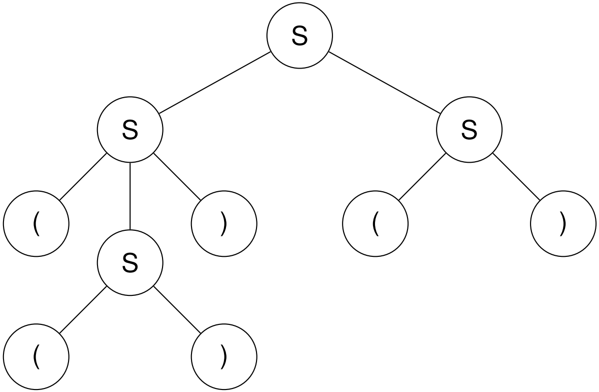
Parsing and Parse Trees

- Parsing, syntax analysis, or syntactic analysis is the process of analysing a string of symbols, conforming to the rules of a formal grammar.
- A parser is a software component that takes input data and builds a data structure - often some kind of parse tree, abstract syntax tree, giving a structural representation of the input while checking for correct syntax.
- The parser is often preceded by a separate lexical analyser, which creates tokens from the sequence of input characters
- Parser repeatedly matches left/right hand-side of a production against a substring in the current left/right-sentential form.

Parsing and Parse Trees

- We can represent a particular derivation by a CFG using a parse tree, where
  - Each internal node is labeled by a non-terminal symbol
  - Each leaf is labeled by terminal symbol
  - Root is labeled by the start symbol

<S>::=<S><S> | (<S>) | ()
Parse tree for
$\omega = (())()$



Ambiguous Grammars

A CFG is said to be ambiguous if there exists a string which has more than one left-most/right-most derivation.

$$< S > ::= < S > 1 \mid 0 < S > 1 \mid 01$$

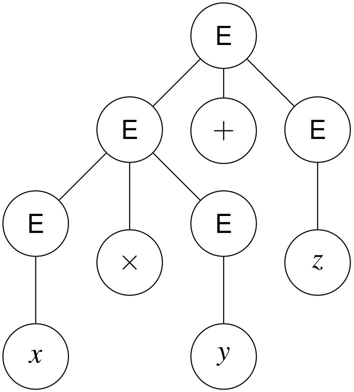
$$S \Rightarrow^* 00111$$

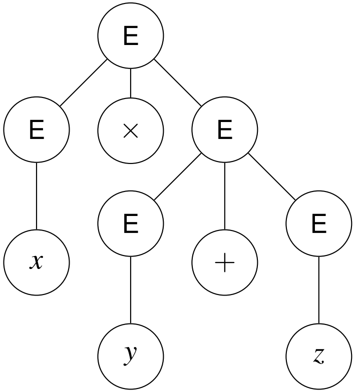
$$\begin{aligned} & < S > \\ & \Rightarrow 0 < S > 1 \\ & \Rightarrow 0 < S > 11 \\ & \Rightarrow 00111 \end{aligned} \qquad\qquad \begin{aligned} & < S > \\ & \Rightarrow < S > 1 \\ & \Rightarrow 0 < S > 11 \\ & \Rightarrow 00111 \end{aligned}$$

Ambiguous Grammars

$$<E>::=<E>\times<E>\ |\ <E>+<E>\ |\ (<E>)\ |\ x\ |\ y\ |\ z$$

$$E \Rightarrow^{*} x \times y + z$$





Ambiguous Grammars

It may be possible to remove ambiguity for some CFGs by rewriting the grammar and imposing rules and restrictions such as precedence.
Let's assume the precedence: $(), \times, +$

Ambiguous grammar

$$< S > ::= < S > + < S >$$
$$< S > ::= < S > \times < S >$$
$$< S > ::= (< S >)$$
$$< S > ::= x \mid y \mid z \mid 0 \mid 1$$

Unambiguous grammar

$$< A > ::= < A > + < B > \mid < B >$$
$$< B > ::= < B > \times < C > \mid < C >$$
$$< C > ::= (< A >) \mid < D >$$
$$< D > ::= x \mid y \mid z \mid 0 \mid 1$$

Ambiguous Grammars

Another example for the balanced parantheses grammar

Ambiguous grammar

$$< S > ::= < S >< S >$$
$$< S > ::= (< S >)$$
$$< S > ::= ()$$

Unambiguous grammar

$$< B > ::= (< R >< B > \mid \Lambda$$
$$< R > ::= \;) \mid (< R >< R >$$

However, for some languages, it may not be possible to remove ambiguity. A CFG is said to be inherently ambiguous if every CFG that describes it is ambiguous.

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

Example: Derivation of $a^n b^n c^n d^n$

$$L = \{0^i 1^j 2^k | i = j \vee j = k\}$$

Parsing Unambiguous Grammars

Unambiguous grammars can be parsed by tracking the unique leftmost derivation

$$<B> ::= (<R><B> \mid \Lambda$$
$$<R> ::= \ ) \mid (<R><R>$$

Parse input: $(())()$

Current token	Remaining String	Rule Replacement
$($	$())()$	$(<R><B>$
$($	$))()$	$((<R><R><B>$
$)$	$)()$	$(()<R><B>$
$)$	$()$	$(())<B>$
$($	$)$	$(())(<R><B>$
$)$	Λ	$(())()<B>$
Λ	Λ	$(())()$

Pay attention to first line, this substitution can be made since there is a single rule where the left-most token can be produced at.

LL and LR Parsing

- A grammar where you can always figure out the production to use in a leftmost derivation by scanning the given string left-to-right and looking only at the next one symbol is called $LL(1)$
- $LL(1)$: ”Leftmost derivation, left-to-right scan, one symbol of lookahead.”
- Most programming languages have $LL(1)$ grammars.
- $LL(1)$ grammars are never ambiguous.
- An LR parser reads input text from left to right without backing up and produces a rightmost derivation in reverse.
- It does a bottom-up parse - not a top-down LL parse.
- The name LR is often followed by a numeric qualifier, as in $LR(1)$ or sometimes $LR(k)$.
- To avoid backtracking or guessing, the LR parser is allowed to peek ahead at k lookahead input symbols before deciding how to parse earlier symbols.

Section 8

Deterministic Finite Automata and Regular Expressions

Definitions

Automaton

An automaton is an abstract model of a machine that perform computations on an input by moving through a series of states or configurations. At each state of the computation, a transition function determines the next configuration on the basis of a finite portion of the present configuration. As a result, once the computation reaches an accepting configuration, it accepts that input. The most general and powerful automata is the Turing machine.

Definitions

Deterministic Finite Automata

A DFA accepts/rejects finite strings of symbols and only produces a unique computation (or run) of the automaton for each input string. *Deterministic* refers to the uniqueness of the computation.

The major objective of automata theory is to develop methods by which computer scientists can describe and analyze the dynamic behavior of discrete systems, in which signals are sampled periodically.

Formal Definition of a DFA

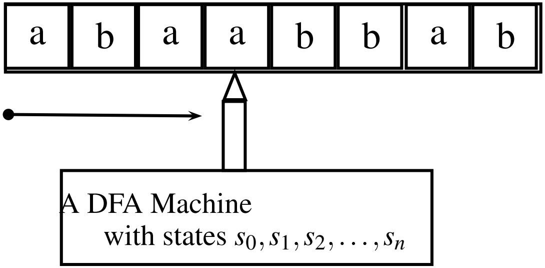
A deterministic finite state machine is a quintuple $M = (\Sigma, S, s_0, \delta, F)$, where:

- S: A finite, non-empty set of states where $s \in S$.
- Σ: Input alphabet (a finite, non-empty set of symbols)
- s_0: An initial state, an element of S.
- δ: The state-transition function $\delta : S \times \Sigma \rightarrow S$
- F: The set of final states where $F \subseteq S$.

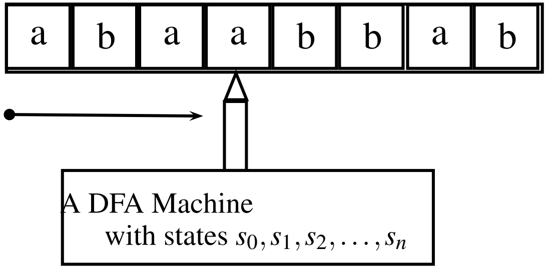
This machine is a Moore machine where each state produces the output in set $Z = \{0, 1\}$ corresponding to the machine's accepting/rejecting conditions.

DFA as a machine

Consider the physical machine below with an *input tape*. The tape is divided into cells, one next to the other. Each cell contains a symbol of a word from some finite alphabet. A machine that is modeled by a DFA, reads the contents of the cell successively and when the last symbol is read, the word is said to be accepted if the DFA is in an accepted state.



DFA as a machine



A run can be seen as a sequence of compositions of transition function with itself. Given an input symbol $\sigma \in \Sigma$ when this machine reads σ from the strip it can be written as $\delta(s, \sigma) = s' \in S$.

Configuration

A computation history is a (normally finite) sequence of configurations of a formal automaton. Each configuration fully describes the status of the machine at a particular point.
$s, \omega \in S \times \Sigma^*$

Configuration derivation is performed by a relation $\vdash_M$. If we denote the tuples in $\vdash_M$ as (s, ω) and (s', ω'), the relation can be defined as:

a $\omega = \sigma\omega' \wedge \sigma \in \Sigma$

b $\delta(s, \sigma) = s'$

A transition defined by this relation is called *derivation in one step* and denoted as $(s, \omega) \vdash_M (s', \omega')$. Following definitions can be defined based on this:

- Derivable configuration: $(s, \omega) \vdash_M^* (s', \omega')$ where $\vdash_M^*$ is the reflexive transitive closure of $\vdash_M$
- Recognized word: $(s_0, \omega) \vdash_M^* (s_i, \Lambda)$ where $s_i \in F$.
- Execution: $(s_0, \omega_0) \vdash (s_1, \omega_1) \vdash (s_2, \omega) \vdash \ldots \vdash (s_n, \Lambda)$ where Λ is the empty string.
- Recognized Language:
  $L(M) = \{\omega \in \Sigma^* | (s_0, \omega) \vdash_M^* (s_i, \Lambda) \wedge s_i \in F\}$

Language Recognizer

The reflexive transitive closure of $\vdash_M$ is denoted as ${\vdash_M}^*$.
$(q,\omega){\vdash_M}^*(q',\omega')$ denotes that (q,ω) yields (q',ω') after some number of steps.
$(s,\omega){\vdash_M}^*(q,\Lambda)$ denotes that $\omega \in \Sigma^*$ is recognized by an automaton if $q \in F$. In other words $L(M) = \{\omega \in \Sigma^* | (s,\omega){\vdash_M}^*(q_i,\Lambda) \wedge q_i \in F\}$

Example 1

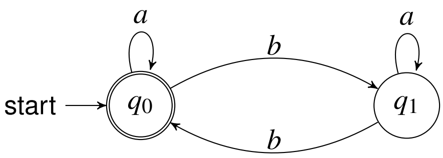
$S = \{q_0, q_1\}$

$\Sigma = \{a, b\}$

$s_0 = q_0$

$F = \{q_0\}$

q	σ	$\delta(q,\sigma)$
q_0	a	q_0
q_0	b	q_1
q_1	a	q_1
q_1	b	q_0



$$(q_0, aabba) \vdash_M (q_0, abba)$$
$$(q_0, abba) \vdash_M (q_0, bba)$$
$$(q_0, bba) \vdash_M (q_1, ba)$$
$$(q_1, ba) \vdash_M (q_0, a)$$
$$(q_0, a) \vdash_M (q_0, \Lambda)$$

Example 1

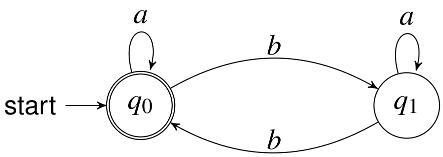
$S = \{q_0, q_1\}$

$\Sigma = \{a, b\}$

$s_0 = q_0$

$F = \{q_0\}$

q	σ	$\delta(q,\sigma)$
q_0	a	q_0
q_0	b	q_1
q_1	a	q_1
q_1	b	q_0



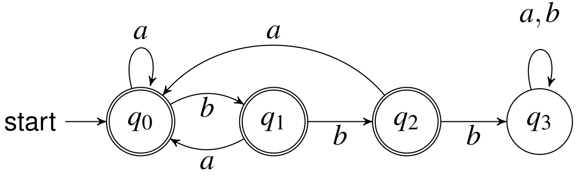
$L(M) = (a \vee ba^*b)^*$. We can write the grammar as:

$$V = S \cup \Sigma$$
$$I = \Sigma = \{a, b\}$$
$$s_0 = q_0 = n_0$$
$$< q_0 > ::= \Lambda \mid a < q_0 > \mid b < q_1 > \mid a$$
$$< q_1 > ::= b < q_0 > \mid a < q_1 > \mid b$$

Example 2

$L(M) = \{\omega | \omega \in \{a,b\}^* \wedge \omega$ should not include three successive b's$\}$
$S = \{q_0, q_1, q_2, q_3\}, \Sigma = \{a,b\}, s_0 = q_0, F = \{q_0, q_1, q_2\}$

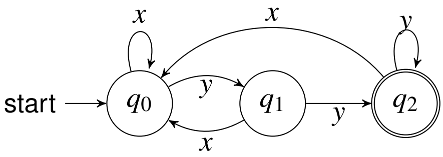
q	σ	$\delta(q,\sigma)$
q_0	a	q_0
q_0	b	q_1
q_1	a	q_0
q_1	b	q_2
q_2	a	q_0
q_2	b	q_3
q_3	a	q_3
q_3	b	q_3



We have a dead state q_3 where the automaton is not able to change state once it visits the dead state. $L(M) = [(\Lambda \vee b \vee bb)a]^*(\Lambda \vee b \vee bb)$

Example 3

$$S = \{q_0, q_1, q_2\}, \Sigma = \{x, y\}, s_0 = q_0 = n_0, F = \{q_2\}$$



$$< q_0 > ::= x < q_0 > \mid y < q_1 >$$
$$< q_1 > ::= y < q_2 > \mid y \mid x < q_0 >$$
$$< q_2 > ::= y \mid y < q_2 > \mid x < q_0 >$$

$$L(M) = ((x \vee yx)^* yy^+ x)^* (x \vee yx)^* yy^+$$
$$L(M) = ((\Lambda \vee y \vee yy^+)x)^* yy^+ = (y^* x)^* yy^+$$
$$L(M) = (x \vee yx \vee yy^+ x)^* yyy^*$$

State equivalence

During automaton design iterations, designers may come up with non-optimal state tables containing equivalent states. In order to overcome this situation mathematical definition of equivalency and state reduction principles are used.

Equivalence Relation

A given binary relation $\sim$ on a set S is said to be an equivalence relation if and only if it is reflexive, symmetric and transitive. Equivalently, $\forall s_i, s_j, s_k \in S$:

- $s_i \sim s_i$
- $s_i \sim s_j \Rightarrow s_j \sim s_i$
- $s_i \sim s_k \wedge s_k \sim s_j \Rightarrow s_i \sim s_j$

State equivalence

Using the mathematical definition of equivalency the set of machine states can be partitioned into equivalence classes. Equivelance classes should include

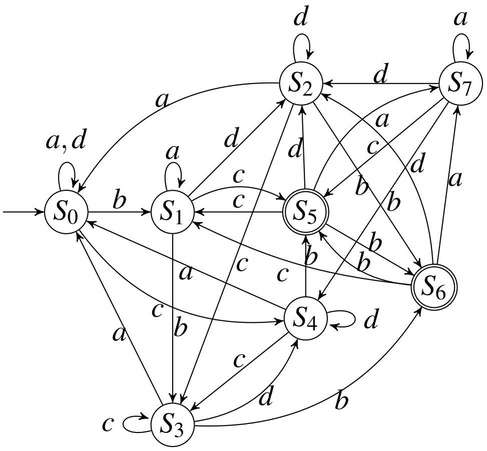
- Explicitly equivalent states
- Implicitly equivalent states, whose equivalency depends other states' equivalency.

State Equivalency Conditions

The necessary and sufficient conditions for two states in a state table to be equivalent are: For all the inputs of those states

- Outputs should be the same
- Successor states should
  - be explicitly or implicitly equivalent
  - have the same outputs

Dependency identification of states can be performed using dependency tables and undirected relation graphs. Let's consider the following automaton.



S/I	a	b	c	d
S_0	S_0	S_1	S_4	S_0
S_1	S_1	S_3	$\textcircled{S_5}$	S_2
S_2	S_0	$\textcircled{S_6}$	S_3	S_2
S_3	S_0	$\textcircled{S_6}$	S_3	S_4
S_4	S_0	$\textcircled{S_5}$	S_3	S_4
$\textcircled{S_5}$	S_7	$\textcircled{S_6}$	S_1	S_2
$\textcircled{S_6}$	S_7	$\textcircled{S_5}$	S_1	S_2
S_7	S_7	S_4	$\textcircled{S_5}$	S_2

We shall build a dependency table step by step based on the state dependencies.

S/I	a	b	c	d
S_0	S_0	S_1	S_4	S_0
S_1	S_1	S_3	$\textcircled{S_5}$	S_2
S_2	S_0	$\textcircled{S_6}$	S_3	S_2
S_3	S_0	$\textcircled{S_6}$	S_3	S_4
S_4	S_0	$\textcircled{S_5}$	S_3	S_4
$\textcircled{S_5}$	S_7	$\textcircled{S_6}$	S_1	S_2
$\textcircled{S_6}$	S_7	$\textcircled{S_5}$	S_1	S_2
S_7	S_7	S_4	$\textcircled{S_5}$	S_2

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
S_0	=							
S_1		=						
S_2			=					
S_3				=				
S_4					=			
S_5						=		
S_6							=	
S_7								=

S/I	a	b	c	d
S_0	S_0	S_1	S_4	S_0
S_1	S_1	S_3	$\textcircled{S_5}$	S_2
S_2	S_0	$\textcircled{S_6}$	S_3	S_2
S_3	S_0	$\textcircled{S_6}$	S_3	S_4
S_4	S_0	$\textcircled{S_5}$	S_3	S_4
$\textcircled{S_5}$	S_7	$\textcircled{S_6}$	S_1	S_2
$\textcircled{S_6}$	S_7	$\textcircled{S_5}$	S_1	S_2
S_7	S_7	S_4	$\textcircled{S_5}$	S_2

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
S_0	=	**X**						
S_1		=						
S_2			=					
S_3				=				
S_4					=			
S_5						=		
S_6							=	
S_7								=

S/I	a	b	c	d
S_0	S_0	S_1	S_4	S_0
S_1	S_1	S_3	$\textcircled{S_5}$	S_2
S_2	S_0	$\textcircled{S_6}$	S_3	S_2
S_3	S_0	$\textcircled{S_6}$	S_3	S_4
S_4	S_0	$\textcircled{S_5}$	S_3	S_4
$\textcircled{S_5}$	S_7	$\textcircled{S_6}$	S_1	S_2
$\textcircled{S_6}$	S_7	$\textcircled{S_5}$	S_1	S_2
S_7	S_7	S_4	$\textcircled{S_5}$	S_2

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
S_0	=	X	X	X	X	X	X	X
S_1		=						
S_2			=					
S_3				=				
S_4					=			
S_5						=		
S_6							=	
S_7								=

S/I	a	b	c	d
S_0	S_0	S_1	S_4	S_0
S_1	S_1	S_3	$\textcircled{S_5}$	S_2
S_2	S_0	$\textcircled{S_6}$	S_3	S_2
S_3	S_0	$\textcircled{S_6}$	S_3	S_4
S_4	S_0	$\textcircled{S_5}$	S_3	S_4
$\textcircled{S_5}$	S_7	$\textcircled{S_6}$	S_1	S_2
$\textcircled{S_6}$	S_7	$\textcircled{S_5}$	S_1	S_2
S_7	S_7	S_4	$\textcircled{S_5}$	S_2

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
S_0	=	X	X	X	X	X	X	X
S_1		=	**X**					
S_2			=					
S_3				=				
S_4					=			
S_5						=		
S_6							=	
S_7								=

S/I	a	b	c	d
S_0	S_0	S_1	S_4	S_0
S_1	S_1	S_3	$\textcircled{S_5}$	S_2
S_2	S_0	$\textcircled{S_6}$	S_3	S_2
S_3	S_0	$\textcircled{S_6}$	S_3	S_4
S_4	S_0	$\textcircled{S_5}$	S_3	S_4
$\textcircled{S_5}$	S_7	$\textcircled{S_6}$	S_1	S_2
$\textcircled{S_6}$	S_7	$\textcircled{S_5}$	S_1	S_2
S_7	S_7	S_4	$\textcircled{S_5}$	S_2

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
S_0	=	X	X	X	X	X	X	X
S_1		=	X	X	X	X	X	(S_3, S_4)
S_2			=					
S_3				=				
S_4					=			
S_5						=		
S_6							=	
S_7								=

S/I	a	b	c	d
S_0	S_0	S_1	S_4	S_0
S_1	S_1	S_3	$\textcircled{S_5}$	S_2
S_2	S_0	$\textcircled{S_6}$	S_3	S_2
S_3	S_0	$\textcircled{S_6}$	S_3	S_4
S_4	S_0	$\textcircled{S_5}$	S_3	S_4
$\textcircled{S_5}$	S_7	$\textcircled{S_6}$	S_1	S_2
$\textcircled{S_6}$	S_7	$\textcircled{S_5}$	S_1	S_2
S_7	S_7	S_4	$\textcircled{S_5}$	S_2

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
S_0	=	X	X	X	X	X	X	X
S_1		=	X	X	X	X	X	(S_3, S_4)
S_2			=	(S_2, S_4)				
S_3				=				
S_4					=			
S_5						=		
S_6							=	
S_7								=

S/I	a	b	c	d
S_0	S_0	S_1	S_4	S_0
S_1	S_1	S_3	$\textcircled{S_5}$	S_2
S_2	S_0	$\textcircled{S_6}$	S_3	S_2
S_3	S_0	$\textcircled{S_6}$	S_3	S_4
S_4	S_0	$\textcircled{S_5}$	S_3	S_4
$\textcircled{S_5}$	S_7	$\textcircled{S_6}$	S_1	S_2
$\textcircled{S_6}$	S_7	$\textcircled{S_5}$	S_1	S_2
S_7	S_7	S_4	$\textcircled{S_5}$	S_2

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
S_0	=	X	X	X	X	X	X	X
S_1		=	X	X	X	X	X	(S_3, S_4)
S_2			=	(S_2, S_4)	(S_5, S_6)			
S_3				=				
S_4					=			
S_5						=		
S_6							=	
S_7								=

S/I	a	b	c	d
S_0	S_0	S_1	S_4	S_0
S_1	S_1	S_3	\textcircled{S_5}	S_2
S_2	S_0	\textcircled{S_6}	S_3	S_2
S_3	S_0	\textcircled{S_6}	S_3	S_4
S_4	S_0	\textcircled{S_5}	S_3	S_4
\textcircled{S_5}	S_7	\textcircled{S_6}	S_1	S_2
\textcircled{S_6}	S_7	\textcircled{S_5}	S_1	S_2
S_7	S_7	S_4	\textcircled{S_5}	S_2

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
S_0	=	X	X	X	X	X	X	X
S_1		=	X	X	X	X	X	(S_3, S_4)
S_2			=	(S_2, S_4)	(S_5, S_6)	X	X	
S_3				=				
S_4					=			
S_5						=		
S_6							=	
S_7								=

S/I	a	b	c	d
S_0	S_0	S_1	S_4	S_0
S_1	S_1	S_3	$\textcircled{S_5}$	S_2
S_2	S_0	$\textcircled{S_6}$	S_3	S_2
S_3	S_0	$\textcircled{S_6}$	S_3	S_4
S_4	S_0	$\textcircled{S_5}$	S_3	S_4
$\textcircled{S_5}$	S_7	$\textcircled{S_6}$	S_1	S_2
$\textcircled{S_6}$	S_7	$\textcircled{S_5}$	S_1	S_2
S_7	S_7	S_4	$\textcircled{S_5}$	S_2

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
S_0	=	X	X	X	X	X	X	X
S_1		=	X	X	X	X	X	(S_3,S_4)
S_2			=	(S_2,S_4)	(S_5,S_6)	X	X	(S_0,S_7)**X**
S_3				=				
S_4					=			
S_5						=		
S_6							=	
S_7								=

S/I	a	b	c	d
S_0	S_0	S_1	S_4	S_0
S_1	S_1	S_3	$\textcircled{S_5}$	S_2
S_2	S_0	$\textcircled{S_6}$	S_3	S_2
S_3	S_0	$\textcircled{S_6}$	S_3	S_4
S_4	S_0	$\textcircled{S_5}$	S_3	S_4
$\textcircled{S_5}$	S_7	$\textcircled{S_6}$	S_1	S_2
$\textcircled{S_6}$	S_7	$\textcircled{S_5}$	S_1	S_2
S_7	S_7	S_4	$\textcircled{S_5}$	S_2

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
S_0	=	X	X	X	X	X	X	X
S_1		=	X	X	X	X	X	(S_3, S_4)
S_2			=	(S_2, S_4)	(S_5, S_6)	X	X	X
S_3				=	(S_5, S_6)			
S_4					=			
S_5						=		
S_6							=	
S_7								=

S/I	a	b	c	d
S_0	S_0	S_1	S_4	S_0
S_1	S_1	S_3	$\textcircled{S_5}$	S_2
S_2	S_0	$\textcircled{S_6}$	S_3	S_2
S_3	S_0	$\textcircled{S_6}$	S_3	S_4
S_4	S_0	$\textcircled{S_5}$	S_3	S_4
$\textcircled{S_5}$	S_7	$\textcircled{S_6}$	S_1	S_2
$\textcircled{S_6}$	S_7	$\textcircled{S_5}$	S_1	S_2
S_7	S_7	S_4	$\textcircled{S_5}$	S_2

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
S_0	=	X	X	X	X	X	X	X
S_1		=	X	X	X	X	X	(S_3, S_4)
S_2			=	(S_2, S_4)	(S_5, S_6)	X	X	X
S_3				=	(S_5, S_6)	X	X	X
S_4					=	X	X	X
S_5						=		
S_6							=	
S_7								=

S/I	a	b	c	d
S_0	S_0	S_1	S_4	S_0
S_1	S_1	S_3	$\textcircled{S_5}$	S_2
S_2	S_0	$\textcircled{S_6}$	S_3	S_2
S_3	S_0	$\textcircled{S_6}$	S_3	S_4
S_4	S_0	$\textcircled{S_5}$	S_3	S_4
$\textcircled{S_5}$	S_7	$\textcircled{S_6}$	S_1	S_2
$\textcircled{S_6}$	S_7	$\textcircled{S_5}$	S_1	S_2
S_7	S_7	S_4	$\textcircled{S_5}$	S_2

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
S_0	=	X	X	X	X	X	X	X
S_1		=	X	X	X	X	X	(S_3, S_4)
S_2			=	(S_2, S_4)	(S_5, S_6)	X	X	X
S_3				=	(S_5, S_6)	X	X	X
S_4					=	X	X	X
S_5						=	◯	
S_6							=	
S_7								=

S/I	a	b	c	d
S_0	S_0	S_1	S_4	S_0
S_1	S_1	S_3	$\textcircled{S_5}$	S_2
S_2	S_0	$\textcircled{S_6}$	S_3	S_2
S_3	S_0	$\textcircled{S_6}$	S_3	S_4
S_4	S_0	$\textcircled{S_5}$	S_3	S_4
$\textcircled{S_5}$	S_7	$\textcircled{S_6}$	S_1	S_2
$\textcircled{S_6}$	S_7	$\textcircled{S_5}$	S_1	S_2
S_7	S_7	S_4	$\textcircled{S_5}$	S_2

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
S_0	=	X	X	X	X	X	X	X
S_1		=	X	X	X	X	X	(S_3, S_4)
S_2			=	(S_2, S_4)	(S_5, S_6)	X	X	X
S_3				=	(S_5, S_6)	X	X	X
S_4					=	X	X	X
S_5						=	◯	X
S_6							=	
S_7								=

S/I	a	b	c	d
S_0	S_0	S_1	S_4	S_0
S_1	S_1	S_3	$\enclose{circle}{S_5}$	S_2
S_2	S_0	$\enclose{circle}{S_6}$	S_3	S_2
S_3	S_0	$\enclose{circle}{S_6}$	S_3	S_4
S_4	S_0	$\enclose{circle}{S_5}$	S_3	S_4
$\enclose{circle}{S_5}$	S_7	$\enclose{circle}{S_6}$	S_1	S_2
$\enclose{circle}{S_6}$	S_7	$\enclose{circle}{S_5}$	S_1	S_2
S_7	S_7	S_4	$\enclose{circle}{S_5}$	S_2

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
S_0	=	X	X	X	X	X	X	X
S_1		=	X	X	X	X	X	(S_3,S_4)
S_2			=	(S_2,S_4)	(S_5,S_6)	X	X	X
S_3				=	(S_5,S_6)	X	X	X
S_4					=	X	X	X
S_5						=	$\bigcirc$	X
S_6							=	X
S_7								=

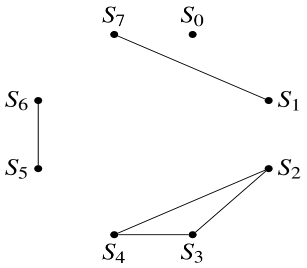
In the dependency table below we can see that some equivalencies depend on others. We can sketch a directed graph based on those dependencies.

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
S_0	=	X	X	X	X	X	X	X
S_1		=	X	X	X	X	X	(S_3,S_4)
S_2			=	(S_2,S_4)	(S_5,S_6)	X	X	X
S_3				=	(S_5,S_6)	X	X	X
S_4					=	X	X	X
S_5						=	○	X
S_6							=	X
S_7								=

$$\begin{tikzcd}
(S_1,S_7) \arrow[d] & & (S_2,S_3) \arrow[d] \\
(S_3,S_4) \arrow[dr] & & (S_2,S_4) \arrow[dl] \\
& (S_5,S_6) &
\end{tikzcd}$$

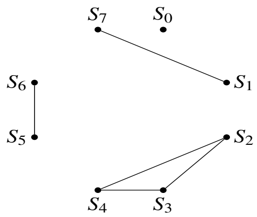
We can sketch an undirected dependency graph using directed dependencies and find the connected components inside the graph. We can combine these cliques and discover equivalent states.

$$\begin{tikzcd}
(S_1,S_7) \arrow[d] & & (S_2,S_3) \arrow[d] \\
(S_3,S_4) \arrow[dr] & & (S_2,S_4) \arrow[dl] \\
& (S_5,S_6) &
\end{tikzcd}$$



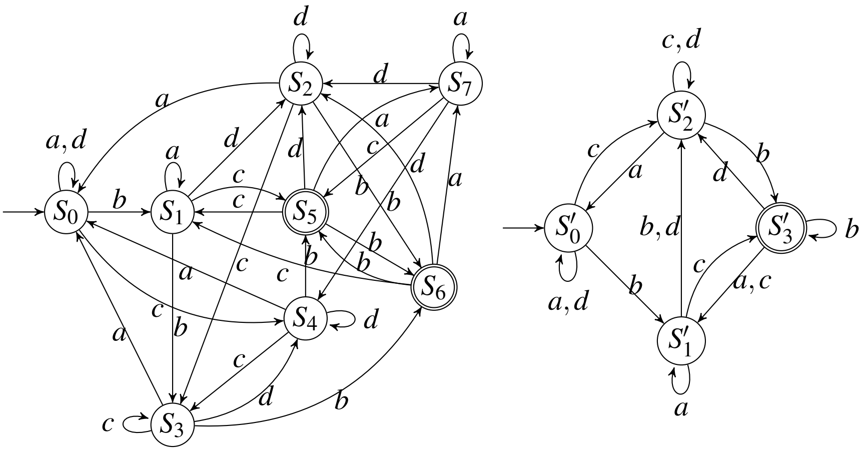
S/I	a	b	c	d
S_0	S_0	S_1	S_4	S_0
S_1	S_1	S_3	$\enclose{circle}{S_5}$	S_2
S_2	S_0	$\enclose{circle}{S_6}$	S_3	S_2
S_3	S_0	$\enclose{circle}{S_6}$	S_3	S_4
S_4	S_0	$\enclose{circle}{S_5}$	S_3	S_4
$\enclose{circle}{S_5}$	S_7	$\enclose{circle}{S_6}$	S_1	S_2
$\enclose{circle}{S_6}$	S_7	$\enclose{circle}{S_5}$	S_1	S_2
S_7	S_7	S_4	$\enclose{circle}{S_5}$	S_2

If we rebuild our state table using the equivalence classes we have just discovered



		a	b	c	d
S'_0	(S_0)	S'_0	S'_1	S'_2	S'_0
S'_1	(S_1, S_7)	S'_1	S'_2	$\enclose{circle}{S'_3}$	S'_2
S'_2	(S_2, S_3, S_4)	S'_0	$\enclose{circle}{S'_3}$	S'_2	S'_2
$\enclose{circle}{S'_3}$	(S_5, S_6)	S'_1	$\enclose{circle}{S'_3}$	S'_1	S'_2

Dependency identification of states can be performed using dependency tables and undirected relation graphs. Let's consider the following automaton.



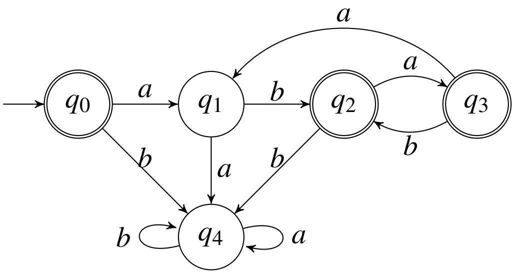
Section 9

Non-Deterministic Finite Automata

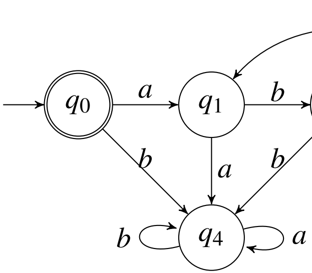
Non-deterministic Finite Automata(NFA)

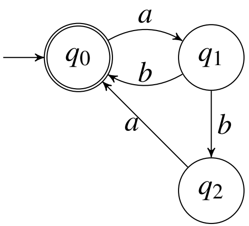
In an NFA, given an input symbol it is possible to jump into several possible next states from each state.

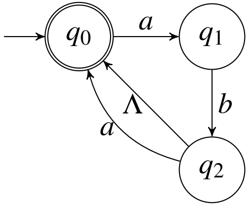
A DFA recognizing $L = (ab \vee aba)^*$ can be diagramatically shown as:

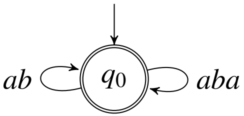


We may construct three different NFAs recognizing the same language.









Formal Definition of an NFA

A non-deterministic finite state automata is a quintuple $M = (\Sigma, S, s_0, \Delta, F)$, where:

- S: A finite, non-empty set of states where $s \in S$.
- Σ: Input alphabet (a finite, non-empty set of symbols)
- s_0: An initial state, an element of S.
- Δ: The state-transition relation $\Delta \subseteq S \times \Sigma^* \times S$
- F: The set of final states where $F \subseteq S$.

A configuration is defined as a tuple in set $S \times \Sigma^*$. Considering the definition of derivation in one step:
$(q, \omega) \vdash_M (q', \omega') \Rightarrow \exists u \in \Sigma^* (\omega = u\omega' \wedge (q, u, q') \in \Delta)$
For deterministic automata $\Delta \subseteq S \times \Sigma^* \times S$ relation becomes a function $S \times \Sigma \to S$. For (q, u, q') triples $|u| = 1 \wedge (\forall q \in S \wedge \forall u \in \Sigma) \exists! q' \in S$
The language that an NFA recognizes is
$L(M) = \{\omega | (s, \omega) \vdash_m{}^* (q, \Lambda) \wedge q \in F\}$

An example NFA

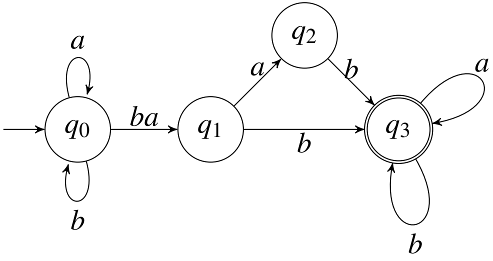
Build an NFA that recognizes languages including bab or baab as substrings.

$S = \{q_0, q_1, q_2, q_3\}$
$\Sigma = \{a, b\}$
$s_0 = q_0$
$F = \{q_3\}$
$\Delta = \{(q_0, a, q_0), (q_0, b, q_0), (q_0, ba, q_1), (q_1, b, q_3), (q_1, a, q_2),$
$(q_2, b, q_3), (q_3, a, q_3), (q_3, b, q_3)\}$

$M = (S, \Sigma, \Delta, s_0, F)$
$< q_0 > ::= a < q_0 > \ | \ b < q_0 > \ | \ ba < q_1 >$
$< q_1 > ::= b < q_3 > \ | \ b \ | \ a < q_2 >$
$< q_2 > ::= b < q_3 > \ | \ b$
$< q_3 > ::= a \ | \ b \ | \ a < q_3 > \ | \ b < q_3 >$

An example NFA

$$< q_0 > ::= a < q_0 > \ | \ b < q_0 > \ | \ ba < q_1 >$$
$$< q_1 > ::= b < q_3 > \ | \ b \ | \ a < q_2 >$$
$$< q_2 > ::= b < q_3 > \ | \ b$$
$$< q_3 > ::= a \ | \ b \ | \ a < q_3 > \ | \ b < q_3 >$$



A possible derivation may follow the path:
$(q_0, aaabbbaabab) \vdash$
$(q_0, aabbbaabab) \vdash$
$(q_0, abbbaabab) \vdash$
$(q_0, bbbaabab) \vdash$
$(q_0, bbaabab) \vdash$
$(q_0, baabab) \vdash (q_1, abab) \vdash$
$(q_2, bab) \vdash (q_3, ab) \vdash$
$(q_3, b) \vdash (q_3, \Lambda)$

Lemma

$M = (S, \Sigma, \Delta, s_0, F) \wedge q, r \in S \wedge x, y \in \Sigma^*$
$\exists p \in S \wedge (q,x) \vdash_M{}^* (p, \Lambda) \wedge (p, y) \vdash_M{}^* (r, \Lambda) \Rightarrow (q, xy) \vdash_M{}^* (r, \Lambda)$

Definition

Regular Grammar: All the production rules are of type-3.
Regular Language: Languages that can be recognized by regular grammars.
Regular Expression: $\varnothing, \{\Lambda\}, \{a | a \in \Sigma\}, A \vee B, A.B, A^*$
Regular set : The sets which can be represented by regular expressions are called regular sets.

Regular grammars can be represented by NFAs.

Definition

Regular Grammar: All the production rules are of type-3.
Regular Language: Languages that can be recognized by regular grammars.
Regular Expression: $\varnothing, \{\Lambda\}, \{a | a \in \Sigma\}, A \vee B, A.B, A^*$
Regular set : The sets which can be represented by regular expressions are called regular sets.

Regular grammars can be represented by NFAs.

a) Non-terminal symbols are assigned to states
b) Inititial state corresponds to initial symbol
c) Accepting states sorresponds to the rules that end with terminal symbols
d) If Λ should be recognized, initial state is an accepting state.

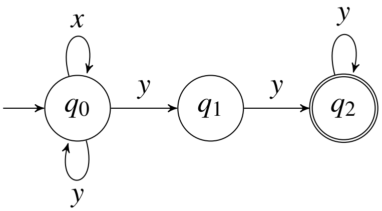
Languages recognized by finite automata(Regular Languages) are closed under union, concatanation and Kleene star operations.

Kleene Theorem

Every regular language can be recognized by a finite automaton and every finite automaton defines a regular language.
$M = (S, \Sigma, \Delta, s_0, F) \Leftrightarrow G = (N, \Sigma, n_0, \mapsto), L = L(G)$ a grammar of type-3.
$S = N \wedge F \subseteq N$
$s_0 = n_0$

$$\begin{aligned} \Delta = \\ & \{(A, \omega, B) : (A \mapsto \omega B) \in \mapsto \wedge (A, B \in N) \wedge \omega \in \Sigma^*\} \\ & \cup \\ & \{(A, \omega, f_i) : (A \mapsto \omega) \in \mapsto \wedge A \in N \wedge f_i \in F \wedge \omega \in \Sigma^*\} \end{aligned}$$

Example



$$< q_0 > ::= x < q_0 > \mid y < q_0 > \mid y < q_1 >$$
$$< q_1 > ::= y < q_2 >$$
$$< q_2 > ::= y < q_2 > \mid \Lambda$$

Section 10

DFA-NFA Equivalency

Kleene Theorem

For every NFA an equivalent DFA can be constructed.

For the NFA $M = (S, \Sigma, \Delta, s_0, F)$ our aim is to...

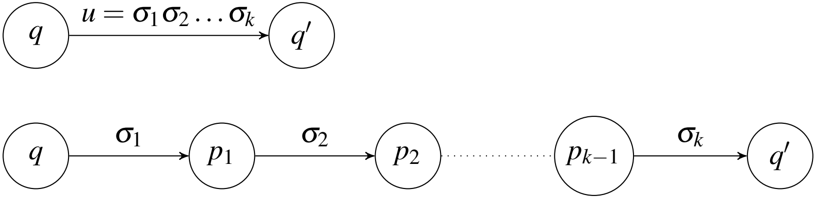
(a) In $(q, u, q') \in \Delta$ there shouldn't be any $u = \Lambda$ and $|u| > 1$

(b) A transition should be present for all symbols in all states

(c) There shouldn't be more than one transitions for each configuration.

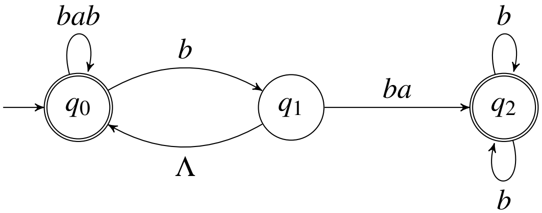
NFA/DFA equivalency-Phase 1

Interim steps are populated to eliminate the $|u| > 1$ in (q, u, q') of Δ.

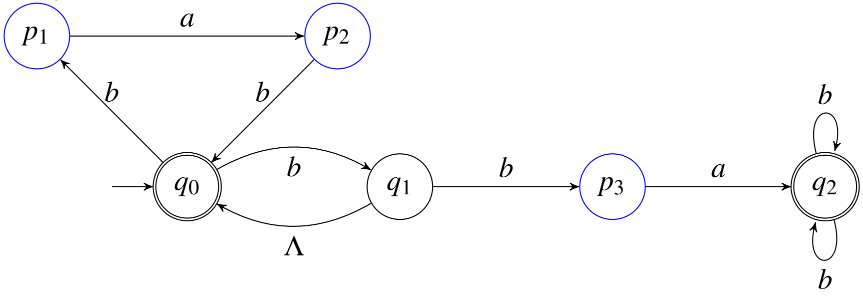

$$\begin{tikzcd}
q \arrow[r, "u = \sigma_1\sigma_2\ldots\sigma_k"] & q'
\end{tikzcd}$$
$$\begin{tikzcd}
q \arrow[r, "\sigma_1"] & p_1 \arrow[r, "\sigma_2"] & p_2 \arrow[r, dotted, no head] & p_{k-1} \arrow[r, "\sigma_k"] & q'
\end{tikzcd}$$

This expansion transforms Δ into Δ' by replacing triples of (q, u, q') with triples like $(q, \sigma_1, p_1), (p_1, \sigma_2, p_2), \ldots, (p_{k-1}, \sigma_k, q')$. A new machine is formed $M' = (S', \Sigma, \Delta', s'_0, F')$ where $F' \equiv F$ and $s'_0 \equiv s_0$

NFA/DFA equivalency-Phase 1



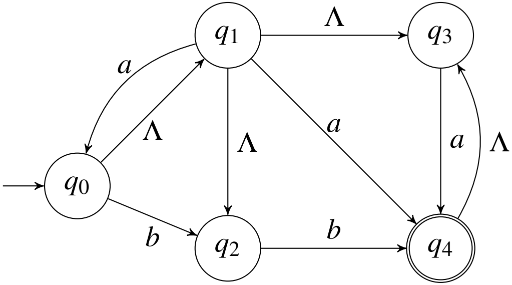
is equivalent to



NFA/DFA equivalency-Phase 2

Reachability set of a state

$R(q) = \{p \in S' | (q, \Lambda) \vdash_{M'}{}^{*} (p, \Lambda)\}$ or
$R(q) = \{p \in S' | (q, \omega) \vdash_{M'}{}^{*} (p, \omega)\}$

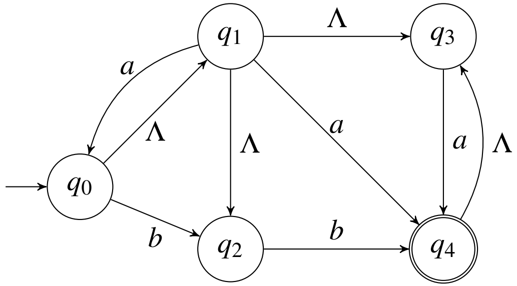


$R(q_0) = \{q_0, q_1, q_2, q_3\}$
$R(q_1) = \{q_1, q_2, q_3\}$
$R(q_2) = \{q_2\}$
$R(q_3) = \{q_3\}$
$R(q_4) = \{q_3, q_4\}$

NFA/DFA equivalency-Phase 2

Constructing an equivalent deterministic machine:
$M'' = (S'', \Sigma, \delta'', F'')$
$S'' \subseteq \mathscr{P}(S') = 2^{S'}$
$s_0'' = R(s_0')$ The states that can be reached from the initial state by Λ transitions
$F'' = \{Q \subseteq S' | Q \cap F' \neq \varnothing\}$

NFA/DFA equivalency-Phase 2



Constructing an equivalent deterministic machine, definition of δ'':

$\forall Q \subseteq S' \wedge \forall \sigma \in \Sigma$

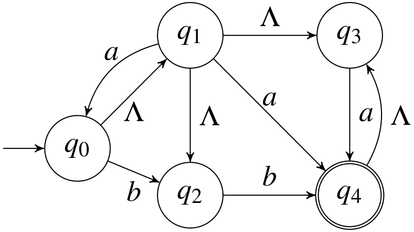
$\delta''(Q,\sigma) = \bigcup_P \{R(p) | \forall q \in Q \wedge \forall p \in S' \wedge \forall (q,\sigma,p) \in \Delta'\}$

Let's write all the possible triplets except empty string:

Transitions with a : $(q_1,a,q_0),(q_1,a,q_4),(q_3,a,q_4),$

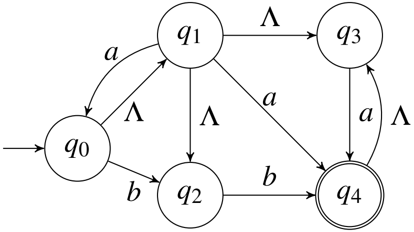
Transitions with b : $(q_0,b,q_2),(q_2,b,q_4)$

NFA/DFA equivalency-Phase 2



Let's write all the possible triplets except empty string:
Transitions with a : $(q_1, a, q_0), (q_1, a, q_4), (q_3, a, q_4),$
Transitions with b : $(q_0, b, q_2), (q_2, b, q_4)$
Let's build δ'' using those transitions:
$s''_0 = R(q_0) = \{q_0, q_1, q_2, q_3\}(d_0)$
$\delta''(d_0, a) = R(q_0) \cup R(q_4) = \{q_0, q_1, q_2, q_3, q_4\}(d_1)$
$\delta''(d_0, b) = R(q_2) \cup R(q_4) = \{q_2, q_3, q_4\}(d_2)$
$\delta''(d_1, a) = \{q_0, q_1, q_2, q_3, q_4\}(d_1)$
$\delta''(d_1, b) = \{q_2, q_3, q_4\}(d_2)$

NFA/DFA equivalency-Phase 2



Let's write all the possible triplets except empty string:

Transitions with a : $(q_1, a, q_0), (q_1, a, q_4), (q_3, a, q_4),$

Transitions with b : $(q_0, b, q_2), (q_2, b, q_4)$

Let's build δ'' using those transitions:

$\delta''(d_2, a) = R(q_4) = \{q_3, q_4\}(d_3)$

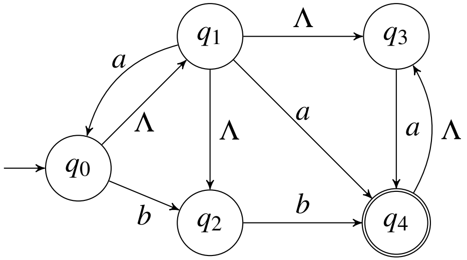
$\delta''(d_2, b) = R(q_4) = \{q_3, q_4\}(d_3)$

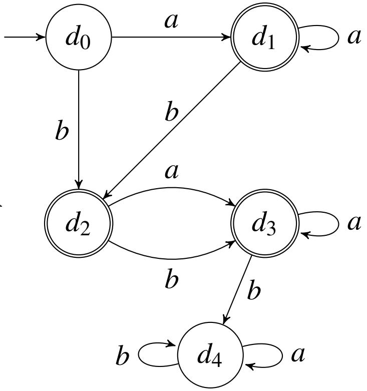
$\delta''(d_3, a) = R(q_4) = \{q_3, q_4\}(d_3)$

$\delta''(d_3, b) = \varnothing(d_4)$

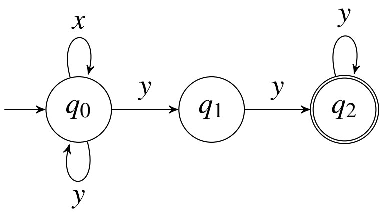
$\delta''(d_4, a) = \delta''(d_4, b) = \varnothing(d_4)$

NFA/DFA equivalency-Phase 2





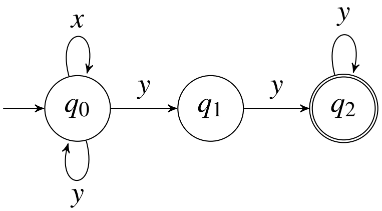
Example for NFA/DFA equivalency



An NFA recognizing the language $L(M) = (x \vee y)^* yy^+$. Let's build an equivalent DFA.

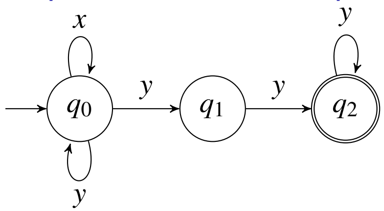
$R(q_0) = \{q_0\}$
$R(q_1) = \{q_1\}$
$R(q_2) = \{q_2\}$

Example for NFA/DFA equivalency

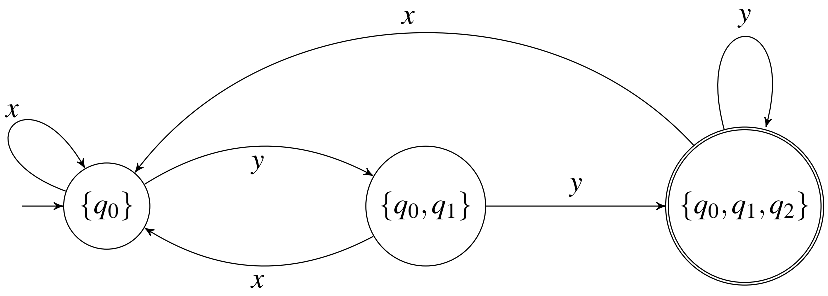


$$\Delta' = \{(q_0,x,q_0),(q_0,y,q_0),(q_0,y,q_1),(q_1,y,q_2),(q_2,y,q_2)\}$$
$$s_0'' = R(q_0) = \{q_0\}$$
$$\delta(s_0'',x) = R(q_0) = \{q_0\}$$
$$\delta(s_0'',y) = R(q_0) \cup R(q_1) = \{q_0,q_1\}$$
$$\delta(\{q_0,q_1\},x) = R(q_0) = \{q_0\}$$
$$\delta(\{q_0,q_1\},y) = R(q_0) \cup R(q_1) \cup R(q_2) = \{q_0,q_1,q_2\}$$
$$\delta(\{q_0,q_1,q_2\},x) = R(q_0) = \{q_0\}$$
$$\delta(\{q_0,q_1,q_2\},y) = R(q_0) \cup R(q_1) \cup R(q_2) = \{q_0,q_1,q_2\}$$

Example for NFA/DFA equivalency



$L(M) = (x \vee y)^* yy^+$



$L(M) = ((\Lambda \vee y \vee yy^+)x)^* yy^+$

Section 11

Recognizing Regular Expressions

Kleene Theorem (cont.)

Any language is regular which is constructed by applying closed language operations on regular languages. e.g. if we know L_1 and L_2 are regular languages, any language built by applying a closed operation on them produces a new regular language.

Kleene Theorem (cont.)

Regular languages recognized by a finite automaton is closed under the following operations

(a) Union, Intersection, Complement, Difference

(b) Concatanation

(c) Kleene star

(d) Reversal

(e) Homomorphism, Inverse Homomorphism

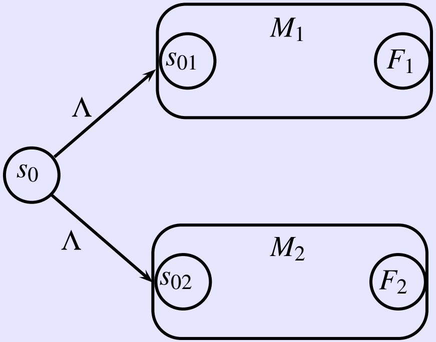
Union (Non-deterministic)

$$M_1 = (S_1, \Sigma, \Delta_1, s_{01}, F_1) \leftarrow L(M_1)$$
$$M_2 = (S_2, \Sigma, \Delta_2, s_{02}, F_2) \leftarrow L(M_2)$$

$$M_= (S, \Sigma, \delta, s_0, F) \leftarrow L(M_1) \cup L(M_2)$$

$$S = S_1 \cup S_2 \cup \{s_0\}\ F = F_1 \cup F_2\ \delta = \Delta_1 \cup \Delta_2 \cup \{(s_0, \Lambda, s_{01}), (s_0, \Lambda, s_{02})\}$$

Union



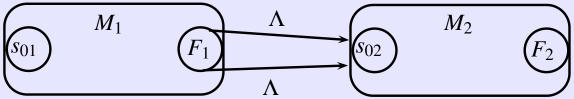
Concatanation (Non-deterministic)

$L(M_1).L(M_2) = L(M)$

$S = S_1 \cup S_2$
$s_0 = s_{01}$
$F = F_2$
$\delta = \Delta_1 \cup \Delta_2 \cup (F_1 \times \{\Lambda\} \times \{s_{02}\})$

Concatanation

$s_0 = s_{0,1}$



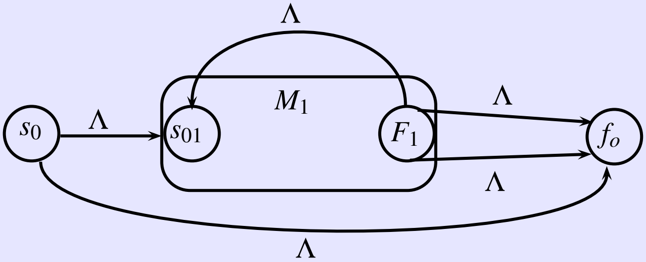
$F = F_2$

Kleene Star (Non-deterministic)

$L(M_1)^* = L(M)$

$S = S_1 \cup \{s_0\}$
$F = \{f_o\}$
$\delta = \Delta_1 \cup (F_1 \times \{\Lambda\} \times \{s_{01}\}) \cup (s_0, \Lambda, s_{01}) \cup (F_1 \times \{\Lambda\} \times F) \cup (s_0, \Lambda, f_o)$

Kleene Star



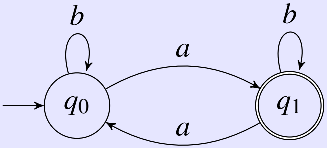
Complement (deterministic)

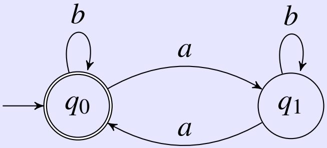
$\overline{L(M_1)} = L(M)$

$F = \{S_1 - F_1\}$

Languages that contain odd number of a's in $\Sigma = \{a, b\}$

Complement



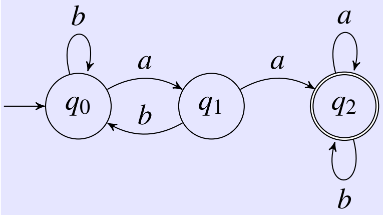


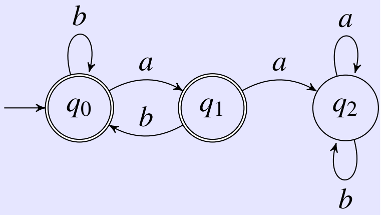
$q_1 = b^*a(b \vee ab^*a)^*$

$q_0 = (b \vee ab^*a)^*$

Languages that contain at least one aa couple in $\Sigma = \{a, b\}$

Complement





$q_2 = (b \vee ab)^* aa(a \vee b)^*$

$q_0 \vee q_1 = (b \vee ab)^* (a \vee \Lambda)$

Intersection (Deterministic)

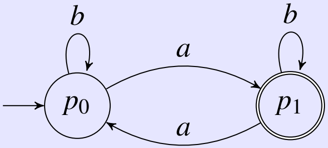
$L(M_1) \cap L(M_2) = L(M)$

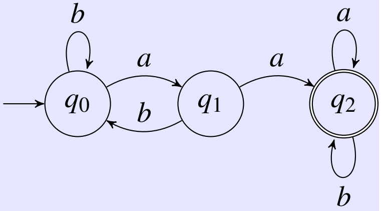
$S \subseteq S_1 \times S_2$
$s_0 = (s_{01}, s_{02})$

$F \subseteq F_1 \times F_2$ where
$[(p_1, p_2) \in F] \iff [[p_1 \in F_1] \wedge [p_2 \in F_2]]$

$\Delta \subseteq \Delta_1 \times \Delta_2$ where
$[((p_1, p_2), \sigma, (q_1, q_2)) \in \Delta] \iff [[(p_1, \sigma, q_1) \in \Delta_1] \wedge [(p_2, \sigma, q_2) \in \Delta_2]]$

Intersection





$$S = \{p_0, p_1\}$$
$$s_0 = \{p_0\}$$
$$F = \{p_1\}$$
$$\Delta = \{(p_0, b, p_0), (p_0, a, p_1), (p_1, b, p_1), (p_1, a, p_0)\}$$

$$S = \{q_0, q_1, q_2\}$$
$$s_0 = \{q_0\}$$
$$F = \{q_2\}$$
$$\Delta = \{(q_0, b, q_0), (q_0, a, q_1), (q_1, b, q_0), (q_1, a, q_2), (q_2, a, q_2), (q_2, b, q_2)\}$$

Intersection

M_1 :

$$\begin{aligned} S &= \{p_0, p_1\} \\ s_0 &= \{p_0\} \\ F &= \{p_1\} \\ \Delta &= \{(p_0, b, p_0), (p_0, a, p_1), \\ &\quad (p_1, b, p_1), (p_1, a, p_0)\} \end{aligned}$$

M_2 :

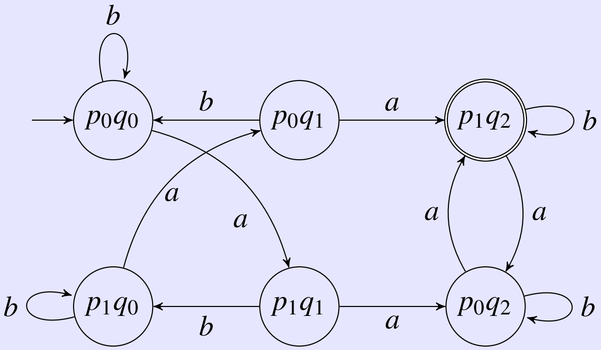
$$\begin{aligned} S &= \{q_0, q_1, q_2\} \\ s_0 &= \{q_0\} \\ F &= \{q_2\} \\ \Delta &= \{(q_0, b, q_0), (q_0, a, q_1), \\ &\quad (q_1, b, q_0), (q_1, a, q_2), \\ &\quad (q_2, a, q_2), (q_2, b, q_2)\} \end{aligned}$$

$M_1 \cap M_2$:

$$\begin{aligned} S &= \{(p_0, q_0), (p_0, q_1), (p_0, q_2), (p_1, q_0), (p_1, q_1), (p_1, q_2)\} \\ s_0 &= \{(p_0, q_0)\} \\ F &= \{(p_1, q_2)\} \\ \Delta &= \{[(p_0, q_0), b, (p_0, q_0)], [(p_0, q_1), b, (p_0, q_0)], [(p_0, q_2), b, (p_0, q_2)] \\ &\quad [(p_1, q_0), b, (p_1, q_0)], [(p_1, q_1), b, (p_1, q_0)], [(p_1, q_2), b, (p_1, q_2)] \\ &\quad [(p_0, q_0), a, (p_1, q_1)], [(p_0, q_1), a, (p_1, q_2)], [(p_0, q_2), a, (p_1, q_2)] \\ &\quad [(p_1, q_0), a, (p_0, q_1)], [(p_1, q_1), a, (p_0, q_2)], [(p_1, q_2), a, (p_0, q_2)]\} \end{aligned}$$

Intersection

$M_1 \cap M_2$:



$$\Delta = \{[(p_0,q_0),b,(p_0,q_0)],[(p_0,q_1),b,(p_0,q_0)],[(p_0,q_2),b,(p_0,q_2)]$$
$$[(p_1,q_0),b,(p_1,q_0)],[(p_1,q_1),b,(p_1,q_0)],[(p_1,q_2),b,(p_1,q_2)]$$
$$[(p_0,q_0),a,(p_1,q_1)],[(p_0,q_1),a,(p_1,q_2)],[(p_0,q_2),a,(p_1,q_2)]$$
$$[(p_1,q_0),a,(p_0,q_1)],[(p_1,q_1),a,(p_0,q_2)],[(p_1,q_2),a,(p_0,q_2)]\}$$

Difference (Deterministic)

$L_1 - L_2 = L_1 \cap \overline{L_2}$

Intersection and complement is closed therefore difference is closed as well.

Reversal (Non-deterministic)

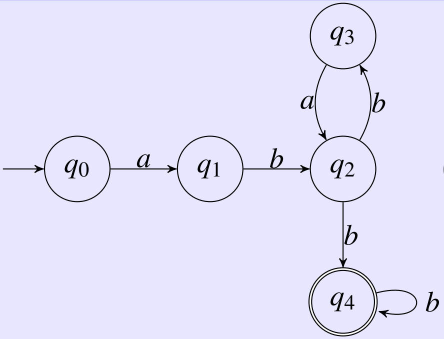
$L(M) = L(M')^R$

$S = S' \cup \{s_0\}$

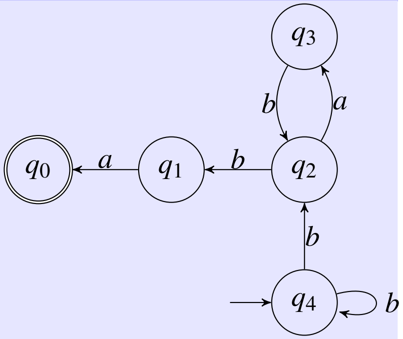
$F = s'_0$

$\delta =$
$\{[(p,\sigma,q) \in \Delta] \iff [[(q,\sigma,p) \in \Delta']\}$
$\cup$
$\{[(s_0,\Lambda,f_i) \in \Delta \quad \forall f_i \in F']\}$

Reversal



$L = ab(ba)^*b^+$



$L = b^+(ab)^*ba$

Homomorphism and Inverse Homomorphism

- Substitute each symbol in Σ by a corresponding string in Σ'^*
- $h : \Sigma \to \Sigma'^*$
- For instance
  - Let $\Sigma = \{0, 1\}$ and $\Sigma' = \{a, b\}$
  - Let a homomorphic function $h()$ on Σ be:
    - $h(0) = ab$ and $h(1) = \Lambda$
  - If $\omega = 10110$ then $h(\omega) = \Lambda ab \Lambda \Lambda ab = abab$
- If $\omega = \sigma_0 \sigma_1 \dots \sigma_n$ then $h(\omega) = h(\sigma_0) h(\sigma_1) \dots h(\sigma_n)$
- Inverse homomorphism is obtained by applying an inverse homomorphic function such as $h^{-1}()$
- $h^{-1} : \Sigma'^* \to \Sigma$

Homomorphism (Non-deterministic)

$$L(M) = h(L(M'))$$

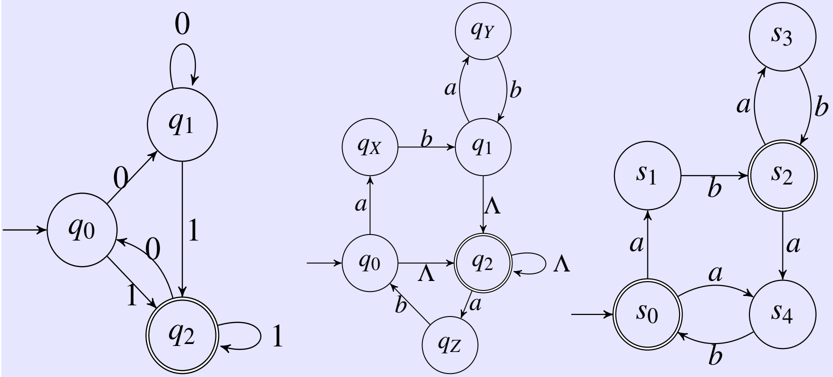
$$S = S'$$

$$F = F'$$

$$\delta = \{[(p, \sigma^*, q) \in \delta] \iff [[(p, \varsigma, q) \in \Delta' \wedge h(\varsigma) = \sigma^*]\}$$

Homomorphsim

Let's apply $h(0) = ab$ and $h(1) = \Lambda$



Inverse Homomorphism (Deterministic)

$$L(M) = h^{-1}(L(M'))$$

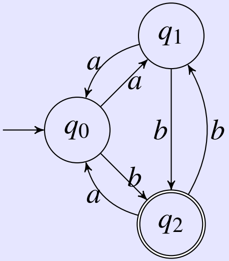
$$S = S'$$

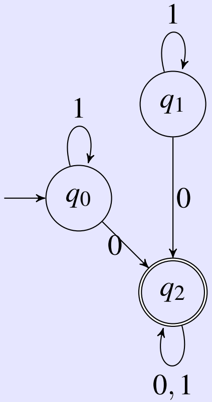
$$F = F'$$

$$\Delta =$$
$$\{[(p,\varsigma,q) \in \Delta] \iff$$
$$[[(p,\sigma_1,q_1) \wedge (q_1,\sigma_2,q_2) \wedge \ldots \wedge (q_{n-1},\sigma_n,q) \wedge h(\sigma_1\sigma_2\ldots\sigma_n) = \varsigma]$$
$$\vee$$
$$[p = q \wedge h(\Lambda) = \varsigma]]\}$$

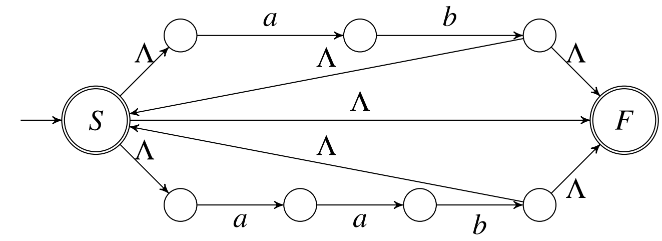
Inverse Homomorphsim

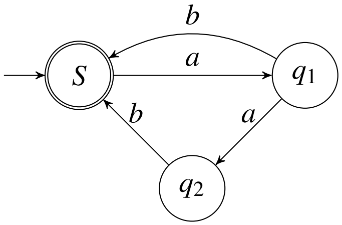
Let's apply $h^{-1}()$ where $h(0) = ab$ and $h(1) = \Lambda$





$(ab \vee aab)^*$



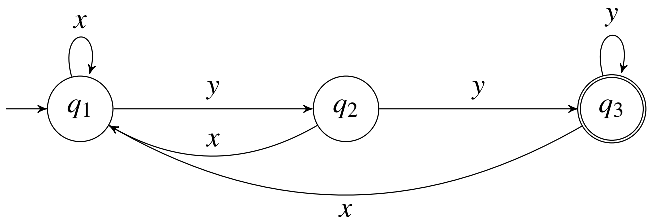


$< S >::= \Lambda \mid a < q_1 >$
$< q_1 >::= b < S > \mid a < q_2 >$
$< q_2 >::= b < S >$

Systematic way to find the regular language recognized by a DFA

Remember the theorem that states the one and only solution to the equation $X = XA \cup B \;\; \wedge \;\; \Lambda \notin A$ is $X = BA^*$.
Let's rewrite the statement using regular expressions:
$x = xa \vee b \;\; \wedge \;\; \Lambda \neq a \Rightarrow x = ba^*$
We shall use this theorem in finding the regular language recognized by a DFA

Example



$q_1 = q_1x \vee q_2x \vee q_3x \vee \Lambda$
$q_2 = q_1y$
$q_3 = q_2y \vee q_3y$
We can use the theorem for q_3
$(q_3) = (q_3)y \vee q_2y$ than we have $q_3 = q_2yy^* \Rightarrow q_3 = q_1yy^+$

Example

$(q_3) = (q_3)y \vee q_2y$ than we have $q_3 = q_2yy^* \Rightarrow q_3 = q_1yy^+$
Using this equality:
$q_1 = q_1x \vee q_1yx \vee q_1yy^+x \vee \Lambda$
$q_1 = q_1(x \vee yx \vee yy^+x) \vee \Lambda$
$q_1 = (x \vee yx \vee yy^+x)^* = (y^*x)^*$
$q_3 = (y^*x)^*yy^+$
The example automaton is actually the DFA equivalent of the NFA given in the previous examples. Heuristically we have found $(x \vee y)^*yy^+$ as the language of the NFA. Let's show that these two are equivalent.

Proof of Example

We need to show $(y^*x)^*yy^+ = (x \vee y)^*yy^+$
$(y^*x)^* = (x \vee y)^*$

Try 1

a) Draw the DFA of $(x \vee y)^*$ and NFA of $(y^*x)^*$
b) Transform the NFA to DFA
c) FAIL!!!

Proof of Example

We need to show $(y^*x)^*yy^+ = (x \vee y)^*yy^+$

Try 2

a) Draw the NFA of $(y^*x)^*yy^+$ and NFA of $(x \vee y)^*yy^+$

b) Transform each NFA to DFA

c) Wait... but how???

Proof of Example

We need to show $(y^*x)^*yy^+ = (x \vee y)^*yy^+$

Try 3

$$(y^*x)^*yy^+ \stackrel{?}{=} (x \vee y)^*yy^+$$
$$(y^*x)^*yyy^* \stackrel{?}{=} (x \vee y)^*yy^+$$
$$(y^*x)^*y^*yy \stackrel{?}{=} (x \vee y)^*yy^+$$
$$(y^*x)^*y^*y^*yy \stackrel{?}{=} (x \vee y)^*yy^+$$
$$(y^*x)^*y^*yyy^* \stackrel{?}{=} (x \vee y)^*yy^+$$
$$(y^*x)^*y^*yy^+ \stackrel{?}{=} (x \vee y)^*yy^+$$
$$(y^*x)^*y^* \stackrel{?}{=} (x \vee y)^*$$

Proof of Example

We need to show $(y^*x)^*yy^+ = (x \vee y)^*yy^+$
$(y^*x)^*y^* = (x \vee y)^*$

Try 3

a) Draw the NFA of $(y^*x)^*y^*$ and DFA of $(x \vee y)^*$

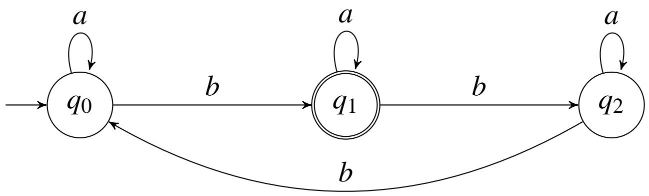
b) Transform the NFA to DFA

c) Simplify the resulting DFA

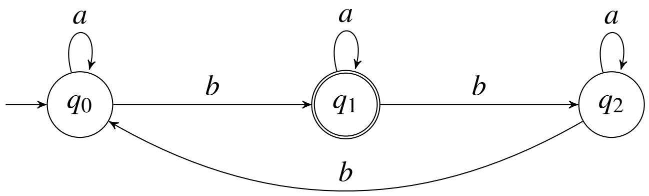
d) Surprise!!!

Example

Construct an automaton recognizing strings containing $3k+1$ b symbols and discover the corresponding regular expression.
$\Sigma = \{a, b\}$


$$\begin{tikzcd}
\arrow[r] & q_0 \arrow[loop above, "a"] \arrow[r, "b"] & q_1 \arrow[loop above, "a"] \arrow[r, "b"] & q_2 \arrow[loop above, "a"] \arrow[ll, bend left, "b"]
\end{tikzcd}$$

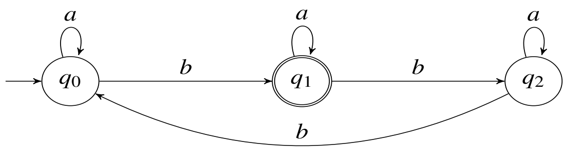
Example



$$\boxed{\begin{aligned} q_0 &= q_0a \vee q_2b \vee \Lambda \\ q_1 &= q_0b \vee q_1a \\ q_2 &= q_1b \vee q_2a \end{aligned}}$$

$$\begin{array}{l} q_2 = q_2a \vee q_1b \Rightarrow q_2 = q_1ba^* \\ q_1 = q_1a \vee q_0b \Rightarrow q_1 = q_0ba^* \\ \hline q_2 = q_0(ba^*)(ba^*) \end{array}$$

Example



$$\boxed{\begin{aligned} q_0 &= q_0 a \vee q_2 b \vee \Lambda \\ q_1 &= q_0 b \vee q_1 a \\ q_2 &= q_1 b \vee q_2 a \end{aligned}}$$

$$\boxed{\begin{aligned} q_2 &= q_0(ba^*)(ba^*) \\ q_1 &= q_0 ba^* \end{aligned}}$$

$q_0 = q_0 a \vee q_0(ba^*)^2 b \vee \Lambda$
$q_0 = q_0(a \vee (ba^*)^2 b) \vee \Lambda$
$q_0 = (a \vee (ba^*)^2 b)^*$
$q_1 = (a \vee (ba^*)^2 b)^* ba^*$
Another possible reg-ex might be: $a^* ba^* [(ba^*)^3]^*$

Section 12

Pumping Lemma for Regular Languages

Pumping Lemma for Regular Languages

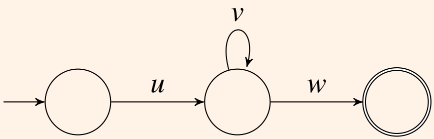
Theorem

If L is a regular language with unrestricted word length than in this language we can build any word longer than n using substrings u, v, w in following form: uv^iw. The following conditions should apply

i $|uv| \leq n$

ii $v \neq \Lambda$

iii $\forall x \in L \;\; x = uv^iw \wedge i \geq 0$



Proof

Each regular language L, can be recognized by a deterministic finite automaton.

- $M = \{S, \Sigma, \delta, s_0, F\}$
- $|S| = n$

$X = \sigma_1\sigma_2 \ldots \sigma_\ell$, is a word of L and $|X| = \ell > n$

$(s_0, \sigma_1\sigma_2 \ldots \sigma_\ell) \vdash_M (s_1, \sigma_2 \ldots \sigma_\ell) \vdash_M \ldots \vdash_M (s_{\ell-1}, \sigma_\ell) \vdash_M (s_\ell, \Lambda)$ ve $s_\ell \in F$

If $\ell > n \wedge |S| = n$ than according to the pigeonhole principle $\exists(i,j) : s_i = s_j \wedge 0 \leq i < j \leq n^4 \wedge i \neq j$

[4] If there is more than one couple, the couple with minimum $|i-j|$ should be selected.

Proof

In this case for our couple of states s_i, s_j
$\sigma_1\sigma_2\ldots\sigma_i = u$ $\sigma_i\ldots\sigma_j = v$ ve $\sigma_j\ldots\sigma_\ell = w$

X can be written as $X = uvw$; this makes $X_0 = uw$ and
$X_m = uv^m w \ \ (m \in \mathbb{N}^+)$ strings to be recognized by M

If $|X_0| = |uw| \geq n$ than proof can be repeated by using uw instead of X. The diagrammatic representation of M is a connected graph which makes the longest length of a path at most n.

$$(s_0, uv^m w) \vdash_M^* (s_i, v^m w) \vdash_M^* (s_i, v^{m-1} w) \vdash_M^* (s_i, w) \vdash_M^* (s_\ell, \Lambda)$$

Proving Using Pumping Lemma

A proof using the pumping lemma that L cannot be accepted by a finite automaton is a proof by contradiction.

1. We assume, for the sake of contradiction, that L can be accepted by M, an FA with n states.
2. We try to select a string in L with length at least n so that statements in Theorem lead to a contradiction.

If we don't get a contradiction, we haven't proved anything, and so we look for a string x that will produce one.

There doesn't exist any language which doesn't satisfy "Pumping lemma".[5]

[5] One way implication

Example 1

$$L(M) = a^n b^n | n \in \mathbb{N}^+$$

Assumptions:

- Suppose that there is an FA M having n states and accepting L
- Choose $x = a^n b^n$. Then $x \in L$ and $|x| \geq n$.

By pumping lemma

- $x = uvw$ and $|uv| \leq n$
- We can get a contradiction by using any number i other than 1 for uv^iw and $v = a^m$

For example, the string uv^2w, is $a^{n+m}b^n$, obtained by inserting m additional a's into the first part of x. This is a contradiction, because the pumping lemma says $uv^2w \in L$, but $n+m \neq n$.

Example 2

$$L(M) = a^{i^2} | i \geq 0$$

Assumptions:

- Suppose that there is an FA M having n states and accepting L
- Choose $x = a^{n^2}$

By pumping lemma

- $x = uvw$ and $0 < |v| \leq n$
- $n^2 = |uvw| < |uv^2w| = n^2 + |v| \leq n^2 + n < n^2 + 2n + 1 = (n+1)^2$
- $|uv^2w|$ must be i^2 for some integer i, but there is no integer i whose square is strictly between n^2 and $(n+1)^2$

Example 3

$$L(M) = \{a^s \mid s \text{ is a prime number } \}$$

Assumptions:

- Suppose that there is an FA M having n states and accepting L
- Let's choose $x = a^p a^q a^r$ and $p, q \geq 0 \;\; r > 0$
- Assume $p + q + r$ is prime

By pumping lemma

- $x = uvw$ ve $0 < |v| \leq n$
- $uv^2w = a^p a^q a^r$
- For $x = uv^{(p+q+r+1)}w$ and $|x| = p + (p + q + r + 1)q + r = (q + 1)(p + q + r)$ is not a prime number.

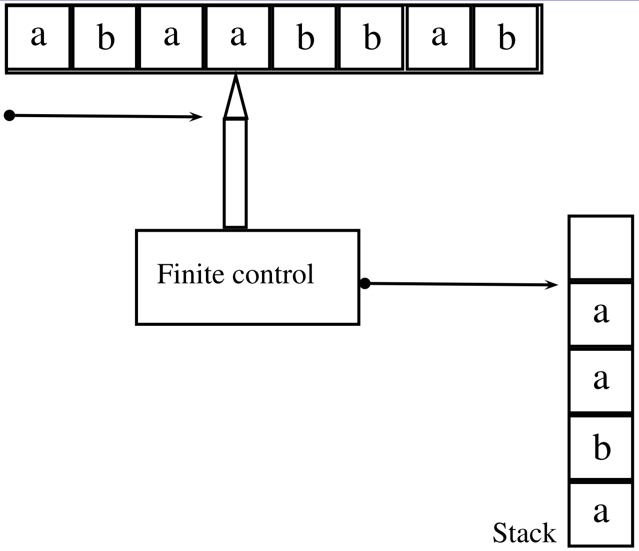
Section 13

Pushdown Automata

It is not possible to design finite automata for every context-free language. For instance the recognizer for the language $\omega\omega^R|\omega \in \Sigma^*$ should contain a memory. We can design a pushdown automaton for every context-free language.

Pushdown Automata

A pushdown automaton is similar in some respects to a finite automaton but has an auxiliary memory that operates according to the rules of a stack. The default mode in a pushdown automaton (PDA) is to allow nondeterminism, and unlike the case of finite automata, the nondeterminism cannot always be eliminated.



PDAs are not deterministic. Input strip is only used to read input while the stack can be written and read from.

Formal Definition of a PDA

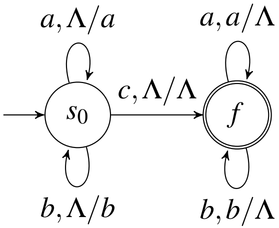
A pushdown automaton (PDA) is a 6-tuple $M = (S, \Sigma, \Gamma, \delta, s_0, F)$, where:

- S: A finite, non-empty set of states where $s \in S$.
- Σ: Input alphabet (a finite, non-empty set of symbols)
- Γ: Stack alphabet
- $s_0 \in S$: An initial state, an element of S.
- δ: The state-transition relation
  $\delta \subseteq (S \times \Sigma \cup \{\Lambda\} \times \Gamma \cup \{\Lambda\}) \times (S \times \Gamma^*)$
- F: The set of final states where $F \subseteq S$.

An example

$$(\omega c \omega^R | \omega \in \{a,b\}^*)$$
$$M = (S, \Sigma, \Gamma, \delta, s_0, F)$$
$$S = \{s_0, f\},\ \Sigma = \{a,b,c\},\ \Gamma = \{a,b\},\ F = \{f\}$$
$$\delta = \{[(s_0,a,\Lambda),(s_0,a)],[(s_0,b,\Lambda),(s_0,b)],[(s_0,c,\Lambda),(f,\Lambda)],$$
$$[(f,a,a),(f,\Lambda)],[(f,b,b),(f,\Lambda)]\}$$

state	tape	stack	trans. rule
s_0	abb **c** bba	Λ	[(s_0,a,Λ),(s_0,a)]
s_0	bb **c** bba	a	[(s_0,b,Λ),(s_0,b)]
s_0	b **c** bba	ba	[(s_0,b,Λ),(s_0,b)]
s_0	**c** bba	bba	[(s_0,c,Λ),(f,Λ)]
f	bba	bba	[(f,b,b),(f,Λ)]
f	ba	ba	[(f,b,b),(f,Λ)]
f	a	a	[(f,a,a),(f,Λ)]
f	Λ	Λ	



An example

state	tape	stack	trans. rule
s	abb **c** bba	Λ	[(s,a,Λ),(s,a)]
s	bb **c** bba	a	[(s,b,Λ),(s,b)]
s	b **c** bba	ba	[(s,b,Λ),(s,b)]
s	**c** bba	bba	[(s,c,Λ),(f,Λ)]
f	bba	bba	[(f,b,b),(f,Λ)]
f	ba	ba	[(f,b,b),(f,Λ)]
f	a	a	[(f,a,a),(f,Λ)]
f	Λ	Λ	

$$G = (N, \Sigma, n_0, \mapsto)$$
$$N = \{S\}$$
$$\Sigma = \{a, b, c\}$$
$$n_0 = S$$
$$< S > ::= a < S > a \mid b < S > b \mid c$$

Definitions

Push: To add a symbol to the stack $[(p,u,\Lambda),(q,a)]$

Pop: To remove a symbol from the stack $[(p,u,a),(q,\Lambda)]$

Configuration: An element of $S \times \Sigma^* \times \Gamma^*$. For instance (q, xyz, abc) where a is the top of the stack, c is the bottom of the stack.
Instantaneous description (to yield in one step):

Let $[(p,u,\beta),(q,\gamma)] \in \delta$ and $\forall x \in \Sigma^* \wedge \forall \alpha \in \Gamma^*$

$(p, ux, \beta\alpha) \vdash_M (q, x, \gamma\alpha)$
Here u is read from the input tape and β is read from the stack while γ is written to the stack.

Definitions

$(p, ux, \beta\alpha) \vdash_M (q, x, \gamma\alpha)$

Let ${\vdash_M}^*$ be the reflexive transitive closure of $\vdash_M$ and let $\omega \in \Sigma^*$ and s_0 be the initial state. For M automaton to accept ω string:

$(s, \omega, \Lambda){\vdash_M}^*(p, \Lambda, \Lambda)$ and $p \in F$
$C_0 = (s, \omega, \Lambda)$ and $C_n = (p, \Lambda, \Lambda)$ where
$C_0 \vdash_M C_1 \vdash_M \ldots \vdash_M C_{n-1} \vdash_M C_n$

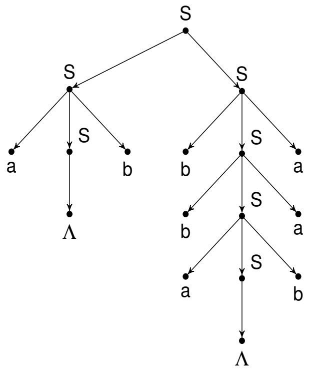
This operation is called *computation* of automaton M, this computation invloves n steps.

Let $L(M)$ be the set of string accepted by M.
$L(M) = \{\omega | (s, \omega, \Lambda){\vdash_M}^*(p, \Lambda, \Lambda) \wedge p \in F\}$

Example 1

$\omega \in \{\{a,b\}^* | \#(a) = \#(b)\}$
$M = (S, \Sigma, \Gamma, \delta, s_0, F)$
$\delta = \{[(s,\Lambda,\Lambda),(q,c)],[(q,a,c),(q,ac)],[(q,a,a),(q,aa)],$
$[(q,a,b),(q,\Lambda)],[(q,b,c),(q,bc)],[(q,b,b),(q,bb)],$
$[(q,b,a),(q,\Lambda)],[(q,\Lambda,c),(f,\Lambda)]\}$

state	tape	stack	trans. rule
s	abbbabaa	Λ	[(s,Λ,Λ),(q,c)]
q	abbbabaa	c	[(q,a,c),(q,ac)]
q	bbbabaa	ac	[(q,b,a),(q,Λ)]
q	bbabaa	c	[(q,b,c),(q,bc)]
q	babaa	bc	[(q,b,b),(q,bb)]
q	abaa	bbc	[(q,a,b),(q,Λ)]
q	baa	bc	[(q,b,b),(q,bb)]
q	aa	bbc	[(q,a,b),(q,Λ)]
q	a	bc	[(q,a,b),(q,Λ)]
q	Λ	c	[(q,Λ,c),(f,Λ)]
f	Λ	Λ	



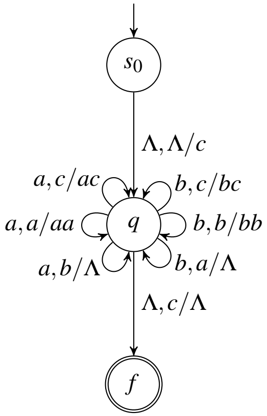
Example 1

$\omega \in \{\{a,b\}^* | \#(a) = \#(b)\}$

$M = (S, \Sigma, \Gamma, \delta, s_0, F)$

$\delta = \{[(s,\Lambda,\Lambda),(q,c)], [(q,a,c),(q,ac)], [(q,a,a),(q,aa)],$
$[(q,a,b),(q,\Lambda)], [(q,b,c),(q,bc)], [(q,b,b),(q,bb)],$
$[(q,b,a),(q,\Lambda)], [(q,\Lambda,c),(f,\Lambda)]\}$

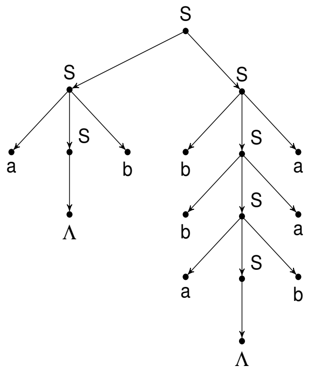
state	tape	stack	trans. rule
s	abbbabaa	Λ	[(s,Λ,Λ),(q,c)]
q	abbbabaa	c	[(q,a,c),(q,ac)]
q	bbbabaa	ac	[(q,b,a),(q,Λ)]
q	bbabaa	c	[(q,b,c),(q,bc)]
q	babaa	bc	[(q,b,b),(q,bb)]
q	abaa	bbc	[(q,a,b),(q,Λ)]
q	baa	bc	[(q,b,b),(q,bb)]
q	aa	bbc	[(q,a,b),(q,Λ)]
q	a	bc	[(q,a,b),(q,Λ)]
q	Λ	c	[(q,Λ,c),(f,Λ)]
f	Λ	Λ	



Example 1

$\omega \in \{\{a,b\}^*|\#(a) = \#(b)\}$
$M = (S,\Sigma,\Gamma,\delta,s_0,F)$
$\delta = \{[(s,\Lambda,\Lambda),(q,c)],[(q,a,c),(q,ac)],[(q,a,a),(q,aa)],$
$[(q,a,b),(q,\Lambda)],[(q,b,c),(q,bc)],[(q,b,b),(q,bb)],$
$[(q,b,a),(q,\Lambda)],[(q,\Lambda,c),(f,\Lambda)]\}$

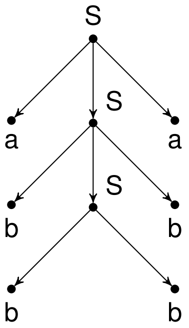
$G = (N,\Sigma,n_0,\mapsto)$
$N = \{S\}$
$\Sigma = \{a,b\}$
$n_0 = S$
$<S>::= a<S>b \mid b<S>a \mid <S><S> \mid \Lambda$



Example 2

$\omega \in \{xx^R | x \in \{a,b\}^*\}$
$M = (S, \Sigma, \Gamma, \delta, s_0, F)$
$\delta =$
$\{[(s,a,\Lambda),(s,a)],[(s,b,\Lambda),(s,b)],[(s,\Lambda,\Lambda),(f,\Lambda)],[(f,a,a),(f,\Lambda)],[(f,b,b),(f,\Lambda)]\}$

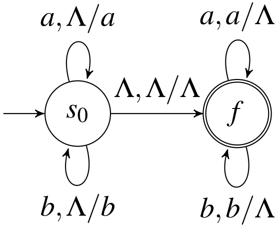
state	tape	stack	trans. rule
s	abbbba	Λ	[(s,a,Λ),(s,a)]
s	bbbba	a	[(s,b,Λ),(s,b)]
s	bbba	ba	[(s,b,Λ),(s,b)]
s	bba	bba	[(s,Λ,Λ),(f,Λ)]
f	bba	bba	[(f,b,b),(f,Λ)]
f	ba	ba	[(f,b,b),(f,Λ)]
f	a	a	[(f,a,a),(f,Λ)]
f	Λ	Λ	



Example 2

$\omega \in \{xx^R | x \in \{a,b\}^*\}$
$M = (S, \Sigma, \Gamma, \delta, s_0, F)$
$\delta =$
$\{[(s,a,\Lambda),(s,a)],[(s,b,\Lambda),(s,b)],[(s,\Lambda,\Lambda),(f,\Lambda)],[(f,a,a),(f,\Lambda)],[(f,b,b),(f,\Lambda)]\}$

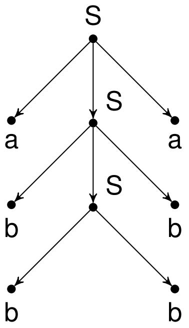
state	tape	stack	trans. rule
s	abbbba	Λ	[(s,a,Λ),(s,a)]
s	bbbba	a	[(s,b,Λ),(s,b)]
s	bbba	ba	[(s,b,Λ),(s,b)]
s	bba	bba	[(s,Λ,Λ),(f,Λ)]
f	bba	bba	[(f,b,b),(f,Λ)]
f	ba	ba	[(f,b,b),(f,Λ)]
f	a	a	[(f,a,a),(f,Λ)]
f	Λ	Λ	



Example 2

$\omega \in \{xx^R | x \in \{a,b\}^*\}$
$M = (S, \Sigma, \Gamma, \delta, s_0, F)$
$\delta =$
$\{[(s,a,\Lambda),(s,a)],[(s,b,\Lambda),(s,b)],[(s,\Lambda,\Lambda),(f,\Lambda)],[(f,a,a),(f,\Lambda)],[(f,b,b),(f,\Lambda)]\}$

$G = (N, \Sigma, n_0, \mapsto)$
$N = \{S\}$
$\Sigma = \{a,b\}$
$< S > ::= a < S > a \mid b < S > b \mid aa \mid bb$



Deterministic PDA

Deterministic PDA

1) $\forall s \in S \wedge \forall \gamma \in \Gamma$ if $\delta(s,\Lambda,\gamma) \neq \varnothing \Rightarrow \delta(s,\sigma,\gamma) = \varnothing; \forall \sigma \in \Sigma$
2) If $a \in \Sigma \cup \{\Lambda\}$ then $\forall s, \forall \gamma$ and $\forall a$ Card$(\delta(s,a,\gamma)) \leq 1$

- In a DPDA
  1. If there exists a lambda transition(yielding in one step) in a configuration no other transitions should be present for any other input.
  2. There should be a unique transition for any (s,σ,γ) tuple
- For nondeterministic PDA, the equivalence problem to deterministic PDA is proven to be undecidable[6].

[6] An undecidable problem is a decision problem for which it is impossible to construct a single algorithm that always leads to a correct yes-or-no answer

Deterministic PDA

- The way a DPDA works is exactly the same as an NPDA, with several modes of acceptance: acceptance on final state, acceptance on empty stack, and acceptance on final state and empty stack.
- However, unlike a NPDA, these acceptance methods are not equivalent. It can be shown that the set of languages accepted on empty stack is a proper subset of the set of languages determined on final state.
- Any regular language can be accepted by a DPDA on empty stack
- Any language accepted by a DPDA on final state is unambiguous, note that the reverse may not hold all the time.

Deterministic PDA

Example 1

The set of palindromes is unambiguous, but not deterministic. The language of even length palindromes $\omega\omega^R$ cannot be recognized by a DPDA, it can only be recognized by an NPDA.

Example 2

The language $L = \{a^m b^n | m \geq n \geq 0\}$ is deterministic, but not prefix-free[7], and hence can not be accepted by any DPDA on empty stack

Example 3

The language $\{a^n b^n | n \geq 0\}$ can be accepted by a DPDA on empty stack, and the language is not regular.

[7] prefix property: There is no whole code word in the system that is a prefix of any other code word in the system

Section 14

Equivalence of PDAs and CFGs

Converting CFGs to PDAs

- Given a CFG G, any left sentential form can be written as $xA\alpha$ where $x \in \Sigma^*$ and $A \in N$ and $\alpha \in V$.
- We construct the PDA that corresponds to G by having PDA simulate the sequence of left-sentential forms that the grammar uses to generate a given terminal string ω in a non-deterministic way.
- $A\alpha$ of each sentential form appears in the stack with A on top while x is being consumed by our PDA.

Converting CFGs to PDAs

Let $G = \{N, \Sigma, n_0, \mapsto\}$ be a CFG. We construct the PDA that accepts $L(G)$ as follows:

$$P = (\{q,f\}, \Sigma, N \cup \Sigma \cup Z_0, \delta, q, f)$$

where δ is defined by

1. An initial mark Z_0 for the empty stack such as $(p, \Lambda, \Lambda) = (q, Z_0)$
2. A rule for pushing the initial rule to the stack such as $(p, \Lambda, Z_0) = (p, SZ_0)$
3. For each variable A, $\delta(q, \Lambda, A) = \{(q, \beta) | A \rightarrow \beta$ is a production of $G\}$
4. For each terminal $\sigma, \delta(q, \sigma, \sigma) = \{q, \Lambda\}$
5. A transition for acceptance with empty stack such as $\delta(q, \Lambda, Z_0) = \{(f, \Lambda)\}$

Converting CFGs to PDAs

$$E \to E+E \mid E\times E \mid (E) \mid F$$
$$F \to xF \mid yF \mid 0F \mid 1F$$
$$F \to x \mid y \mid 0 \mid 1$$

The transition function of the PDA is

- $\delta(p,\Lambda,\Lambda) = (q,Z_0)$; $\delta(q,\Lambda,Z_0) = (q,EZ_0)$
- $\delta(q,\Lambda,F) = \{(q,x),(q,y),(q,0),(q,1)\}$
- $\delta(q,\Lambda,F) = \{(q,xF),(q,yF),(q,0F),(q,1F)\}$
- $\delta(q,\Lambda,E) = \{(q,F),(q,E+E),(q,E\times E),(q,(E))\}$
- $\delta(q,x,x) = (q,\Lambda)$; $\delta(q,y,y) = (q,\Lambda)$
- $\delta(q,0,0) = (q,\Lambda)$; $\delta(q,1,1) = (q,\Lambda)$
- $\delta(q,(,() = (q,\Lambda)$; $\delta(q,),)) = (q,\Lambda)$
- $\delta(q,+,+) = (q,\Lambda)$; $\delta(q,\times,\times) = (q,\Lambda)$
- $\delta(q,\Lambda,Z_0) = \{(f,\Lambda)\}$

Converting PDAs to CFGs

- Main idea is to reverse engineer the productions from transitions.
- For $\delta(p,a,Z) = (q, Y_1Y_2Y_3\dots Y_k)$
  1. State is changed from p to q
  2. Terminal a is consumed
  3. Stack top Z is popped and replaced with a sequence of k variables/symbols
- For each δ rule stated as above we create the rule in grammar
  - $[pZq] \rightarrow a[qY_1p_1][p_1Y_2p_2][p_2Y_3p_3]\dots[p_{k-1}Y_kp_k]$
  - Unless $Y_1Y_2Y_3\dots Y_k = \Lambda$. This case we just add $[pZq] \rightarrow a$

Converting PDAs to CFGs

Following PDA checks for the matching parantheses in a string. To avoid confusion we represent a left paranthesis with l and a right paranthesis with r symbol.

1. $\delta(q_0, \ell, \Lambda) = (q_0, \ell)$
2. $\delta(q_0, \ell, \ell) = (q_0, \ell\ell)$
3. $\delta(q_0, r, \ell) = (q_0, \Lambda)$

We produce the following rules correspondingly

1. $[q_0 q_0] \rightarrow \ell[q_0 \ell q_0]$
2. $[q_0 \ell q_0] \rightarrow \ell[q_0 \ell q_0][q_0 \ell q_0]$
3. $[q_0 \ell q_0] \rightarrow r$

If we substitute stated non-terminals with simpler labels

1. $A \rightarrow \ell B$
2. $B \rightarrow \ell BB$
3. $B \rightarrow r$

Section 15

CFL Properties

Closure of CFLs

- CFL's are closed under
  (a) Union
  (b) Concatenation
  (c) Kleene star
  (d) Reversal
  (e) Homomorphism and inverse homomorphsim
- CFL's are **not** closed under
  (a) Intersection
  (b) Complement
  (c) Difference

Closure Under Union

Union

$$G_1 = \{N_1, \Sigma_1, n_{01}, \mapsto_1\}$$
$$G_2 = \{N_2, \Sigma_2, n_{02}, \mapsto_2\}$$

$$G = L(G_1) \cup L(G_2)$$

$$G = \{N, \Sigma, n_0, \mapsto\}$$

$N = N_1 \uplus$[8] N_2

$$\Sigma = \Sigma_1 \uplus \Sigma_2$$
$$\mapsto = \mapsto_1 \uplus \mapsto_2 \cup \{n_0 \to n_{01} \mid n_{02}\}$$

[8] Disjoint Union operation

Closure Under Concatenation

Concatenation

$$G_1 = \{N_1, \Sigma_1, n_{01}, \mapsto_1\}$$
$$G_2 = \{N_2, \Sigma_2, n_{02}, \mapsto_2\}$$

$$G = L(G_1) \cup L(G_2)$$

$$G = \{N, \Sigma, n_0, \mapsto\}$$

$$N = N_1 \uplus N_2$$
$$\Sigma = \Sigma_1 \uplus \Sigma_2$$
$$\mapsto = \mapsto_1 \uplus \mapsto_2 \cup \{n_0 \to n_{01} n_{02}\}$$

Closure Under Kleene Star

Kleene Star

$G_1 = \{N_1, \Sigma_1, n_{01}, \mapsto_1\}$

$G = L(G_1)^*$

$G = \{N, \Sigma, n_0, \mapsto\}$

$N = N_1$
$\Sigma = \Sigma_1$
$\mapsto = \mapsto_1 \cup \{n_0 \to n_{01} n_0 \mid \Lambda\}$

Closure Under Reversal

Reversal

$G_1 = \{N_1, \Sigma_1, n_{01}, \mapsto_1\}$

$G = L(G_1)^R$

$G = \{N, \Sigma, n_0, \mapsto\}$

$N = N_1$
$\Sigma = \Sigma_1$
$n_0 = n_{01}$
$\mapsto = \forall[(\omega_l \to \omega_r) \in \mapsto_1] \leftrightarrow [(\omega_l \to \omega_r^R) \in \mapsto]$

Closure Under Reversal

CFG for $\{0^m 1^n | m \geq n\}$

$$< S > ::= 0 < S > 1 \mid A$$
$$< A > ::= 0 < A > \mid \Lambda$$

CFG for $\{1^m 0^n | m < n\}$

$$< S > ::= 1 < S > 0 \mid < A >$$
$$< A > ::= < A > 0 \mid \Lambda$$

Closure Under Homomorphsim

Homomorphsim

$G_1 = \{N_1, \Sigma_1, n_{01}, \mapsto_1\}$

$G = h(L(G_1))$

$G = \{N, \Sigma, n_0, \mapsto\}$

$N = N_1$
$\Sigma = h(\Sigma_1)$
$n_0 = n_{01}$
$\mapsto = \mapsto_1$ [9]

[9] with new Σ

Closure Under Inverse Homomorphsim

Inverse Homomorphsim

$M_1 = (S_1, \Sigma_1, \Gamma_1, \delta_1, s_{01}, F_1)$

$L(M) = h^{-1}(L(M_1))$

$M = (S, \Sigma, \Gamma, \delta, s_0, F)$

- There is no simpler way to demonstrate this property using CFGs so we will use PDAs.
- M accepts symbols from $h()$'s range set where each symbol correspond to a string in Σ_1
- We need to transform the states and transitions M_1 likewise.

Closure Under Inverse Homomorphsim

States of M

- States are $[q,b]$ pairs where:
  - q is a state of M_1
  - b is a valid suffix from $h^{-1}()$'s range.
- Since Σ_1 is finite, possible values for b is finite as well.
- Start state of M is $[s_{01},\Lambda]$
- $\forall q \in F_1 : [q,\Lambda] \in F$

- M's states act as an expander buffer for incoming string from the domain of $h^{-1}()$.
- With every incoming symbol $\sigma \in \Sigma$, M kicks in the related state which contains the corresponding state in M_1 together with $h(\sigma)$.
- M works exactly as M_1, uses the same Γ, until its buffer states consume $h(\sigma)$.

Closure Under Inverse Homomorphsim

Transitions of M

- $\delta([q,\Lambda],\sigma,\gamma) = ([q,h(\sigma),\gamma])$ for $\sigma \in \Sigma$ and $\gamma \in \Gamma$.
- $\delta([q,b\omega],\Lambda,\gamma)$ contains $([p,\omega],\varsigma)$ if $\delta_1(q,b,\gamma)$ contains (p,ς) where $\varsigma \in \Sigma_1$ and $p \in S_1$

Closure Under Intersection

- Unlike the regular languages, the CFLs are not closed under intersection
- We may prove it with a counterexample
- $L_1 = \{0^n 1^n 2^i \mid n \geq 1, i \geq 1\}$
- $L_2 = \{0^i 1^n 2^n \mid n \geq 1, i \geq 1\}$
- $L = L_1 \cap L_2$
- $L = \{0^n 1^n 2^n \mid n \geq 1\}$ which is not a CFL.

Closure Under Difference

- Any class of languages that is closed under difference is closed under intersection
- $L_1 - (L_1 - L_2) = L_1 \cap L_2$
- If CFLs were closed under difference then they would have been closed under intersection as well
- But they are not closed under intersection, nor they do under difference.

Closure Under Complement

- If CFLs were closed under complement then they would have been closed under intersection as well
- $\overline{(\overline{L_1} \cup \overline{L_2})} = L_1 \cap L_2$
- CFLs are closed under union but they are not closed under intersection, nor they do under complement.

Unsolvable (Undecidable) CFL Problems

- There are algorithms to decide, for a given grammar/PDA belonging to a CFL, if:
  - String ω is in $L(G)$
  - $L(G)$ is empty
  - $L(G)$ is infinite
- There are also questions which is proven to be unsolvable, such as:
  - Is a given CFG G ambiguous?
  - Is it possible to provide an unambiguous grammar for a given CFL L ambiguous?
  - Are two CFL's the same?
  - Are two CFL's disjoint? (Empty intersection)
  - For a given CFG G, is $L(G)$ equal to Σ^*?

Section 16

Chomsky Normal Form

Chomsky Hierarchy

Type	Language (Grammars)	Form of Productions in Grammar	Accepting Device
0	Recursively enumerable (unrestricted)	$\alpha \to \beta$ $(\alpha, \beta \in (N \cup \Sigma)^*$, α contains a variable$)$	Turing machine
1	Context-sensitive	$\alpha \to \beta$ $(\alpha, \beta \in (N \cup \Sigma)^*, \lvert\beta\rvert \geq \lvert\alpha\rvert$, α contains a variable$)$	Linear-bounded automaton
2	Context-free	$A \to \alpha$ $(A \in N, \alpha \in (N \cup \Sigma)^*)$	Pushdown automaton
3	Regular	$A \to \sigma B, A \to \sigma$ $(A, B \in N, \sigma \in \Sigma^*)$	Finite automaton

- Questions about the strings generated by a context-free grammar G are sometimes easier to answer if we know something about the form of the productions.
- Sometimes this means knowing that certain types of productions never occur, and sometimes it means knowing that every production has a certain simple form.
- For example, suppose we want to know whether a string x is generated by G, and we look for an answer by trying all derivations.
  If we don't find a derivation that produces x, how long do we have to keep trying?

Definition

A context-free grammar is said to be in *Chomsky normal form*(CNF) if every production is of one of these two types:

- $A \to BC$ (where B and C are non-terminals)
- $A \to \sigma$ (where σ is a terminal symbol)

Theorem

For every context-free grammar G, there is another CFG G_1 in Chomsky normal form such that $L(G_1) = L(G) - \{\Lambda\}$.

CNF Transformation

Following algorithm that can be used to construct the CNF grammar G_1 from a Type-2 grammar G:

1. Eliminate unit productions **and then** Λ productions.
2. Break-down productions whose right side has at least two terminal symbols.
3. Replace each production having more than two non-terminal occurrences on the right by an equivalent set of double-non-terminal productions.

Example CNF Transformation

CNF Transformation

CNF grammar G_1 from a Type-2 grammar G:

A Type-2 Grammar

$S = a, b, c$
$N = S, T, U, V, W$
$\mapsto = \{$
S $\rightarrow$ TU | V
T $\rightarrow$ aTb | Λ
U $\rightarrow$ cU | Λ
V $\rightarrow$ aVc | W
W $\rightarrow$ bW | Λ
$\}$
which generates the language $a^i b^j c^k | i = j$ or $i = k$.

CNF Transformation

1. Eliminate unit productions **and then** Λ productions.
   Unit production ex.: $V \rightarrow W$
   Λ production ex.: $T \rightarrow \Lambda$

A Type-2 Grammar

S $\rightarrow$ TU | V
T $\rightarrow$ aTb | Λ
U $\rightarrow$ cU | Λ
V $\rightarrow$ aVc | W
W $\rightarrow$ bW | Λ

Unit productions eliminated

S $\rightarrow$ TU | aVc | bW | Λ
T $\rightarrow$ aTb | Λ
U $\rightarrow$ cU | Λ
V $\rightarrow$ aVc | bW | Λ
W $\rightarrow$ bW | Λ

CNF Transformation

1. Eliminate unit productions **and then** Λ productions.
   Unit production ex.: $V \to W$
   Λ production ex.: $T \to \Lambda$

A Type-2 Grammar

S $\to$ TU | aVc | bW
T $\to$ aTb | Λ
U $\to$ cU | Λ
V $\to$ aVc | bW | Λ
W $\to$ bW | Λ

Λ productions eliminated

S $\to$ TU | aVc | bW | aTb | ab | cU | c | b | ac
T $\to$ aTb | ab
U $\to$ cU | c
V $\to$ aVc | bW | b | ac
W $\to$ bW | b

CNF Transformation

2. Break-down productions whose right side has at least two terminal symbols.
   Introduce for every terminal symbol σ a variable X_σ and a production rule $X_\sigma \rightarrow \sigma$

A Type-2 Grammar

S $\rightarrow$ TU | aTb | aVc | cU
S $\rightarrow$ ab | ac | c | b | bW
T $\rightarrow$ aTb | ab
U $\rightarrow$ cU | c
V $\rightarrow$ aVc | ac | bW | b
W $\rightarrow$ bW | b

Non-single-terminals eliminated

S $\rightarrow$ TU | X_aTX_b | X_aVX_c | X_cU
S $\rightarrow$ X_a X_b | X_a X_c | c | b | X_bW
T $\rightarrow$ X_aTX_b | X_a X_b
U $\rightarrow$ X_cU | c
V $\rightarrow$ X_aVX_c | X_a X_c | X_bW | b
W $\rightarrow$ X_bW | b
$X_a \rightarrow$ a
$X_b \rightarrow$ b
$X_c \rightarrow$ c

CNF Transformation

3. Replace each production having more than two non-terminal occurrences on the right by an equivalent set of double-non-terminal productions.

A Type-2 Grammar

S $\rightarrow$ TU | X_aTX_b | X_cU | X_aVX_c
S $\rightarrow$ X_a X_b | X_a X_c | c | b | X_bW
T $\rightarrow$ X_aTX_b | X_a X_b
U $\rightarrow$ X_cU | c
V $\rightarrow$ X_aVX_c | X_a X_c | X_bW | b
W $\rightarrow$ X_bW | b
$X_a \rightarrow$ a
$X_b \rightarrow$ b
$X_c \rightarrow$ c

Non-double-non-terminals elim.

S $\rightarrow$ TU | X_a Y_1 | X_cU | X_a Y_2
$Y_1 \rightarrow$ TX_b
$Y_2 \rightarrow$ VX_c
S $\rightarrow$ X_a X_b | X_a X_c | X_bW | b | c
T $\rightarrow$ X_a Y_1 | X_a X_b
U $\rightarrow$ X_cU | c
V $\rightarrow$ X_a Y_2 | X_a X_c | X_bW | b
W $\rightarrow$ X_bW | b
$X_a \rightarrow$ a
$X_b \rightarrow$ b
$X_c \rightarrow$ c

Another Example

1. Eliminate unit productions

A Type-2 Grammar

S $\rightarrow$ AaA | bA | BaB
A $\rightarrow$ aaBa | CDA | aa | DC
B $\rightarrow$ bB | bAB | bb | aS
C $\rightarrow$ Ca | bC | D
D $\rightarrow$ bD | Λ

Unit productions partly eliminated

S $\rightarrow$ AaA | bA | BaB
A $\rightarrow$ aaBa | CDA | aa | DC
B $\rightarrow$ bB | bAB | bb | aS
C $\rightarrow$ Ca | bC | bD | Λ
D $\rightarrow$ bD | Λ

1. Eliminate Λ productions.

A Type-2 Grammar

S $\rightarrow$ AaA | bA | BaB
A $\rightarrow$ aaBa | CDA | aa | DC
B $\rightarrow$ bB | bAB | bb | aS
C $\rightarrow$ Ca | bC | bD | Λ
D $\rightarrow$ bD | Λ

Λ productions partly eliminated

S $\rightarrow$ AaA | bA | BaB
A $\rightarrow$ aaBa | CDA | aa | DC | CA
A $\rightarrow$ DA | C | D | Λ
B $\rightarrow$ bB | bAB | bb | aS
C $\rightarrow$ Ca | bC | bD | b | a
D $\rightarrow$ bD | b

1. Eliminate Λ productions.

A Type-2 Grammar

S $\rightarrow$ AaA | bA | BaB
A $\rightarrow$ aaBa | CDA | aa | DC | CA
A $\rightarrow$ DA | C | D | Λ
B $\rightarrow$ bB | bAB | bb | aS
C $\rightarrow$ Ca | bC | bD | b | a
D $\rightarrow$ bD | b

Λ productions eliminated

S $\rightarrow$ AaA | bA | BaB | a | b
A $\rightarrow$ aaBa | CDA | aa | DC | CA
A $\rightarrow$ DA | C | D | CD
B $\rightarrow$ bB | bAB | bb | aS
C $\rightarrow$ Ca | bC | bD | b | a
D $\rightarrow$ bD | b

1. Eliminate unit productions.

A Type-2 Grammar

S → AaA | bA | BaB | a | b
A → aaBa | CDA | aa | DC | CA
A → DA | CD | C | D
B → bB | bAB | bb | aS
C → Ca | bC | bD | b | a
D → bD | b

Unit productions eliminated

S → AaA | bA | BaB | a | b
A → aaBa | CDA | DC | CA | aa
A → DA | CD | Ca | bC | bD | a | b
B → bB | bAB | aS | bb
C → Ca | bC | bD | a | b
D → bD | b

Transformation continues with the 2nd and 3rd steps.

Section 17

Pumping Lemma for Context-Free Languages

Consider the following languages. Can you design a PDA to recgonize them.

Example 1

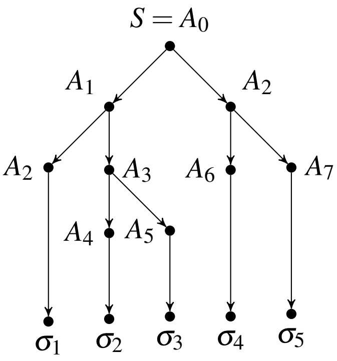
$L(G_1) = \{a^n b^n c^n | n \geq 0\}$

Example 2

$L(G_2) = \{xx \mid x \in \{a,b\}^*\}$

Some languages require more capable memory architectures than a stack to be recognized!

Pumping Lemma for Context-Free Languages



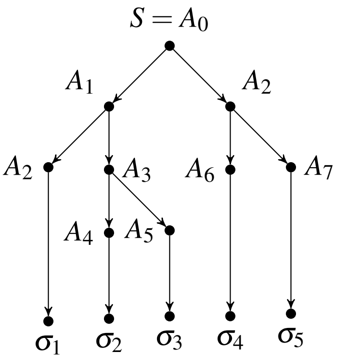
Chomsky Normal Form

The parse tree of a CNF defined context free language is a binary tree. When the leaves of the parse tree is traversed from left to right a word of the language is formed.

Theorem

Gor every context fee grammar G, there is another grammar in CNF such as $L(G_1) = L(G) - \Lambda$. In G_1 grammar rules there exists either a couple of non-terminals or a single terminal.

Pumping Lemma for Context-Free Languages



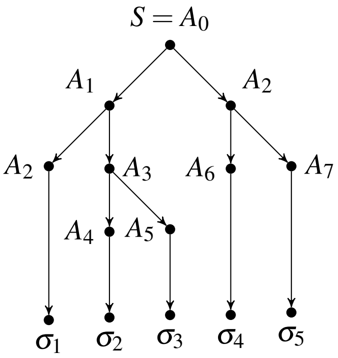
Theorem

Let's assume we produce the string $u = \sigma_1\sigma_2 \ldots \sigma_i \ldots \sigma_n \quad (\sigma_i \in \Sigma)$ for a parse tree produced by a CNF grammar such as G_{CNF}.

If we write u using only the non-terminals that directly produce terminals
$u = A_1A_2 \ldots A_n$

e.g. $S = \sigma_1\sigma_2\sigma_3\sigma_4\sigma_5$ ya da
e.g. $S = A_2A_4A_5A_6A_7$

Pumping Lemma for Context-Free Languages



The parse tree of the grammar can be a complete binary tree for the most extreme case. In such a situation all the paths that navigate to the leaves of this tree are equals to the depth of this tree[10]. In a complete binary tree if the depth is p than the number of leaves is equal to 2^p.

In a CNF grammar's parse tree each leaf is a single child of a non-terminal parent increasing the tree's depth by one ($h = p + 1$). On the other hand the length of the produced word is the number of leaves in a complete binary tree of depth p, which is ($|w| = 2^{h-1} = 2^p$).

Tree depth: The longest path from a leaf to the root

Pumping Lemma for Context-Free Languages

In that case we can at most produce a word of length 2^p from a tree of depth $p+1$. In this tree

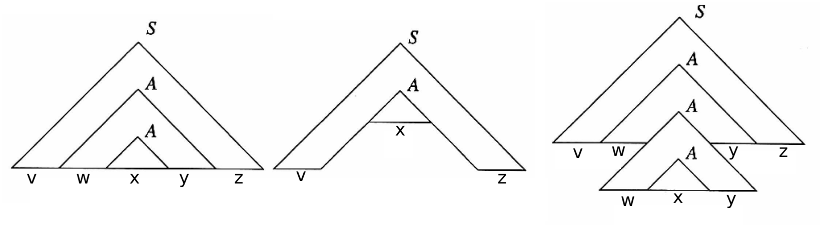
1. If each inner node does not correspond to a different non-terminal, in other words there exists a rule repetition, there exists substrings in the word that can be repeated (pumped).
2. If each inner node corresponds to a different non-terminal then there will be a repetition when producing a word of length larger than 2^p.

Theorem

Let L be a context-free language. For all strings(u) that satisfy $u \in L$ and $|u| \geq n$, string u can be written as $u = vwxyz$ where v, w, x, y, and z satisfies the following:

1. $|wy| > 0$
2. $|wxy| \leq n = 2^p$
3. For all $m \geq 0$, $vw^m xy^m z \in L$.

Pumping Lemma for Context-Free Languages



Example 1

$$L = \{a^m b^m c^m | m \geq 0\}$$

Assumptions:

- L has a CFG
- $u = vwxyz$ and $|u| \geq n$.
- n corresponds to the number of non-terminals that produce this string.
- $|wxy| \leq n$

$$\begin{array}{ccccccl} a^{n-1} & a & b^{n-1} & b & c^n & \quad & vw^2xy^2z = a^{n+1}b^nc^n \\ v & w & x & y & z & \quad & vxz = a^{n-1}b^{n-2}c^n \end{array}$$

Example 2

$$L = \{xx \mid x \in \{a,b\}^*\}$$

Assumptions:

- L has a CFG
- $u = a^n b^n a^n b^n$ and $|u| \geq n$
- n corresponds to the number of non-terminals.
- $|wxy| \leq n$

1. $$\begin{array}{cccccc} a^n b^3 & b & b^{n-4} & a & a^{n-1} b^n & a^n b^3 b^{n-4} a^{n-1} b^n \\ v & w & x & y & z & vw^0xy^0z \end{array}$$
2. or $$\begin{array}{cccccc} a^n b & b & b^{n-3} & b & a^n b^n & a^n b^{n-2} a^n b^n \\ v & w & x & y & z & vw^0xy^0z \end{array}$$
3. or $$\begin{array}{cccccc} a^{n-1} & a & b^{n-3} & b & b^2 a^n b^n & a^{n-1} b^{n-1} a^n b^n \\ v & w & x & y & z & vw^0xy^0z \end{array}$$

Example 3

$$L = \{x \in \{a,b,c\}^* \mid \#(a) < \#(b) \text{ and } \#(a) < \#(c)\}$$

Assumptions:

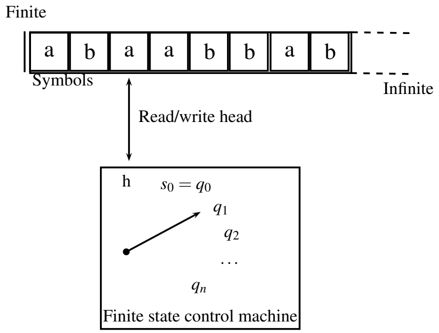
- L has a CFG
- $u = a^n b^{n+1} c^{n+1}$ and $|u| \geq n$
- n corresponds to the number of non-terminals.
- $|wxy| \leq n$

1. $\begin{array}{cccccc} a^{n-1} & a & b^{n-3} & b & b^3 c^{n+1} & a^{n+1} b^{n+2} c^{n+1} \\ v & w & x & y & z & vw^2xy^2z \end{array}$
2. or $\begin{array}{cccccc} a^n b^5 & b & b^{n-5} & c & c^n & a^n b^n c^n \\ v & w & x & y & z & vw^0xy^0z \end{array}$

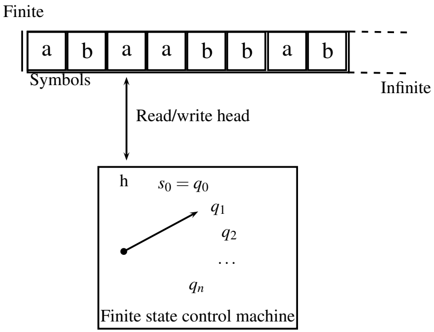
Section 18

Turing Machines

Classical automata (a.k.a. language recognizers(DFA,NFA,PDA)) sometimes is incapable of recognizing very simple language(e.g. $a^n b^n c^n : n \geq 0$).
There exists more general language recognizers which perform transformation between chains of tokens. Turing machine is an example to this kind of language recognizers.



Read/write head reads the corresponding symbol on the tape and according to the state of the control machine it either writes a new symbol, or it moves left(L) or right(R). If the machine attempts to move further left from the leftmost symbol on the tape than this situation is denoted as the “machine hangs”. Head may move as much as possible towards right.



Symbols:

- h: End of computation. Machine halts. Halt is not included in input, output and state alphabet.
- #: blank symbol
- L,R: Symbols indicating left and right movement. Those are not included in input or output alphabet.
- Input symbols are written on the leftmost side by convention.

Formal Definition of a TM

A turing machine(TM) is a quadruple $M = (S, \Sigma, \delta, s_0)$, where:

- S: A finite, non-empty set of states. Usually $h \notin S$.
- Σ: Input/output alphabet. $\# \in \Sigma$ but $L, R \notin \Sigma$
- $s_0 \in S$: An initial state, an element of S.
- δ: The state-transition function $S \times \Sigma \to S \cup \{h\} \times (\Sigma \cup \{L, R\})$

$$q \in S \wedge a \in \Sigma \wedge \delta(q,a) = (p,b)$$
$$q \to p$$

i) (Read)$a \to b \in \Sigma$ (write on tape, in place of a)
ii) $b = L$ (head left)
iii) $b = R$ (head right)

Formal Definition of a TM

We can match modern computers with Turing Machines; RAM can be matched with the tape, computer programs can be matched with the state table, microprocessor(excluding I/O untis) can be matched with control unit of the TM. In order to improve understandibility of the state table, following notion can be used:
$q, \sigma, q', \sigma', HM$ where

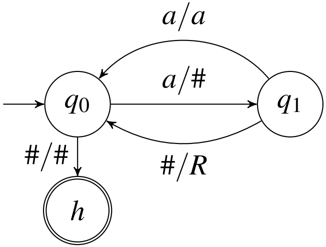
- q stands for current state of the machine
- q' stands for the next state
- σ stands for read symbol
- σ' stands for symbol to be written (same with σ for no write)
- HM stands for head move where $HM \in -1 : L, 0 : None, 1 : R$

Example 1

A machine that erases all the symbols from left to right.

$S = \{q_0, q_1\}$
$\Sigma = \{a, \#\}$
$s_0 = q_0$

q	σ	$\delta(q,\sigma)$
q_0	a	$(q_1, \#)$
q_0	$\#$	$(h, \#)$
q_1	a	(q_0, a)[1]
q_1	$\#$	(q_0, R)


$$\begin{tikzcd}
\arrow[r] & q_0 \arrow[r, "a/\#"] \arrow[d, "\#/\#"'] & q_1 \arrow[l, bend right, "a/a"'] \arrow[l, bend left, "\#/R"] \\
& h &
\end{tikzcd}$$

[1]:This row is to conform to the formal definition of a function

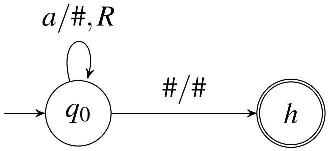
Example 1

A machine that erases all the symbols from left to right.

$S = \{q_0, q_1\}$
$\Sigma = \{a, \#\}$
$s_0 = q_0$

Now the same machine with a different notion and fewer lines

q	σ	$\delta(q,\sigma), HM$
q_0	a	$(q_0, \#), 1$
q_0	$\#$	$(h, \#), 0$

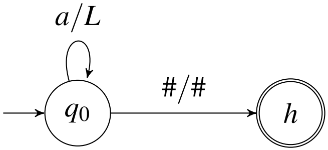

$$\begin{tikzcd} \arrow[r] & q_0 \arrow[loop above, "a/\#,R"] \arrow[r, "\#/\#"] & h \end{tikzcd}$$

Example 2

A machine that moves towards left and recognizes a symbols, halting with the first # symbol

$S = \{q_0\}$
$\Sigma = \{a, \#\}$
$s_0 = q_0$

q	σ	$\delta(q,\sigma)$
q_0	a	(q_0, L)
q_0	$\#$	$(h, \#)$



Configuration of a TM

A configuration is an element of the following set:
$S \cup \{h\} \times \Sigma^* \times \Sigma \times (\Sigma^*(\Sigma - \{\#\}) \cup \{\Lambda\})$

For instance(assume the head is on the underlined symbol):
(q, aba, a, bab) or $(q, aba\underline{a}bab)$
$(h, \#\#, \#, \#a)$ or $(h, \#\#\underline{\#}\#a)$
(q, Λ, a, aba) or $(q, \Lambda\underline{a}aba)$ or $(q, \underline{a}aba)$
$(q, \#a\#, \#, \Lambda)$ or $(q, \#a\#\underline{\#}\Lambda)$ or $(q, \#a\#\underline{\#})$

Following situation doesn't conform with configuration definition:
$(q, baa, a, bc\#)$ or $(q, baa\underline{a}bc\#)$

Configuration of a TM

Yields in one step for TM

$(q_1, \omega_1, a_1, u_1) \vdash_M (q_2, \omega_2, a_2, u_2)$
Here $\delta(q_1, a_1) = (q_2, b)$ and $b \in \Sigma \cup \{L, R\}$

1) $b \in \Sigma, \omega_2 = \omega_1, u_2 = u_1, a_2 = b$ (a_2 is written over a_1)
2) $b = L, \omega_1 = \omega_2 a_2; (a_1 = \# \wedge u_1 = \Lambda \Rightarrow u_2 = \Lambda) \vee (a_1 \neq \# \vee u_1 \neq \Lambda \Rightarrow u_2 = a_1 u_1)$. Left movement: If the head is on a blank and no more symbols to the right blank is erased. If the head is on a symbol or some symbols exist on the right the situation is preserved.
3) $b = R, \omega_2 = \omega_1 a_1; (u_1 = a_2 u_2) \vee (u_1 = \Lambda \Rightarrow u_2 = \Lambda \wedge a_2 = \#)$. Right movement: If there are no more symbols on the right blank is written.

Example 3

$\omega, u \in \Sigma^*; a, b \in \Sigma$ (u cannot end with #)

$$\delta(q_1, a) = (q_2, b) \Rightarrow (q_1, \omega\underline{a}u) \vdash_M (q_2, \omega\underline{b}u)$$

$$\begin{aligned}\delta(q_1, a) = (q_2, L) \Rightarrow\ & i)(q_1, \omega b\underline{a}u) \vdash_M (q_2, \omega\underline{b}au)\\ & ii)(q_1, \omega b\underline{\#}) \vdash_M (q_2, \omega\underline{b})\end{aligned}$$

$$\begin{aligned}\delta(q_1, a) = (q_2, R) \Rightarrow\ & i)(q_1, \omega\underline{a}bu) \vdash_M (q_2, \omega a\underline{b}u)\\ & ii)(q_1, \omega\underline{a}) \vdash_M (q_2, \omega a\underline{\#})\end{aligned}$$

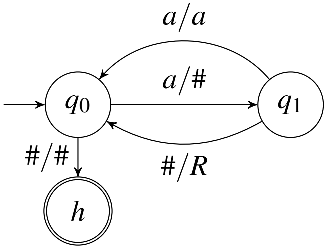
n steps computation

A computation of length n

A computation of length n or in other words n steps computation can be defined as:
$C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \dots C_{n-1} \vdash_M C_n$

Example 1: A machine that erases all the symbols from left to right.

q	σ	$\delta(q,\sigma)$
q_0	a	$(q_1,\#)$
q_0	$\#$	$(h,\#)$
q_1	a	(q_0,a)
q_1	$\#$	(q_0,R)


$$\begin{tikzcd}
\arrow[r] & q_0 \arrow[r, "a/\#"] \arrow[d, "\#/\#"'] & q_1 \arrow[l, bend right=50, "a/a"'] \arrow[l, bend left=40, "\#/R"] \\
& h &
\end{tikzcd}$$

$(q_0,\underline{a}aaa) \vdash_M (q_1,\underline{\#}aaa) \vdash_M (q_0,\#\underline{a}aa) \vdash_M (q_1,\#\underline{\#}aa) \vdash_M (q_0,\#\#\underline{a}a) \vdash_M$
$(q_1,\#\#\underline{\#}a) \vdash_M (q_0,\#\#\#\underline{a}) \vdash_M (q_1,\#\#\#\underline{\#}) \vdash_M (q_0,\#\#\#\#\underline{\#}) \vdash_M (h,\#\#\#\#\underline{\#})$

Turing Computable Function

Turing Computable Function

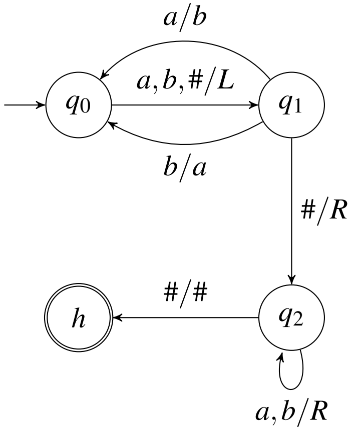
Turing computable functions can perform transformations over character strings.
$M = (S, \Sigma, \delta, s_0)$
Let's distinguish Σ_I as input alphabet and Σ_O as output alphabet;
$f(\omega) = u \wedge \omega \in {\Sigma_I}^* \wedge u \in {\Sigma_O}^* \subseteq \Sigma^*$ and
$(s_0, \#\omega\underline{\#}) \vdash_M^* (h, \#u\underline{\#}), \# \notin \Sigma_I \wedge \# \notin \Sigma_O$

Example 4

A machine that inverses strings (writes b in place of a ; a in place of b)

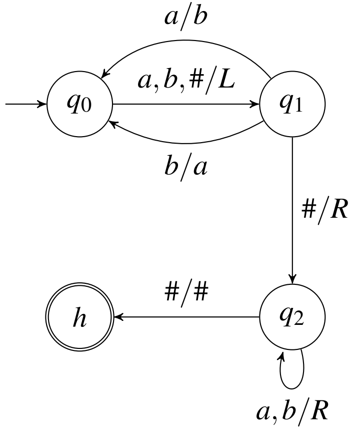
$S = \{q_0, q_1, q_2\}$
$\Sigma = \{a, b, \#\}$
$s_0 = q_0$

q	σ	$\delta(q,\sigma)$
q_0	a	(q_1, L)
q_0	b	(q_1, L)
q_0	$\#$	(q_1, L)
q_1	a	(q_0, b)
q_1	b	(q_0, a)
q_1	$\#$	(q_2, R)
q_2	a	(q_2, R)
q_2	b	(q_2, R)
q_2	$\#$	$(h, \#)$



Example 4

A machine that inverses strings (writes b in place of a ; a in place of b)



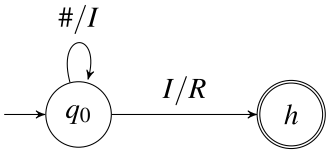
$$
\begin{aligned}
&(q_0,\#aab\underline{\#}) \vdash_M (q_1,\#aa\underline{b}) \vdash_M\\
&(q_0,\#aa\underline{a}) \vdash_M (q_1,\#a\underline{a}a) \vdash_M\\
&(q_0,\#a\underline{b}a) \vdash_M (q_1,\#\underline{a}ba) \vdash_M\\
&(q_0,\#\underline{b}ba) \vdash_M (q_1,\underline{\#}bba) \vdash_M\\
&(q_2,\#\underline{b}ba) \vdash_M (q_2,\#b\underline{b}a) \vdash_M\\
&(q_2,\#bb\underline{a}) \vdash_M (q_2,\#bba\underline{\#}) \vdash_M\\
&(h,\#bba\underline{\#})
\end{aligned}
$$

Example 5

Let's represent a number n with n I symbols like $III\dots I$. Let's build a machine that computes $f(n) = n+1$

$S = \{q_0\}$
$\Sigma = \{I, \#\}$
$s_0 = q_0$

q	σ	$\delta(q,\sigma)$
q_0	I	(h, R)
q_0	#	(q_0, I)



a) $(q_0, \#II\underline{\#}) \vdash_M (q_0, \#II\underline{I}) \vdash_M (h, \#III\underline{\#})$

b) $(q_0, \#I^n\underline{\#}) \vdash_M (h, \#I^{n+1}\underline{\#})$

c) $(q_0, \#\#) \vdash_M (q_0, \#\underline{I}) \vdash_M (h, \#I\underline{\#})$

Some examples

1) $\#\omega\underline{\#} \Rightarrow \#\omega\#\omega\underline{\#}$
2) $\#\omega\underline{\#} \Rightarrow \#\omega\omega^R\underline{\#}$
3) $\#\omega\omega^R\underline{\#} \Rightarrow \#\textcircled{Y}\underline{\#}$

Let's make a demonstration run for each of these machines and try to construct a transition table.

Some examples

1)$\#\omega\underline{\#} \Rightarrow \#\omega\#\omega\underline{\#}$

Example run:
$\#aba\underline{\#} \rightarrow \#aba\underline{\gamma} \rightarrow \#ab\underline{a}\gamma \rightarrow \ldots \rightarrow \underline{\#}aba\gamma \rightarrow \#\underline{a}ba\gamma \rightarrow \#\underline{\alpha}ba\gamma \rightarrow \ldots \rightarrow$
$\#\alpha ba\gamma\underline{\#} \rightarrow \#\alpha ba\gamma\underline{a} \rightarrow \ldots \rightarrow$
$\#\underline{\alpha}ba\gamma a \rightarrow \#\underline{a}ba\gamma a \rightarrow \#a\underline{b}a\gamma a \rightarrow \#a\underline{\beta}a\gamma a \rightarrow \ldots \rightarrow$
$\#a\beta a\gamma a\underline{\#} \rightarrow \#a\beta a\gamma a\underline{b} \rightarrow \ldots \rightarrow$
$\#a\underline{\beta}a\gamma ab \rightarrow \#a\underline{b}a\gamma ab \rightarrow \#ab\underline{a}\gamma ab \rightarrow \#ab\underline{\alpha}\gamma ab \rightarrow \ldots \rightarrow$
$\#ab\alpha\gamma ab\underline{\#} \rightarrow \#ab\alpha\gamma ab\underline{a} \rightarrow \ldots \rightarrow$
$\#ab\underline{\alpha}\gamma aba \rightarrow \#ab\underline{a}\gamma aba \rightarrow \#aba\underline{\gamma}aba \rightarrow \#aba\underline{\#}aba \rightarrow \ldots \rightarrow \#aba\#aba\underline{\#}$

Let’s consider a special situation:
Example run:
$\#\underline{\#} \rightarrow \#\underline{\gamma} \rightarrow \underline{\#}\gamma \rightarrow \#\underline{\#} \rightarrow \#\#\underline{\#}$

Some examples

2)$\#\omega\underline{\#} \Rightarrow \#\omega\omega^R\underline{\#}$

Example run:

$\#abb\underline{\#} \rightarrow \#ab\underline{b} \rightarrow \#ab\underline{\beta} \rightarrow \#ab\beta\underline{\#} \rightarrow \#ab\beta\underline{b} \rightarrow \#ab\underline{\beta}b \rightarrow \#ab\underline{b}b \rightarrow \#a\underline{b}bb \rightarrow$
$\#a\underline{\beta}bb \rightarrow \#a\beta\underline{b}b \rightarrow \#a\beta b\underline{b} \rightarrow \#a\beta bb\underline{\#} \rightarrow \#a\beta bb\underline{b} \rightarrow \ldots \rightarrow$
$\#a\underline{\beta}bbb \rightarrow \#a\underline{b}bbb \rightarrow \#\underline{a}bbbb \rightarrow$
$\#\underline{\alpha}bbbb \rightarrow \#\alpha bbb\underline{b} \rightarrow \#\alpha bbbb\underline{\#} \rightarrow \#\alpha bbbb\underline{a} \rightarrow \#\alpha bbb\underline{b}a \rightarrow \ldots \rightarrow$
$\#\underline{\alpha}bbbba \rightarrow \#\underline{a}bbbba \rightarrow \underline{\#}abbbba \rightarrow$
$\#\underline{a}bbbba \rightarrow \ldots \rightarrow \#abbbb\underline{a} \rightarrow \#abb|bba\underline{\#}$

Some examples

3)$\#\omega\omega^R\underline{\#} \Rightarrow \#\textcircled{Y}\underline{\#}$

Example run:

$\#abbbba\underline{\#} \rightarrow \#abbbb\underline{a}\# \rightarrow \#abbb\underline{b} \rightarrow \ldots \rightarrow$
$\underline{\#}abbbb \rightarrow \#\underline{a}bbbb \rightarrow \#\underline{\theta}bbbb \rightarrow \ldots \rightarrow$
$\#\theta bbbb\underline{\#} \rightarrow \#\theta bbb\underline{b} \rightarrow \#\theta bb\underline{b} \rightarrow \ldots \rightarrow$
$\#\underline{\theta}bbb \rightarrow \#\theta\underline{b}bb \rightarrow \#\theta\underline{\theta}bb \rightarrow \ldots \rightarrow$
$\#\theta\theta b\underline{b} \rightarrow \#\theta\theta bb\underline{\#} \rightarrow \#\theta\theta b\underline{b} \rightarrow \#\theta\theta\underline{b} \rightarrow \#\theta\underline{\theta}b \rightarrow \#\theta\theta\underline{b} \rightarrow \#\theta\theta\underline{\theta}$[11] $\rightarrow$
$\#\theta\theta\underline{\theta} \rightarrow \#\theta\underline{\theta} \rightarrow \#\underline{\theta} \rightarrow \underline{\#} \rightarrow \#\underline{\textcircled{Y}} \rightarrow \#\textcircled{Y}\underline{\#}$

[11] Point where machine decides to output YES

Some examples

3)$\#\omega\omega^R\underline{\#} \Rightarrow \#\text{Ⓨ}\underline{\#}$

Example run:
$\#\theta\theta bb\underline{b} \rightarrow \#\theta\underline{\theta}bb \rightarrow \#\theta\theta\theta b\underline{\#} \rightarrow \#\theta\theta\theta\underline{b} \rightarrow \#\theta\theta\underline{\theta} \rightarrow \#\theta\theta\theta\underline{\#}$[12] $\rightarrow \#\theta\theta\theta\# \rightarrow$
$\#\theta\theta\underline{\theta} \rightarrow \ldots \rightarrow \#\underline{\theta} \rightarrow \underline{\#} \rightarrow \#\text{Ⓝ}\underline{\#}$

[12] Point where machine decides to output NO

Some examples

3)$\#\omega\omega^R\underline{\#} \Rightarrow \#\textcircled{Y}\underline{\#}$

Example run:
$\#ab\underline{\#} \rightarrow \#a\underline{b} \rightarrow \#\underline{a} \rightarrow \underline{\#}a \rightarrow \#\underline{a}$[13] $\rightarrow \#a\underline{\#} \rightarrow \#\underline{a} \rightarrow \underline{\#} \rightarrow \#\underline{\textcircled{N}} \rightarrow \#\textcircled{N}\underline{\#}$

[13] Point where machine decides to output NO

Some examples

3)$\#\omega\omega^R\underline{\#} \Rightarrow \#\textcircled{Y}\underline{\#}$

Example run:
$\#aaab\underline{\#} \rightarrow \#aaa\underline{b} \rightarrow \#aa\underline{a} \rightarrow \ldots \rightarrow$
$\underline{\#}aaa \rightarrow \#\underline{a}aa$[14] $\rightarrow \#aaa\underline{\#} \rightarrow \#aa\underline{a} \rightarrow \#a\underline{a} \rightarrow \#\underline{a} \rightarrow \underline{\#} \rightarrow \#\underline{\textcircled{N}} \rightarrow \#\textcircled{N}\underline{\#}$

[14] Point where machine decides to output NO

Section 19

Turing Decidability

Turing Decidable Machine

Turing Recognazibility vs. Turing Decidability

- Turing recognizability is the ability to design a Turing machine that can halt if a string belongs to a language.
- Turing recognizability do not guarantee halting for strings outside the Turing recognizable language.
- A stronger property is Turing decidability where it is possible to design a machine that can halt for the strings both inside and outside the language.

Turing Decidable Machine

Turing Decidable Machine

Let Σ_0 be our alphabet and # $\notin \Sigma_0$ and L be a language $L \subseteq {\Sigma_0}^*$
If we can compute the function x_L like:

$$\forall \omega \in \Sigma_0^*, F_L(\omega) = \begin{cases} \textcircled{Y} \Rightarrow \omega \in L \\ \textcircled{N} \Rightarrow \omega \notin L \end{cases}$$

Turing Decidable Machine

Example:

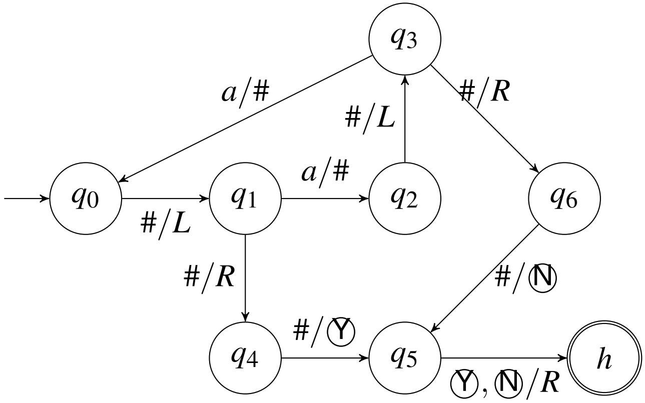
$\Sigma_0 = \{a\}; L = \{\omega \in \Sigma_0{}^* | \ ||\omega||$even number$\}$

$S = \{q_0, \ldots, q_6\}$

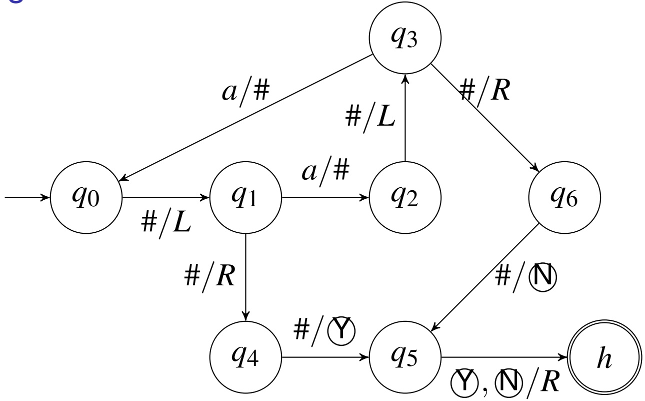
$\Sigma = \{a, Ⓨ, Ⓝ, \#\}$

$s_0 = q_0$

q	σ	$\delta(q,\sigma)$
q_0	#	(q_1, L)
q_1	a	$(q_2, \#)$
q_1	#	(q_4, R)
q_2	#	(q_3, L)
q_3	a	$(q_0, \#)$
q_3	#	(q_6, R)
q_4	#	$(q_5$, Ⓨ$)$
q_5	Ⓨ	(h, R)
q_5	Ⓝ	(h, R)
q_6	#	$(q_5$, Ⓝ$)$

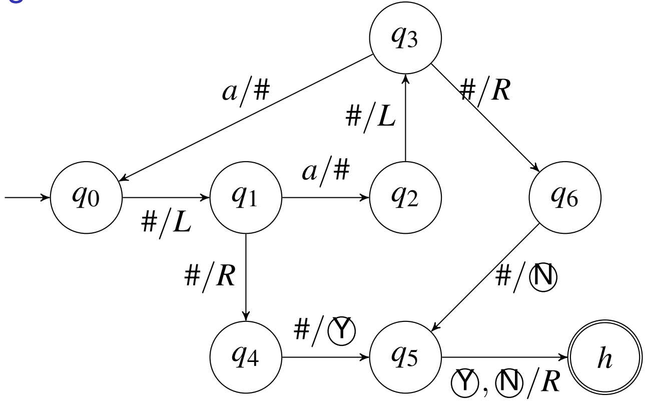


Turing Decidable Machine


q_3
$a/\#$
$\#/L$
$\#/R$
q_0
q_1
$a/\#$
q_2
q_6
$\#/L$
$\#/R$
$\#/\text{\textcircled{N}}$
$\#/\text{\textcircled{Y}}$
q_4
q_5
h
$\text{\textcircled{Y}},\text{\textcircled{N}}/R$

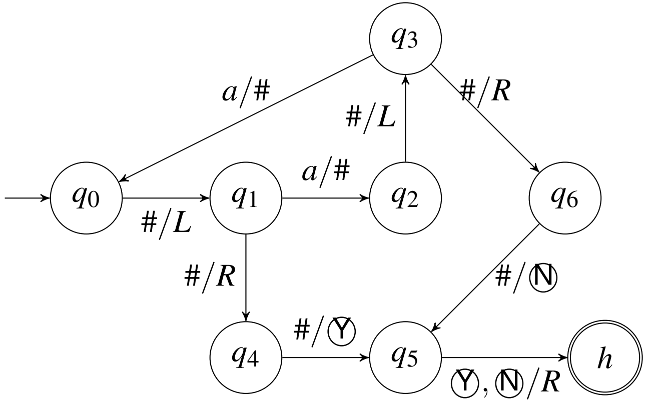
a) $(q_0,\#aa\underline{\#}) \vdash_M (q_1,\#a\underline{a}) \vdash_M (q_2,\#a\underline{\#}) \vdash_M$
$(q_3,\#\underline{a}) \vdash_M (q_0,\#\underline{\#}) \vdash_M (q_1,\underline{\#}) \vdash_M$
$(q_4,\#\underline{\#}) \vdash_M (q_5,\#\underline{\text{\textcircled{Y}}}) \vdash_M (h,\#\text{\textcircled{Y}}\underline{\#})$

Turing Decidable Machine



b) $(q_0,\#aaa\underline{\#}) \vdash_M (q_1,\#aa\underline{a}) \vdash_M (q_2,\#aa\underline{\#}) \vdash_M$
$(q_3,\#a\underline{a}) \vdash_M (q_0,\#a\underline{\#}) \vdash_M (q_1,\#\underline{a}) \vdash_M$
$(q_2,\#\underline{\#}) \vdash_M (q_3,\underline{\#}) \vdash_M (q_6,\#\underline{\#}) \vdash_M$
$(q_5,\#\underline{\textcircled{N}}) \vdash_M (h,\#\textcircled{N}\underline{\#})$

Turing Decidable Machine



a) $(q_0, \#a^n\underline{\#}) \vdash_M^* (h, \#\textcircled{Y}\underline{\#}) \Rightarrow n$ is even

b) $(q_0, \#a^n\underline{\#}) \vdash_M^* (h, \#\textcircled{N}\underline{\#}) \Rightarrow n$ is odd

When we start a Turing machine on an input it may **accept**, **reject**, or **loop**. Looping may entail any simple or complex behavior that never leads to a halting state.

Definition

A language is Turing-recognizable if some Turing machine is able **accept** the words in a language.

Definition

A language is Turing-decidable or simply decidable if some Turing machine is able to decide it. Turing machines that halt on all inputs and never loop are called deciders because they always make a decision to **accept** or **reject**.

Corollary

Every decidable language is Turing-recognizable.

Definition (The Acceptance Problem)

Testing whether a particular deterministic finite automaton accepts a given string can be expressed as a language, A_{DFA}. This language contains the encodings of all DFAs together with strings that the DFAs accept.

$$A_{DFA} = \{\langle B, \omega\rangle | B \text{ is a } DFA \text{ that accepts input string } \omega\}$$

Theorem

A_{DFA} *is a decidable language*

Proof.

M = On input $\langle B, \omega \rangle$, where B is a DFA and ω is a string:

1. Simulate B on input ω
2. If the simulation ends in an accept state, accept . If it ends in a nonaccepting state, reject.

$\square$

$$A_{NFA} = \{\langle B, \omega \rangle | B \text{ is a } NFA \text{ that accepts input string } \omega\}$$

Theorem

A_{NFA} is a decidable language

Proof.

N = On input $\langle B, \omega \rangle$, where B is a NFA and ω is a string:

1. Convert NFA B to an equivalent DFA C
2. Run TM M from previous slide on input $C, \omega \rangle$
3. If M accepts, accept ; otherwise, reject.

□

$$A_{Rex} = \{\langle R, \omega\rangle | R \text{ is a regular expression that generates string } \omega\}$$

Theorem

A_{Rex} is a decidable language

Proof.

P = On input $\langle R, \omega\rangle$, where R is a regular expression and ω is a string:

1. Convert regular expression R to an equivalent NFA A
2. Run TM N from previous slide on input $\langle A, \omega\rangle$
3. If N accepts, accept ; otherwise, reject.

□

Definition (Emptiness Testing)

Testing whether or not a finite automaton accepts any strings at all.

$E_{DFA} = \{\langle A \rangle | A$ is a DFA and L(A)=$\emptyset$ $\}$

Theorem

E_{DFA} is a decidable language

Proof.

T = On input $\langle A \rangle$, where A is a DFA:

1. Mark the start state of A
2. Repeat until no new states get marked:
    1. Mark any state that has a transition coming into it from any state that is already marked.
3. If no accept state is marked, accept; otherwise, reject.

$\square$

Definition (Automaton Equality)

Testing whether two DFAs recognize the same language is decidable.

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

Definition (Symmetric Difference of two Sets)

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

$L(C) = \emptyset$ iff $L(A) = L(B)$

Theorem

EQ_{DFA} *is a decidable language*

Proof.

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

1. Construct DFA C as described
2. Run TM T from previous slides on input $\langle C \rangle$
3. If T accepts, accept. If T rejects, reject.

$\square$

$$A_{CFG} = \{\langle G, \omega\rangle | G \text{ is a } CFG \text{ that generates string } \omega\}$$

Theorem

A_{CFG} is a decidable language

Proof.

F = On input $\langle G, \omega\rangle$, where G is a CFG and ω is a string:

1. Convert G to an equivalent grammar in Chomsky normal form.
2. List all derivations with $2n1$ steps, where n is the length of w; except if $n = 0$, then instead list all derivations with one step.
3. If any of these derivations generate w, accept; if not, reject.

□

$$E_{CFG} = \{\langle G, \omega\rangle | G \text{ is a } CFG \text{ and } L(G) = \emptyset\}$$

Theorem

E_{CFG} is a decidable language

Proof.

R = On input $\langle G\rangle$, where G is a CFG:

1. Mark all terminal symbols in G.
2. Repeat until no new variables get marked:
    1. Mark any variable A where G has a rule $A \rightarrow U_1 U_2 \dots U_k$ and each symbol $U_1, \dots, U_k$ has already been marked.
3. If the start variable is not marked, accept; otherwise, reject.

□

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

Theorem

A_{TM} is undecidable.

A_{TM} is Turing-recognizable. Recognizers are more powerful than deciders. Requiring a TM to halt on all inputs restricts the kinds of languages that it can recognize.

Proof Trial

On input $\langle M\rangle, \omega$, where M is a TM and ω is a string:

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

The Halting Problem

The Halting Problem

Given a TM M and a string ω, does M halt on input ω?

Theorem

$HALT_{TM}$ is undecidable.

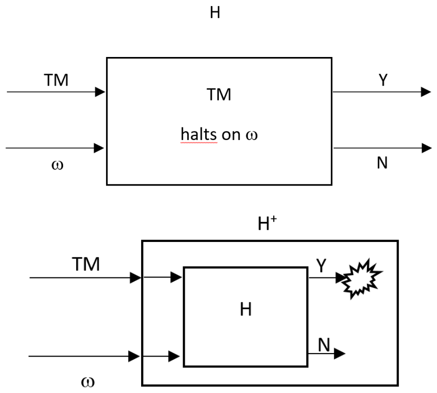
The Halting Problem

Proof.

1. Assume that TM H can decide the halting problem.
2. We construct TM H^+ that accepts the same input set as H. H^+ contains a copy of H inside and works as follows
    1. Run TM H on input $\langle M, \omega \rangle$
    2. If H rejects, accept
    3. If H accepts, loop forever
3. Feed H^+ to H^+. What should the answer be?

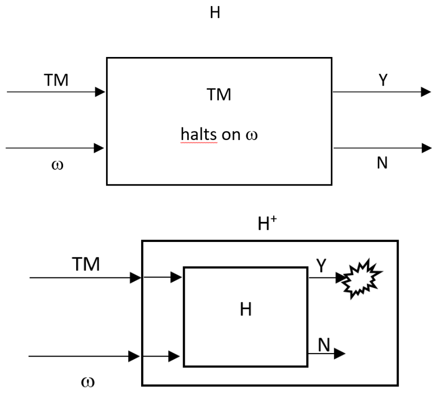
□

The Halting Problem



- H$^+$ says "no" which means H$^+$ will not stop, but it just did.
- H$^+$ doesn't stop which means H decided that it can stop but it just didn't.

The Halting Problem



- H^+ says "no" which means H^+ will not stop, but it just did.
- H^+ doesn't stop which means H decided that it can stop but it just didn't.

The Halting Problem

```
n = 4
while (n is the sum of two primes)
  n = n+2
```

- Goldbach's conjecture[15]: every even integer greater than 2 is the sum of two primes
- Can you write a program to decide if this program continues forever or not?

[15] Hasn't been proven yet

The Halting Problem

```
def check_halting(function, parameter_set):
    """
      — Return True if function(parameter_set)
        terminates
      — Return False if function(parameter_set)
        runs forever.
    """
    #rest of the check_halting

def tricky(func_call):
    if check_halting(func_call.func, func_call.parameters):
        while True:
            pass
    else:
        return
```

What if we call `tricky(tricky)`

Undecidability Examples

Theorem

$E_{TM}\{\langle M\rangle \mid M$ is a TM and $L(M)=\emptyset\}$ is undecidable.

Theorem

$REGULAR_{TM}\{\langle M\rangle \mid M$ is a TM and $L(M)$ is a regular language$\}$ is undecidable.

Theorem

$EQ_{TM}\{\langle M_1,M_2\rangle \mid M_1$ and M_2 are TMs and $L(M_1)=L(M_2)\}$ is undecidable.

Theorem

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

Theorem

$EQ_{CFG} = \{\langle G_1, G_2 \rangle \mid G_1$ and G_2 are $CFGs$ and $L(G_1) = L(G_2)\}$ is undecidable.

Theorem

*$INTERSECT_{CFG} =$
$\{\langle G_1, G_2 \rangle \mid G_1$ and G_2 are $CFGs$ and $L(G_1) \cap L(G_2) \neq \emptyset\}$ is undecidable.*

Church-Turing Thesis

- Church-Turing thesis states that a function is algorithmically computable if and only if it is computable by a Turing machine.
- If some method (algorithm) exists to carry out a calculation, then the same calculation can also be carried out by a Turing machine (as well as by a recursively definable function, and by a λ-function).
- The three processes mentioned above proved to be equivalent
- The notion of what it means for a function to be effectively calculable remains a conjecture.

Example I

$$M \text{ decides } A = \{0^{2^n} | n \geq 0\}$$

For string w

1. Sweep left to right across the tape, crossing off every other 0.
2. If in stage 1 the tape contained a single 0, accept.
3. If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, reject.
4. Return the head to the left-hand end of the tape.
5. Go to step 1.

Example - DIY

$$M \text{ decides } A = \{\omega\#\omega | \omega \in \{0,1\}^*\}$$

Example II

$$M \text{ decides } A = \{a^i b^j c^k | i \times j = k \text{ and } i, j, k \geq 1\}$$

For string w

1. Scan the input from left to right to determine whether it is a member of $a^+b^+c^+$ and reject if it isn't.
2. Return the head to the left-hand end of the tape.
3. Cross off an a and scan to the right until a b occurs. Shuttle between the b's and the c's, crossing off one of each until all b's are gone. If all c's have been crossed off and some b's remain, reject.
4. Restore the crossed off b's and repeat stage 3 if there is another a to cross off. If all a's have been crossed off, determine whether all c's also have been crossed off. If yes, accept; otherwise, reject.

Example III

$$M \text{ decides}$$
$$A = \{\#x_1\#x_2\#\ldots\#x_\ell \mid \text{each } x_i \in \{0,1\}^* \text{ and } x_i \neq x_j \text{ for each } i \neq j\}$$

For string w

1. Place a mark on top of the leftmost tape symbol. If that symbol was a blank, accept. If that symbol was a #, continue with the next step. Otherwise, reject.
2. Scan right to the next # and place a second mark on top of it. If no # is encountered before a blank symbol, only x_1 was present, so accept
3. By zig-zagging, compare the two strings to the right of the marked #s. If they are equal, reject.
4. Move the rightmost of the two marks to the next # symbol to the right. If no # symbol is encountered before a blank symbol, move the leftmost mark to the next # to its right and the rightmost mark to the # after that. This time, if no # is available for the rightmost mark, all the strings have been compared, so accept.
5. Go to step 3.

Example IV

Hilbert's Tenth Problem

Devise a process according to which it can be determined by a finite number of operations if a multivariable polynomial has integral(integer) roots.

For now, let's consider a polynomial over a single variable and define the machine M as
On input p: where p is a polynomial over the variable x.

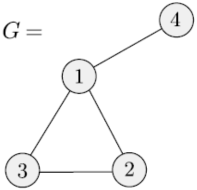
1. Evaluate p with x set successively to the values 0, 1, -1, 2, -2, 3, -3,. . . If at any point the polynomial evaluates to 0, accept.

M is a recognizer, but not a decider. We can make M a decider by defining a range that the roots can reside in.

For a single variable polynomial this range is inside $\pm k\frac{c_{\text{max}}}{c_1}$, however for a multivariable polynomial it is not possilbe to calculate such a range.

Example V - Graph Connectivity

$G =$



$\langle G \rangle =$

```
(1,2,3,4)((1,2),(2,3),(3,1),(1,4))
```

M decides $A = \{\langle G \rangle | G$ is a connected undirected graph$\}$

On input $\langle G \rangle$, the encoding of graph G:

1. Select the first node of G and mark it.
2. Repeat the following step until no new nodes are marked:
3. For each node in G, mark it if it is attached by an edge to a node that is already marked.
4. Scan all the nodes of G to determine whether they all are marked. If they are, accept; otherwise, reject.