**Lecture Notes:**

**UNIT I:**

**Fundamentals**

* **Symbol** – An atomic unit, such as a digit, character, lower-case letter, etc. Sometimes a word. *[Formal language does not deal with the “meaning” of the symbols.]*
* **Alphabet** – A finite set of symbols, usually denoted by Σ.

Σ = {0, 1} Σ = {0, a, 9, 4} Σ = {a, b, c, d}

* **String** – A finite length sequence of symbols, presumably from some alphabet. w = 0110 y = 0aa x = aabcaa z = 111

Special string: ε (also denoted by λ) Concatenation: wz = 0110111

Length: |w| = 4 |ε| = 0 |x| = 6

Reversal: yR = aa0

* Some special sets of strings:

Σ\* All strings of symbols from Σ Σ+ Σ\* - {ε}

* Example: Σ = {0, 1}

Σ\* = {ε, 0, 1, 00, 01, 10, 11, 000, 001,…}

Σ+ = {0, 1, 00, 01, 10, 11, 000, 001,…}

* A **language** is:

1. A set of strings from some alphabet (finite or infinite). In other words,
2. Any subset L of Σ\*

* Some special languages:

{} The empty set/language, containing no string.

{ε} A language containing one string, the empty string.

* Examples:

Σ = {0, 1}

L = {x | x is in Σ\* and x contains an even number of 0’s}

Σ = {0, 1, 2,…, 9, .}

L = {x | x is in Σ\* and x forms a finite length real number}

= {0, 1.5, 9.326,…}

Σ = {a, b, c,…, z, A, B,…, Z}

L = {x | x is in Σ\* and x is a Pascal reserved word}

= {BEGIN, END, IF,…}

Σ = {Pascal reserved words} U { (, ), ., :, ;,…} U {Legal Pascal identifiers} L = {x | x is in Σ\* and x is a syntactically correct Pascal program}

Σ = {English words}

L = {x | x is in Σ\* and x is a syntactically correct English sentence}

Finite State Machines

* A finite state machine has a set of states and two functions called the next-state function and the output function
  + The set of states correspond to all the possible combinations of the internal storage
    - If there are n bits of storage, there are 2n possible states
  + The next state function is a combinational logic function that given the inputs and the current state, determines the next state of the system
* The output function produces a set of outputs from the current state and the inputs
  + There are two types of finite state machines
  + In a Moore machine, the output only depends on the current state
  + While in a Mealy machine, the output depends both the current state and the current input
  + We are only going to deal with the Moore machine.

– These two types are equivalent in capabilities

* A Finite State Machine consists of:

K states: S = {s1, s2, … ,sk}, s1 is initial state N inputs: I = {i1, i2, … ,in}

M outputs: O = {o1, o2, … ,om}

Next-state function T(S, I) mapping each current state and input to next state Output Function P(S) specifies output

Finite Automata

* Two types – both describe what are called regular languages
* Deterministic (DFA) – There is a fixed number of states and we can only be in one state at a time
* Nondeterministic (NFA) –There is a fixed number of states but we can be in multiple states at one time
* While NFA’s are more expressive than DFA’s, we will see that adding nondeterminism does not let us define any language that cannot be defined by a DFA.
* One way to think of this is we might write a program using a NFA, but then when it is “compiled” we turn the NFA into an equivalent DFA.

Formal Definition of a Finite Automaton

1. Finite set of states, typically Q.
2. Alphabet of input symbols, typically ∑
3. One state is the start/initial state, typically q0 // q0 ∈ Q
4. Zero or more final/accepting states; the set is typically F. // F ⊆Q
5. A transition function, typically δ. This function
   * Takes a state and input symbol as arguments.

Deterministic Finite Automata (DFA)

* A DFA is a five-tuple: M = (Q, Σ, δ, q0, F)

Q A finite set of states

Σ A finite input alphabet

q0 The initial/starting state, q0 is in Q

F A set of final/accepting states, which is a subset of Q

δ A transition function, which is a total function from Q x Σ to Q

δ: (Q x Σ) –> Q δ is defined for any q in Q and s in Σ, and δ(q,s) = q’ is equal to another state q’ in Q.

Intuitively, δ(q,s) is the state entered by M after reading symbol s while in state q.

* Let M = (Q, Σ, δ, q , F) be a DFA and let w be in Σ\*. Then w is *accepted* by M iff

0

δ(q ,w) = p for some state p in F.

0

* Let M = (Q, Σ, δ, q , F) be a DFA. Then the *language accepted* by M is the set:

0

L(M) = {w | w is in Σ\* and δ(q ,w) is in F}

0

* Another equivalent definition:

L(M) = {w | w is in Σ\* and w is accepted by M}

* Let L be a language. Then L is a *regular language* iff there exists a DFA M such that L = L(M).
* Let M

= (Q , Σ , δ , q , F ) and M = (Q , Σ , δ , p , F ) be DFAs. Then M and M

are

1 1 1 1 0 1 2 2 2 2 0 2 1 2

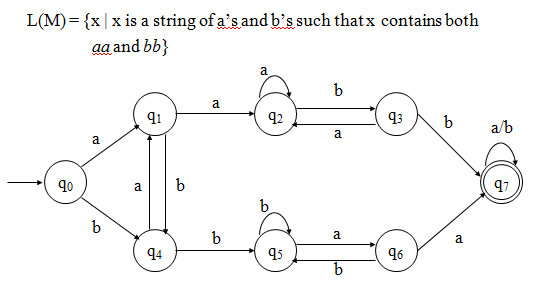
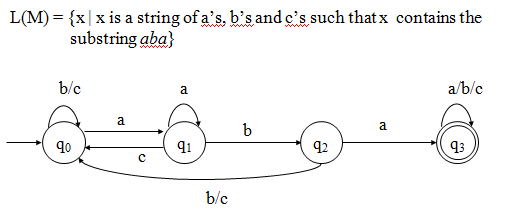
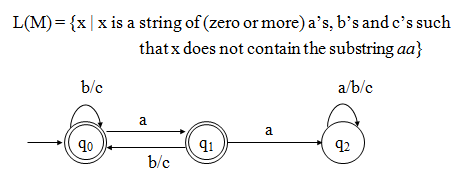
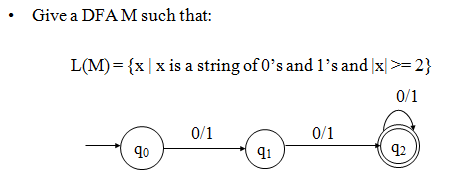
*equivalent* iff L(M ) = L(M ).

1 2

* Notes:
  + A DFA M = (Q, Σ, δ,q0,F) partitions the set Σ\* into two sets: L(M) and Σ\* - L(M).
  + If L = L(M) then L is a subset of L(M) and L(M) is a subset of L.
  + Similarly, if L(M1) = L(M2) then L(M1) is a subset of L(M2) and L(M2) is a subset of L(M1).
  + Some languages are regular, others are not. For example, if

L1 = {x | x is a string of 0's and 1's containing an even number of 1's} and L2 = {x | x = 0n1n for some n >= 0}

then L1 is regular but L2 is not.

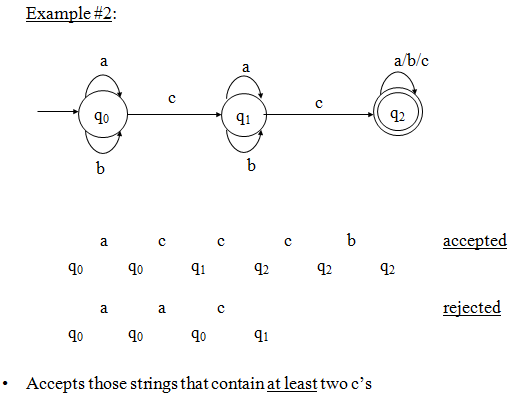
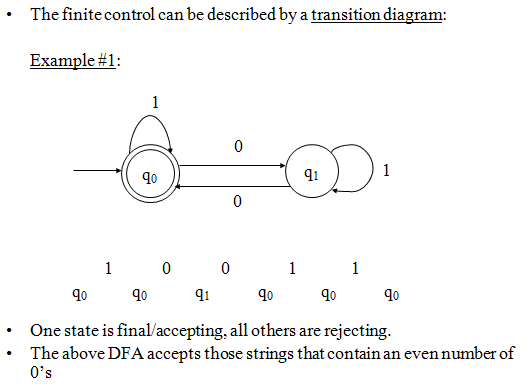


δ: (Q x Σ) -> **2Q** -2Q is the power set of Q, the set of all subsets of Q δ(q,s) -The set of all states p such that there is a transition

labeled s from q to p δ(q,s) is a function from Q x S to 2Q (but not to Q)

* Let M = (Q, Σ, δ,q0,F) be an NFA and let w be in Σ\*. Then w is *accepted* by M iff δ({q0}, w) contains at least one state in F.
* Let M = (Q, Σ, δ,q0,F) be an NFA. Then the *language accepted* by M is the set: L(M) = {w | w is in Σ\* and δ({q0},w) contains at least one state in F}
* Another equivalent definition:

L(M) = {w | w is in Σ\* and w is accepted by M}



NFAs with ε Moves

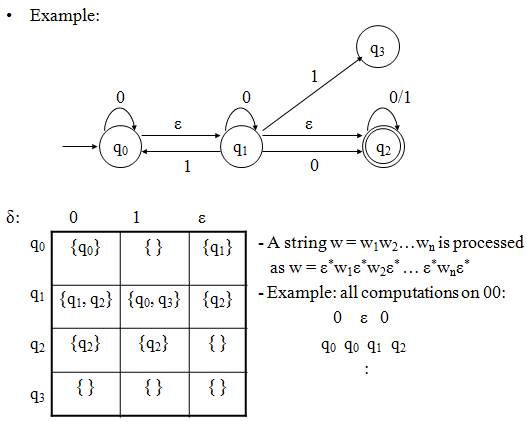
* An NFA-ε is a five-tuple: M = (Q, Σ, δ, q0, F)

Q A finite set of states

Σ A finite input alphabet

q0 The initial/starting state, q0 is in Q

F A set of final/accepting states, which is a subset of Q



δ

A transition function, which is a total function from Q x Σ U {ε} to 2Q

δ: (Q x (Σ U {ε})) –> 2Q

δ(q,s) -The set of all states p such that there is labeled a from q to p, where a is in Σ U {ε}

* Sometimes referred to as an NFA-ε other times, simply as an NFA.

a transition

* Let M = (Q, Σ, δ,q0,F) be an NFA-ε and let w be in Σ\*. Then w is *accepted* by M iff δ^({q0}, w) contains at least one state in F.
* Let M = (Q, Σ, δ,q0,F) be an NFA-ε. Then the *language accepted* by M is the set: L(M) = {w | w is in Σ\* and δ^({q0},w) contains at least one state in F}
* Another equivalent definition:

L(M) = {w | w is in Σ\* and w is accepted by M}

Equivalence of NFA and NFA-ε

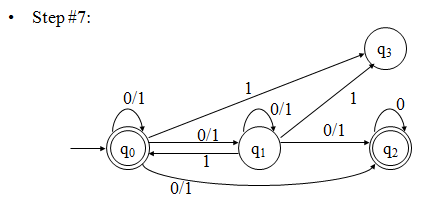
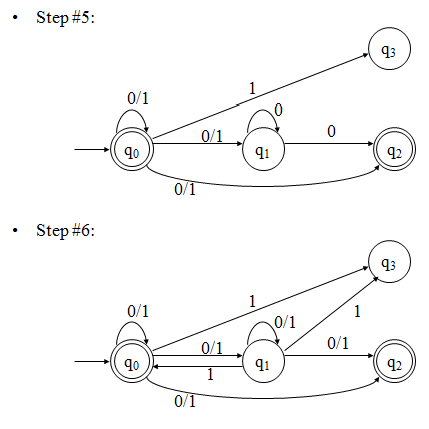
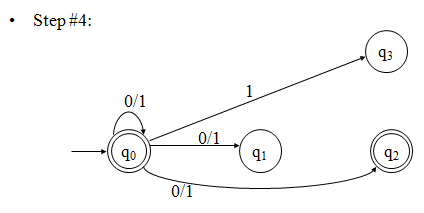
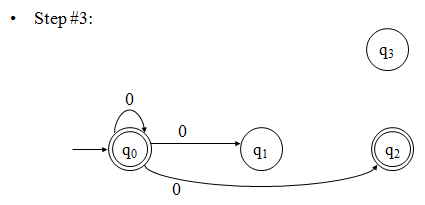
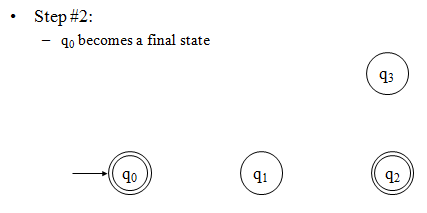
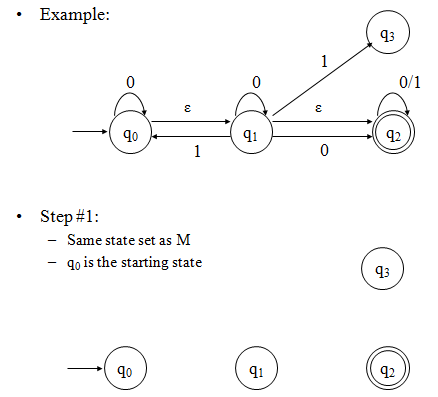
* Do NFAs and NFA-ε machines accept the same *class* of languages?
  + Is there a language L that is accepted by a NFA, but not by any NFA-ε?
  + Is there a language L that is accepted by an NFA-ε, but not by any DFA?
* Observation: Every NFA is an NFA-ε.
* Therefore, if L is a regular language then there exists an NFA-ε M such that L = L(M).
* It follows that NFA-ε machines accept all regular languages.
* But do NFA-ε machines accept more?
* **Lemma 1:** Let M be an NFA. Then there exists a NFA-ε M’ such that L(M) = L(M’).
* **Proof:** Every NFA is an NFA-ε. Hence, if we let M’ = M, then it follows that L(M’) = L(M).
* **Lemma 2:** Let M be an NFA-ε. Then there exists a NFA M’ such that L(M) = L(M’).
* Proof:

Let M = (Q, Σ, δ,q0,F) be an NFA-ε. Define an NFA M’ = (Q, Σ, δ’,q0,F’) as:

F’ = F U {q0} if ε-closure(q0) contains at least one state from F F’ = F otherwise

δ’(q, a) = δ^(q, a) - for all q in Q and a in Σ

* Notes:
  + δ’: (Q x Σ) –> 2Q is a function
  + M’ has the same state set, the same alphabet, and the same start state as M
  + M’ has no ε transitions



* **Theorem:** Let L be a language. Then there exists an NFA M such that L= L(M) iff there exists an NFA-ε M’ such that L = L(M’).
* Proof:

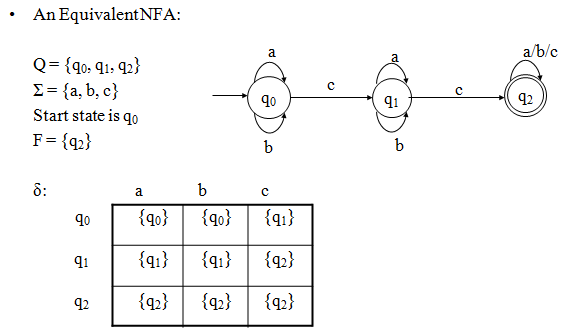
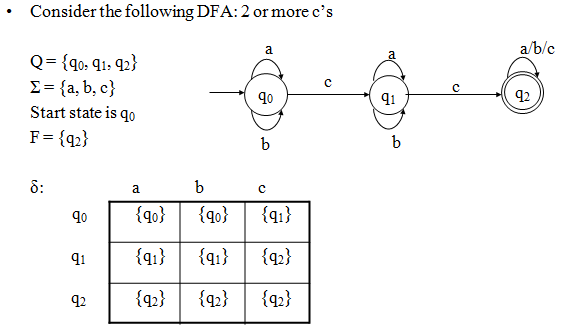
(if) Suppose there exists an NFA-ε M’ such that L = L(M’). Then by Lemma 2 there exists an NFA M such that L = L(M).

(only if) Suppose there exists an NFA M such that L = L(M). Then by Lemma 1 there exists an NFA-ε M’ such that L = L(M’).

* **Corollary:** The NFA-ε machines define the regular languages.

Equivalence of DFAs and NFAs

* Do DFAs and NFAs accept the same *class* of languages?
  + Is there a language L that is accepted by a DFA, but not by any NFA?
  + Is there a language L that is accepted by an NFA, but not by any DFA?
* Observation: Every DFA is an NFA.
* Therefore, if L is a regular language then there exists an NFA M such that L = L(M).
* It follows that NFAs accept all regular languages. But do NFAs accept all?
* **Lemma 1:** Let M be an DFA. Then there exists a NFA M’ such that L(M) = L(M’).



* **Proof:** Every DFA is an NFA. Hence, if we let M’ = M, then it follows that L(M’) = L(M).

The above is just a formal statement of the observation from the above example.

* **Lemma 2:** Let M be an NFA. Then there exists a DFA M’ such that L(M) = L(M’).
* **Proof:** (sketch)

Let M = (Q, Σ, δ,q0,F).

Define a DFA M’ = (Q’, Σ, δ’,q’ ,F’) as:

0

Q’ = 2Q Each state in M’ corresponds to a

= {Q0, Q1,…,} subset of states from M where Qu = [qi0, qi1,…qij]

F’ = {Qu | Qu contains at least one state in F}

* **Theorem:** Let L be a language. Then there exists an DFA M such that L = L(M) iff there exists an NFA M’ such that L = L(M’).
* Proof:

(if) Suppose there exists an NFA M’ such that L = L(M’). Then by Lemma 2 there exists an DFA M such that L = L(M).

(only if) Suppose there exists an DFA M such that L = L(M). Then by Lemma 1 there exists an NFA M’ such that L = L(M’).

**Corollary:** The NFAs define the regular languages.

Finite Automata with Output

* **Acceptor:**

The symbols of the sequence s(1) s(2) … s(i) … s(t)

are presented sequentially to a machine M. M responds with a binary signal to each input. If the string scanned so far is accepted, then the light goes on, else the light is off.

with the

s(t) … s(i) … s(2) s(1)

Input channel

Output signal

Initialize

**A language acceptor**

* **Transducer**

Abstract machines that operate as *transducers* are of interest in connection translation of languages. The following transducer produces a sentence

r(1) r(2) … r(n)

in response to the input sentence s(1) s(2) … s(m)

s(m) … s(j) … s(2) s(1)

r(n) … r(i) … r(2) r(1)

Input channel

Output channel

M

M

Initialize

If this machine is *deterministic*, then each sentence of an *input language* is translated into a specific sentence of an *output language.*

Generator

When M is started from its initial state, it emits a sequence of symbols r(1) r(2) … r(i) … r(t)

from a set known as its *output alphabet*.

Only a finite number of operations may be performed in a finite amount of time. Such systems are necessarily **discrete.**

Problems are quite naturally decomposed into sequences of steps – hence our model is

sequential.

We require that our machine not be subject to uncertainty, hence its behavior is

deterministic.

There are two finite state machine models :

1. **Mealy model** – in which outputs occur during transitions.
2. **Moore model** – outputs are produced upon arrival at a new state.

Mealy Model of FSM

**Mealy model** – transition assigned output, Mt = <Q, S, R, f, g, qI> Where,

Q = finite set of states // the machine’s memory S = input alphabet // set of stimuli

R = output alphabet // set of responses qI = the machine’s initial state

f : state transition function (or next state function) f : Q \* S  Q

g : output function g : Q \* S  R

* Example#1:

Design a FSM (Mealy model) which takes in binary inputs and produces a ‘1’ as output whenever the parity of the input string ( *so far* ) is even.

S = R = {0, 1}

When designing such models, we should ask ourselves *“What is the state set of the machine?”.*

*The state set Q corresponds to what we need to remember about input strings*. We note that the number of possible input strings corresponds to |S\*| which is *countably infinite.*

We observe, however, that a string may have only one of two possible parities.

**even parity** – if n1(w) is even.

**odd parity** – if n1(w) is odd.

*And this is all that our machine must remember about a string scanned so far.*

Hence |Q| = 2 where Q = {E, σ} with qI = E indicating the string has *even parity* and if Mt

*is in state σ*, then the string has *odd parity*.

* + And finally, of course, we must specify *the output function g* for this Mealy machine.
  + According to this machine’s specifications, it is supposed to produce an output of ‘1’ whenever the parity of the input string so far is even. Hence, *all arcs leading into state E should be labeled with a ‘1’ output.*

Parity Checker (Mealy machine)

0/1 0/0

1/0

E σ

1/1

Observe our notation that **g(σ, 1) = 1** is indicated by the arc from state σ to state E with a ‘1’ after a slash.

The output of our machine is 0 when the current string ( *so far* ) has odd parity.

|  |  |  |  |
| --- | --- | --- | --- |
| state table | present state | input = 0  next state, output | input = 1  next state, output |
| for this  parity machine | E | E, 1 | σ, 0 |
|  | σ | σ, 0 | E, 1 |

Observe for the input 10100011 our machine produces the output sequence 00111101

1/0 0/0 1/1 0/1 0/1 0/1 1/0 1/1

E σ σ

E E E E σ E

the corresponding *admissible state sequence*

* Example#2:

Construct a Mealy model of an FSM that behaves as a two-unit delay. i.e. r(t) = {s(t - 2), t > 2

{ 0 , otherwise

*A sample input/output session is given below :*

|  |  |
| --- | --- |
| time | 1 2 3 4 5 6 7 8 9 |
| stimulus | 0 0 0 1 1 0 1 0 0 |
| response | 0 0 0 0 0 1 1 0 1 |

Observe that r(1) = r(2) = 0

r(6) = 1 which equals s(4) and so on We know that S = R = {0, 1}.

Moore model of FSM

**Moore model of FSM** – the output function assigns an output symbol to each state.

Ms = <Q, S, R, f, h, qI>

Q = finite set of internal states S = finite input alphabet

R = finite output alphabet f : state transition function

f : Q \* S  Q h : output function

h : Q → R

qI = Є Q is the initial state

* Example#1:

Design a Moore machine that will analyze input sequences in the binary alphabet S = {0, 1}. Let w = s(1) s(2) … s(t) be an input string

N0(w) = number of 0’s in w N1(w) = number of 1’s in w

then we have that |w| = N0(w) + N1(w) = t.

The last output of Ms should equal : r(t) = [N1(w) – N0(w)] mod 4.

So naturally, *the output alphabet R = {0, 1, 2, 3}*

A sample stimulus/response is given below : stimulus 1 1 0 1 1 1 0 0

response 0 1 2 1 2 3 0 3 2

Observe that the length of the output sequence is one longer than the input sequence. Why is this so?

Btw : This will always be the case.

* The corresponding Moore machine :

B, 1

1

1

0

0

A, 0 C, 2

0

1

0

D, 3 1

State diagram

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 | 1 |  |
| A | D | B | 0 |
| B | A | C | 1 |
| C | B | D | 2 |
| D | C | A | 3 |

State table

This machine is referred to as an *up-down counter.*

For the previous input sequence : 11011100 the *state sequence is :*

1 1 0 1

(A, 0) (B, 1) (C, 2) (B, 1) (C, 2)

1 1 0 0

(D, 3) (A, 0) (D, 3) (C, 2)

* Example#2:

Design a Moore machine that functions as a *pattern recognizer* for “1011”. Your machine should output a ‘1’ whenever this pattern matches the last four inputs, and there has been no overlap, otherwise output a ‘0’.

Hence S = R = {0, 1}.

Here is a sample input/output sequence for this machine : t = 1 2 3 4 5 6 7 8 9 10 11 12

S = 0 1 0 1 1 0 1 1 0 1 1 0

R = 0 0 0 0 1 0 0 0 0 0 0 1 0

We observe that r(5) = 1 because s(2) s(3) s(4) s(5) = 1011 however r(8) = 0 because there has been overlap

r(11) = 1 since s(8) s(9) s(10) s(11) = 1011

Machine Identification Problem

The following input-output behavior was exhibited by a transition-assigned machine (Mealy machine) Mt known to contain three states. Find an appropriate state table for M. Is the table unique?

time 1 2 3 4 5 6 7 8 9 10 11 12 13 14

input 0 0 0 0 1 0 0 0 1 0 0 0 1 0

output 0 1 0 1 0 0 0 0 1 0 1 0 0 1

This problem is useful in fault detection and fault location experiments with sequential circuits ( i.e. *digital circuits with memory* ).

One designs a computer circuit. Six months (or six years) later, how does one know that the circuit is working correctly?

The procedure to solve this problem is helpful in fault diagnosis of digital circuits.

Equivalence of Mealy and Moore Models

The Mealy and Moore models of finite state machines are equivalent ( actually similar ).

i.e. Mt ≈ Ms

What does this mean ?

And how would be prove it ?

*We will employ the following machines in our proof.*

0

q1,1

1

1

0 0

q0,0 q2,2

1

Ms : A mod 3 counter

M1 :

0/0 0/1

1/1



q q

1/0

Highlights:

**UNIT II:**

**Regular Expressions**

* A regular expression is used to specify a language, and it does so precisely.
* Regular expressions are very intuitive.
* Regular expressions are very useful in a variety of contexts.
* Given a regular expression, an NFA-ε can be constructed from it automatically.
* Thus, so can an NFA, a DFA, and a corresponding program, all automatically!

Definition:

* Let Σ be an alphabet. The regular expressions over Σ are:
* Ø Represents the empty set { }
* ε Represents the set {ε}
* a Represents the set {a}, for any symbol a in Σ

Let r and s be regular expressions that represent the sets R and S, respectively.

* r+s Represents the set R U S (precedence 3)
* rs Represents the set RS (precedence 2)
* r\* Represents the set R\* (highest precedence)
* (r) Represents the set R (not an op, provides precedence)
* If r is a regular expression, then L(r) is used to denote the corresponding language.
* **Examples:** Let Σ = {0, 1}

(0 + 1)\* All strings of 0’s and 1’s

0(0 + 1)\* All strings of 0’s and 1’s, beginning with a 0

(0 + 1)\*1 All strings of 0’s and 1’s, ending with a 1

(0 + 1)\*0(0 + 1)\* All strings of 0’s and 1’s containing at least one 0 (0 + 1)\*0(0 + 1)\*0(0 + 1)\* All strings of 0’s and 1’s containing at least two 0’s (0 + 1)\*01\*01\* All strings of 0’s and 1’s containing at least two 0’s (1 + 01\*0)\* All strings of 0’s and 1’s containing an even number of 0’s 1\*(01\*01\*)\* All strings of 0’s and 1’s containing an even number of 0’s (1\*01\*0)\*1\* All strings of 0’s and 1’s containing an even number of 0’s

Identities:

1. Øu = uØ = Ø Multiply by 0
2. εu = uε = u Multiply by 1
3. Ø\* = ε
4. ε\* = ε
5. u+v = v+u
6. u + Ø = u
7. u + u = u 8. u\* = (u\*)\*
8. u(v+w) = uv+uw
9. (u+v)w = uw+vw

11. (uv)\*u = u(vu)\* 12. (u+v)\* = (u\*+v)\*

= u\*(u+v)\*

= (u+vu\*)\*

= (u\*v\*)\*

= u\*(vu\*)\*

= (u\*v)\*u\*

Equivalence of Regular Expressions and NFA-ε

* **Note:** Throughout the following, keep in mind that a string is accepted by an NFA-ε if there exists a path from the start state to a final state.
* **Lemma 1:** Let r be a regular expression. Then there exists an NFA-ε M such that L(M) = L(r). Furthermore, M has exactly one final state with no transitions out of it.
* **Proof:** (by induction on the number of operators, denoted by OP(r), in r).
* **Basis:** OP(r) = 0

Then r is either Ø, ε, or **a**, for some symbol **a** in Σ

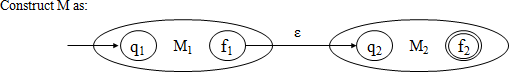
* **Inductive Hypothesis:** Suppose there exists a k  0 such that for any regular expression r where 0  OP(r)  k, there exists an NFA-ε such that L(M) = L(r). Furthermore, suppose that M has exactly one final state.
* **Inductive Step:** Let r be a regular expression with k + 1 operators (OP(r) = k + 1), where k + 1 >= 1.

Case 1) r = r1 + r2

Since OP(r) = k +1, it follows that 0<= OP(r1), OP(r2) <= k. By the inductive hypothesis there exist NFA-ε machines M1 and M2 such that L(M1) = L(r1) and L(M2) = L(r2). Furthermore, both M1 and M2 have exactly one final state.

Case 2) r = r1r2

Since OP(r) = k+1, it follows that 0<= OP(r1), OP(r2) <= k. By the inductive hypothesis there exist NFA-ε machines M1 and M2 such that L(M1) = L(r1) and L(M2) = L(r2). Furthermore, both M1 and M2 have exactly one final state.



Case 3) r = r1\*

Since OP(r) = k+1, it follows that 0<= OP(r1) <= k. By the inductive hypothesis there exists an NFA-ε machine M1 such that L(M1) = L(r1). Furthermore, M1 has exactly one final state.

* Example:

Problem: Construct FA equivalent to RE, r = 0(0+1)\*

Solution: r = r1r2 r1 = 0

r2 = (0+1)\*

r2 = r3\* r3 = 0+1

r3 = r4 + r5 r4 = 0

r5 = 1

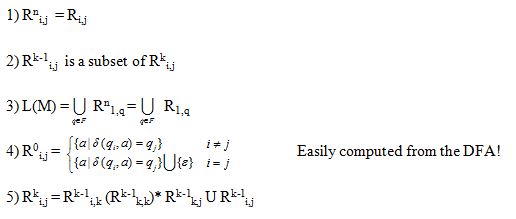
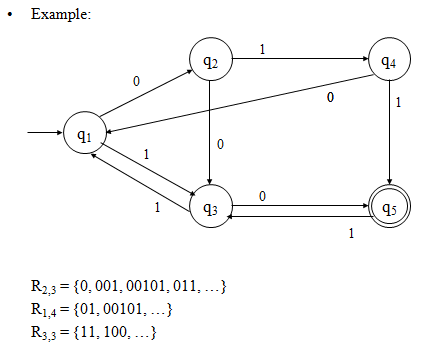
Transition graph:

Definitions Required to Convert a DFA to a Regular Expression

* Let M = (Q, Σ, δ, q1, F) be a DFA with state set Q = {q1, q2, …, qn}, and define: Ri,j = { x | x is in Σ\* and δ(qi,x) = qj}

Ri,j is the set of all strings that define a path in M from qi to qj.

* Note that states have been numbered starting at 1!
* Observations:



* **Lemma 2:** Let M = (Q, Σ, δ, q1, F) be a DFA. Then there exists a regular expression r such that L(M) = L(r).
* Proof:

First we will show (by induction on k) that for all i,j, and k, where 1  i,j  n And 0  k  n, that there exists a regular expression r such that L(r) = Rki,j .

Basis: k=0

R0i,j contains single symbols, one for each transition from qi to qj, and possibly ε if i=j.

Case 1) No transitions from qi to qj and i != j r0i,j = Ø

Case 2) At least one (m  1) transition from qi to qj and i != j

r0i,j = a1 + a2 + a3 + … + am where δ(qi, ap) = qj,

for all 1  p  m

Case 3) No transitions from qi to qj and i = j r0i,j = ε

Case 4) At least one (m  1) transition from qi to qj and i = j

r0i,j = a1 + a2 + a3 + … + am + ε where δ(qi, ap) = qj

for all 1  p  m

* Inductive Hypothesis:

Suppose that Rk-1i,j can be represented by the regular expression rk-1i,j for all 1  i,j  n, and some k1.

* Inductive Step:

Consider Rk

i,j

= Rk-1

i,k

k-1

k,k

(R

\* k-1 k,j

U Rk-1

i,j

. By the inductive hypothesis there

exist regular expressions rk-1i,k , rk-1k,k , rk-1k,j , and rk-1i,j generating Rk-1i,k , Rk-1 ,

) R

k,k

k-1

R

k,j

, and Rk-1

i,j

, respectively. Thus, if we let

k k-1 k-1 \* k-1 k-1 i,k k,k k,j i,j

r i,j = r (r ) r + r

then rki,j is a regular expression generating Rki,j ,i.e., L(rki,j) = Rk .

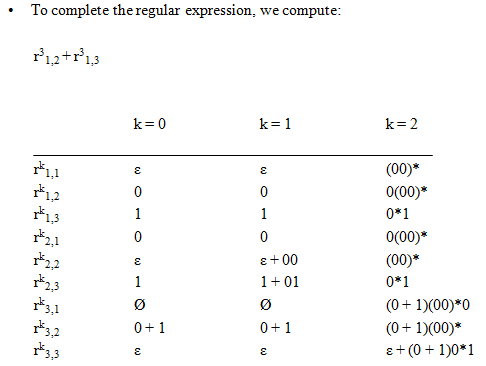
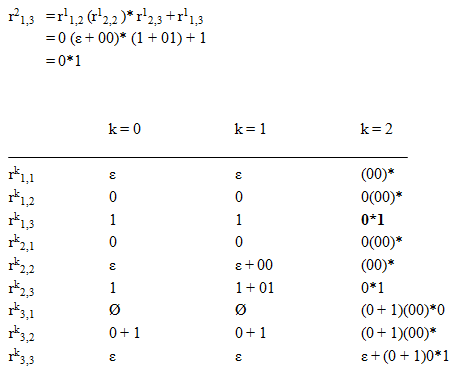
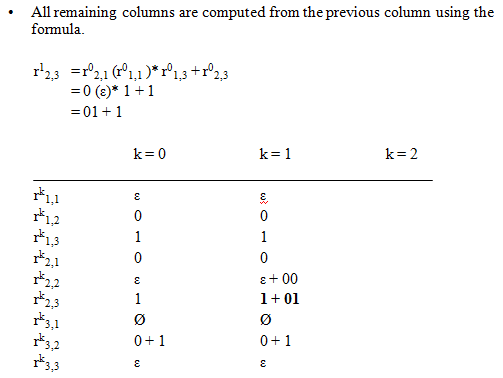
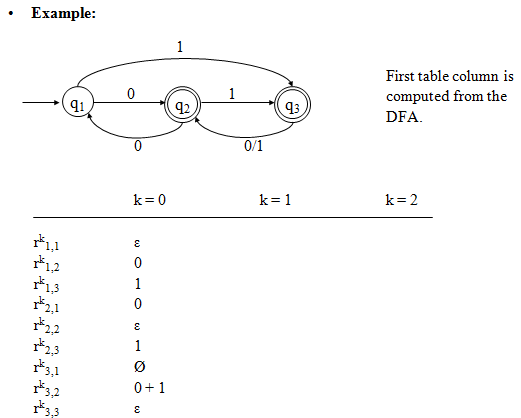
i,j

* Finally, if F = {qj1, qj2, …, qjr}, then

rn1,j1 + rn + … + rn1,jr

1,j2

is a regular expression generating L(M).•

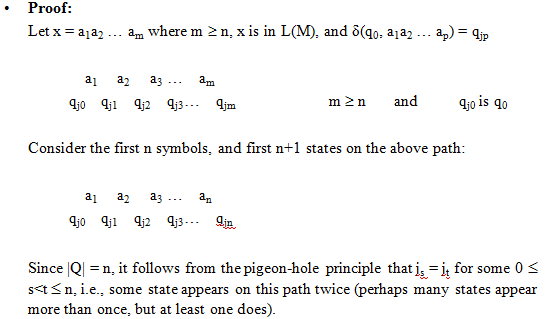


Pumping Lemma for Regular Languages

* Pumping Lemma relates the size of string accepted with the number of states in a DFA
* What is the largest string accepted by a DFA with *n* states?
* Suppose there is no loop?

Now, if there is a loop, what type of strings are accepted *via* the loop(s)?

* **Lemma:** (the pumping lemma)



Let M be a DFA with |Q| = n states. If there exists a string x in L(M), such that |x|  n, then there exists a way to write it as x = uvw, where u,v, and w are all in Σ\* and:

– 1 |uv|  n

– |v|  1

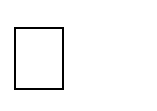
* + such that, the strings uviw are also in L(M), for all i  0
* Let:
  + u = a1…as
  + v = as+1…at
* Since 0  s<t  n and uv = a1…at it follows that:
  + 1  |v| and therefore 1  |uv|
  + |uv|  n and therefore 1  |uv|  n
* In addition, let:
  + w = at+1…am
* It follows that uviw = a1…as(as+1…at)iat+1…am is in L(M), for all i  0.

*In other words, when processing the accepted string x, the loop was traversed once, but could have been traversed as many times as desired, and the resulting string would still be accepted.*

Closure Properties of Regular Languages

1. Closure Under Union

If L and M are regular languages, so is L ⋃ M.



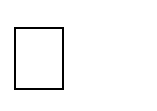
Proof: Let L and M be the languages of regular expressions R and S, respectively. Then R+S is a regular expression whose language is L ⋃ M.

1. Closure Under Concatenation and Kleene Closure RS is a regular expression whose language is LM. R\* is a regular expression whose language is L\*.



1. Closure Under Intersection

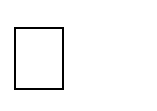
If L and M are regular languages, then so is L ⋂ M.



Proof: Let A and B be DFA’s whose languages are L and M, respectively.

1. Closure Under Difference

If L and M are regular languages, then so is L – M = strings in L but not M. Proof: Let A and B be DFA’s whose languages are L and M, respectively.

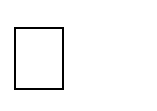


1. Closure Under Complementation

The complement of language L (w.r.t. an alphabet Σ such that Σ\* contains L) is Σ\* – L. Since Σ\* is surely regular, the complement of a regular language is always regular.



1. Closure Under Homomorphism

If L is a regular language, and h is a homomorphism on its alphabet, then h(L) = {h(w) | w is in L} is also a regular language.

Grammar

* **Definition:** A grammar G is defined as a 4-tuple, G = (V, T, S, P) Where,
  + V is a finite set of objects called variables,
  + T is a finite set of objects called terminal symbols,
  + S ∈ V is a special symbol called start variable,
  + P is a finite set of productions.

Assume that V and T are non-empty and disjoint.

* Example:

Consider the grammar G = ({S}, {a, b}, S, P) with P given by S  aSb, S ε\_.

For instance, we have S ⇒ aSb ⇒ aaSbb ⇒ aabb.

It is not hard to conjecture that L(G) = {anbn | n ≥ 0}.

Right, Left-Linear Grammar

* **Right-linear Grammar:** A grammar G = (V, T, S, P) is said to be right-linear if all productions are of the form:

A  xB, A  x,

Where A, B ∈ V and x ∈ T\*.

* Example#1:

***S* → *abS* | *a*** is an example of a right-linear grammar.

* + Can you figure out what language it generates?
  + ***L =* {*w ∈* {*a,b*}*\* | w***

Contains alternating ***a***'s and ***b***'s , begins with an ***a***, and ends with a ***b*}**

⋃ **{*a*}**

* + ***L*((*ab*)*\*a*)**
* **Left-linear Grammar:** A grammar G = (V, T, S, P) is said to be left-linear if all productions are of the form:

A  Bx, A  x,

Where A, B ∈ V and x ∈ T\*.

* Example#2:

***S* → *Aab***

***A* → *Aab* | *aB B* → *a***

is an example of a left-linear grammar.

* + Can you figure out what language it generates?
  + ***L* = {*w* Î {*a,b*}\* | *w*** is ***aa*** followed by at least one set of alternating ***ab***'s**}**
  + ***L*(*aaab*(*ab*)\*)**
* Example#3:

Consider the grammar

***S* → *A***

***A* → *aB* | λ**

***B* → *Ab***

This grammar is NOT regular.

* + No "mixing and matching" left- and right-recursive productions.

Regular Grammar

* A linear grammar is a grammar in which at most one variable can occur on the right side of any production without restriction on the position of this variable.
* An example of linear grammar is G = ({S, S1, S2}, {a, b}, S, P) with S  S1ab,

S1  S1ab | S2, S2  a.

* A **regular grammar** is one that is either right-linear or left-liner.

Testing Equivalence of Regular Languages

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

To test if L = M

* 1. Convert both L and M to DFA's.
  2. Imagine the DFA that is the union of the two DFA's (never mind there are two start states)
  3. If TF-algo says that the two start states are distinguishable, then L 6= M, otherwise, L = M.

Example:

We can “see" that both DFA accept L(ε+(0+1)\*0). The result of the TF-algo is

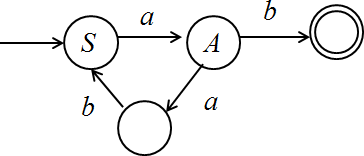
Therefore the two automata are equivalent.

Regular Grammars and NFA's

* It's not hard to show that regular grammars generate and nfa's accept the same class of languages: the regular languages!
* It's a long proof, where we must show that
* Any finite automaton has a corresponding left- or right-linear grammar,
* And any regular grammar has a corresponding nfa.
* Example:

o We get a feel for this by example.

Let ***S* → *aA A → abS* | *b***



CONTEXT FREE-GRAMMAR

* **Definition:** Context-Free Grammar (CFG) has 4-tuple: G = (V, T, P, S)

Where,

V - A finite set of variables or *non-terminals*

T - A finite set of *terminals* (V and T do not intersect) P - A finite set of *productions*, each of the form A –> α,

Where A is in V and α is in (V U T)\* Note: that α may be ε.

S - A starting non-terminal (S is in V)

* Example#1 CFG:

G = ({S}, {0, 1}, P, S) P:

1. S –> 0S1 or just simply S –> 0S1 | ε
2. S –> ε

* Example Derivations:

S => 0S1 (1)

S => ε (2)

=> 01 (2)

S => 0S1 (1)

=> 00S11 (1)

=> 000S111 (1)

=> 000111 (2)

* Note that G “generates” the language {0k1k | k>=0}

Derivation (or Parse) Tree

* **Definition:** Let G = (V, T, P, S) be a CFG. A tree is a derivation (or parse) tree if:
  + Every vertex has a label from V U T U {ε}
  + The label of the root is S
  + If a vertex with label A has children with labels X1, X2,…, Xn, from left to right, then

A –> X1, X2,…, Xn

must be a production in P

* + If a vertex has label ε, then that vertex is a leaf and the only child of its’ parent
* More Generally, a derivation tree can be defined with any non-terminal as the root.

a sentential

**Definition:** A derivation is *leftmost (rightmost)* if at each step in the derivation a production is applied to the leftmost (rightmost) non-terminal in the sentential form.

* The first derivation above is **leftmost**, second is **rightmost** and the third is neither.

UNIV III:

**Ambiguity in Context Free Grammar**

* **Definition:** Let G be a CFG. Then G is said to be ambiguous if there exists an x in L(G) with >1 leftmost derivations. Equivalently, G is said to be ambiguous if there exists an x in L(G) with >1 parse trees, or >1 rightmost derivations.
* Note: Given a CFL L, there may be more than one CFG G with L = L(G). Some ambiguous and some not.
* Definition: Let L be a CFL. If every CFG G with L = L(G) is ambiguous, then L is inherently ambiguous.
* **Example:** Consider the string aaab and the preceding grammar.
* The string has two left-most derivations, and therefore has two distinct parse trees and is ambiguous .

Eliminations of Useless Symbols

* **Definition:**

Let *G* = (*V*, *T*, *S*, *P*) be a context-free grammar. A variable *A*  *V* is said to be useful if and only if there is at least one *w*  *L*(*G*) such that

*S*  *xAy*  *w*

with *x*, *y*  (*V*  *T*).

In words, a variable is useful if and only if it occurs in at least on derivation. A variable that is not useful is called useless. A production is useless if it involves any useless variable

* For a grammar with productions

*S*  *aSb* |  | *A A*  *aA*

*A* is useless variable and the production *S*  *A* plays no role since *A* cannot be eventually transformed into a terminal string; while *A* can appear in a sentential form derived from *S*, this sentential form can never lead to sentence!

Hence, removing *S*  *A* (and *A*  *aA*) does not change the language, but does simplify the grammar.

* For a grammar with productions

*S*  *A*

*A*  *aA* | 

*B*  *bA*

*B* is useless so is the production *B*  *bA*! Observe that, even though a terminal string can be derived from *B*, there is no way to get to *B* from *S*, i.e. cannot achieve

*S*  *xBy*.

* Example:

Eliminate useless symbols and productions from *G* = (*V*, *T*, *S*, *P*), where

*V* = {*S*, *A*, *B*, *C*}, *T* = {*a*, *b*} and

*P* consists of

*S*  *aS* | *A* | *C A*  *a*

*B*  *aa C*  *aCb*

First, note that the variable *C* cannot lead to any terminal string, we can then remove *C* and its associated productions, we get *G*1 with *V*1 = {*S*, *A*, *B*}, *T*1 = {*a*} and *P*1 consisting of

*S*  *aS* | *A A*  *a*

*B*  *aa*

Next, we identify variables that cannot be reached from the start variable. We can create a dependency graph for *V*1. For a context-free grammar, a dependency graph has its vertices labeled with variables with an edge between any two vertices *I* and *J* if there is a production of the form

*I*  *xJy*

Consequently, the variable *B* is shown to be useless and can be removed together with its associated production.

The resulting grammar *G*’ = (*V*’, *T*’, *S*, *P*’) is with *V*’ = {*S*, *A*}, *T*’ = {*a*} and *P*’ consisting of

*S*  *aS* | *A A*  *a*

Eliminations of -Production

* **Definition :**

1. Any production of a context-free grammar of the form

*A*  

is called a -production.

1. Any variable *A* for which the derivation

*A*  

is possible is called nullable.

* If a grammar contains some -productions or nullable variables but does not generate the language that contains an empty string, the -productions can be removed!
* Example:

Consider the grammar, *G* with productions

*S*  *aS*1*b*

*S*1  *aS*1*b* | 

*L*(*G*) = {*anbn* | *n*  1} which is a -free language. The -production can be removed after adding new productions obtained by substituting  for *S*1 on the right hand side.

We get an equivalent *G*’ with productions

*S*  *aS*1*b* | *ab S*1  *aS*1*b* | *ab*

* Theorem:

Let *G* be any context-free grammar with   *L*(*G*). There exists an equivalent grammar

*G*’ without -productions.

Proof :

Find the set *VN* of all nullable variables of *G*

1. For all productions *A*  , put *A* in *VN*
2. Repeat the following step until no further variables are added to *VN*: For all productions

*B*  *A*1*A*2…*An*

where *A*1, *A*2, …, *An* are in *VN*, put *B* in *VN*.

With the resulting *VN*, *P*’ can be constructed by looking at all productions in *P* of the form

*A*  *x*1*x*2…*xm*, *m*  1 where each *xi*  *V*  *T*.

For each such production of *P*, we put in *P*’ the production plus all productions generated by replacing nullable variables with  in all possible combinations. However, if all *xi* are nullable, the resulting production *A*   is not put in *P*’.

* Example:

For the grammar *G* with

*S*  *ABaC A*  *BC*

*B*  *b* |  *C*  *D* |  *D*  *d*

the nullable variables are *A*, *B*, and *C*.

The equivalent grammar *G*’ without -productions has *P’* containing

*S*  *ABaC* | *BaC* | *AaC* | *ABa* | *aC* | *Ba* | *Aa* | *a A*  *BC* | *C* | *B*

*B*  *b C*  *D D*  *d*

Eliminations of Unit-Production

* **Definition:**

Any production of a context-free grammar of the form

*A*  *B*

where *A*, *B*  *V* is called a unit-production.

* Theorem:

Let *G* = (*V*, *T*, *S*, *P*) be any context-free grammar without -productions. There exists a context-free grammar *G*’ = (*V*’, *T*’, *S*, *P*’) that does not have any unit-productions and that is equivalent to *G*.

Proof:

First of all, Any unit-production of the form *A*  *A* can be removed without any effect. We then need to consider productions of the form *A*  *B* where *A* and *B* are different variables.

Straightforward replacement of *B* (with *x*1 = *x*2 = ) runs into a problem when we have

*A*  *B B*  *A*

We need to find for each *A*, all variables *B* such that

*A*  *B*

This can be done via a dependency graph with an edge (*I*, *J*) whenever the grammar *G*

has a unit-production *I*  *J*; *A*  *B* whenever there is a walk from *A* to *B* in the graph.

The new grammar *G*’ is generated by first putting in *P*’ all non-unit-productions of *P*. Then, for all *A* and *B* with *A*  *B*, we add to *P*’

*A*  *y*1 | *y*2 | … | *yn*

where *B*  *y*1 | *y*2 | … | *yn* is the set of all rules in *P*’ with *B* on the left. Not that the rules are taken from *P*’, therefore, none of *yi* can be a single variable! Consequently, no unit- productions are created by this step.

* Example:

Consider a grammar *G* with

*S*  *Aa* | *B*

*A*  *a* | *bc* | *B B*  *A* | *bb*

Its unit-production dependency graph is show below

We have *S*  *A*, *S*  *B*, *A*  *B* and

*B*  *A*.

First, for the set of original non-unit-productions, we have

*S*  *Aa*

*A*  *a* | *bc B*  *bb*

We then add the new rules

*S*  *a* | *bc* | *bb A*  *bb*

*B*  *a* | *bc*

We finally obtain the equivalent grammar *G*’ with *P*’ consisting of

*S*  *Aa* | *a* | *bc* | *bb A*  *a | bc | bb*

*B*  *bb* | *a* | *bc*

Notice that B and its associate production become useless.

Minimization of Context Free Grammar

* **Theorem:**

Let *L* be a context-free language that does not contain . There exists a context-free grammar that generates *L* and that does not have any useless productions, -productions or unit-productions.

Proof:

We need to remove the undesirable productions using the following sequence of steps.

* 1. Remove -productions
  2. Remove unit-productions
  3. Remove useless productions
* Definition:

**Chomsky Normal Form**

A context-free grammar is in Chomsky normal form if all productions are of the form

*A*  *BC*

or

*A*  *a*

where *A*, *B*, *C*  *V*, and *a*  *T*.

**Note:** that the number of symbols on the right side of productions is strictly limited; not more than two symbols.

* Example:

The following grammar is in Chomsky normal form.

*S*  *AS* | *a A*  *SA* | *b*

On the other hand, the grammar below is not.

*S*  *AS* | *AAS*

*A*  *SA* | *aa*

* Theorem:

Any context-free grammar *G* = (*V*, *T*, *S*, *P*) with   *L*(*G*) has an equivalent grammar *G*’

= (*V*’, *T*’, *S*, *P*’) in Chomsky normal form.

Proof:

First we assume (based on previous Theorem) without loss of generality that *G* has no - productions and no unit-productions. Then, we show how to construct *G*’ in two steps.

Step 1:

Construct a grammar *G*1 = (*V*1, *T*, *S*, *P*1) from *G* by considering all productions in

*P* of the form

*A*  *x*1*x*2…*xn*

Where each *xi* is a symbol either in *V* or in *T*.

Note that if *n* = 1, *x*1 must be a terminal because there is no unit-productions in *G*. In this case, put the production into *P*1.

If *n*  2, introduce new variables *Ba* for each *a*  *T*. Then, for each production of the form *A*  *x*1*x*2…*xn*, we shall remove all terminals from productions whose right side has length greater than one

This is done by putting into *P*1 a production

*A*  *C*1*C*2…*Cn*

Where

And

*Ci* = *xi* if *xi*  *V Ci* = *Ba* if *xi* = *a*

And, for every *Ba*, we also put into *P*1 a production

*Ba*  *a*

As a consequence of Theorem 6.1, it can be claimed that

*L*(*G*1) = *L*(*G*)

Step 2:

The length of right side of productions is reduced by means of additional variables wherever necessary. First of all, all productions with a single terminal or two variables (*n* = 2) are put into *P*’. Then, for any production with *n*  2, new variables *D*1, *D*2, … are introduced and the following productions are put into *P*’.

*A*  *C*1*D*1 *D*1  *C*2*D*2

…

*Dn*-2  *Cn*-1*Cn*

*G*’ is clearly in Chomsky normal form.

* Example:

Convert to Chomsky normal form the following grammar *G* with productions.

*S*  *ABa A*  *aab B*  *Ac*

Solution:

Step 1:

New variables *Ba*, *Bb*, *Bc* are introduced and a new grammar *G*1 is obtained.

*S*  *ABBa*

*A*  *BaBaBb B*  *ABc*

*Ba*  *a Bb*  *b Bc*  *c*

Step 2:

Additional variables are introduced to reduce the length of the first two productions making them into the normal form, we finally obtain *G*’.

*S*  *AD*1

*D*1  *BBa A*  *BaD*2 *D*2  *BaBb B*  *ABc Ba*  *a*

*Bb*  *b Bc*  *c*

* Definition:

**Greibach normal form**

A context-free grammar is said to be in Greibach normal form if all productions have the form

*A*  *ax*

where *a*  *T* and *x*  *V*

**Note** that the restriction here is not on the number of symbols on the right side, but rather on the positions of the terminals and variables.

* Example:

The following grammar is not in Greibach normal form.

*S*  *AB*

*A*  *aA* | *bB* | *b B*  *b*

It can, however, be converted to the following equivalent grammar in Greibach normal form.

* Theorem:

*S*  *aAB | bBB | bB A*  *aA* | *bB* | *b*

*B*  *b*

For every context-free grammar *G* with  *L*(*G*), there exists an equivalent grammar *G*’ in Greibach normal form.

Conversion

* Convert from Chomsky to Greibach in two steps:

1. From Chomsky to intermediate grammar
   1. Eliminate direct left recursion
   2. Use *A*  *uBv* rules transformations to improve references (explained later)
2. From intermediate grammar into Greibach 1.a) Eliminate direct left recursion

Step1:

* Before

*A*  *Aa* | ***b***

* After

*A*  ***b****Z* | ***b***

*Z*  *aZ* | *a*

* Remove the rule with direct left recursion, and create a new one with recursion on the right

Step2:

* Before

*A*  *Aa* | *Ab* | ***b*** | ***c***

* After

*A*  ***b****Z* | ***c****Z* | ***b*** | ***c***

*Z*  *aZ* | *bZ* | *a* | *b*

* Remove the rules with direct left recursion, and create new ones with recursion on the right

Step3:

* Before

*A*  *AB* | ***BA*** | ***a***

*B*  *b* | *c*

* After

*A*  ***BA****Z* | ***a****Z* | ***BA*** | ***a***

*Z*  *BZ* | *B B*  *b* | *c*

* 1. Transform *A*  *uBv* rules
     + Before

*A*  *uBb*

*B*  *w1 | w1 |…| wn*

* + - After

Add *A*  *uw1b | uw1b |…| uwnb*

Delete *A*  *uBb*

Background Information for the Pumping Lemma for Context-Free Languages

* **Definition:** Let G = (V, T, P, S) be a CFL. If every production in P is of the form

A –> BC

or A –> a

where A, B and C are all in V and a is in T, then G is in Chomsky Normal Form (CNF).

* Example:

S –> AB | BA | aSb A –> a

B –> b

* **Theorem:** Let L be a CFL. Then L – {ε} is a CFL.
* **Theorem:** Let L be a CFL not containing {ε}. Then there exists a CNF grammar G such that L = L(G).
* **Definition:** Let T be a tree. Then the height of T, denoted h(T), is defined as follows:
  + If T consists of a single vertex then h(T) = 0
  + If T consists of a root r and subtrees T1, T2, … Tk, then h(T) = maxi{h(Ti)} + 1
* **Lemma:** Let G be a CFG in CNF. In addition, let w be a string of terminals where A=>\*w and w has a derivation tree T. If T has height h(T)1, then |w|  2h(T)-1.
* **Proof:** By induction on h(T) (exercise).
* **Corollary:** Let G be a CFG in CNF, and let w be a string in L(G). If |w|  2k, where k 

0, then any derivation tree for w using G has height at least k+1.

* **Proof:** Follows from the lemma.

Pumping Lemma for Context-Free Languages

* **Lemma:**

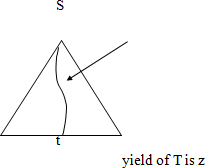
Let G = (V, T, P, S) be a CFG in CNF, and let n = 2|V|. If z is a string in L(G) and |z|  n, then there exist strings u, v, w, x and y in T\* such that z=uvwxy and:

– |vx|  1 (i.e., |v| + |x|  1)

* + |vwx|  n
  + uviwxiy is in L(G), for all i  0
* Proof:

Since |z|  n = 2k, where k = |V|, it follows from the corollary that any derivation tree for z has height at least k+1.

By definition such a tree contains a path of length at least k+1. Consider the longest such path in the tree:



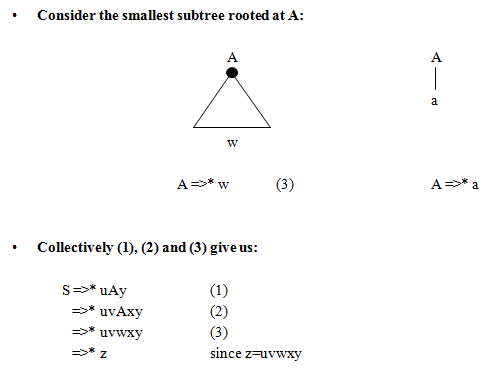
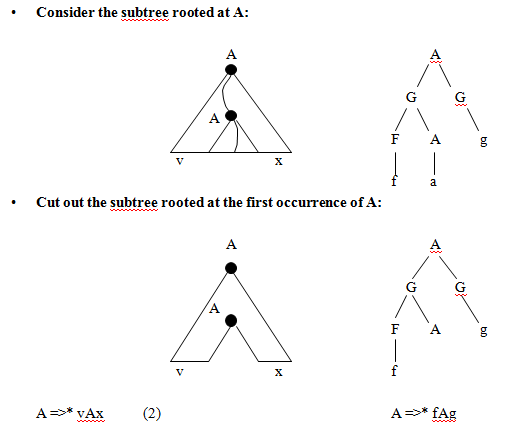
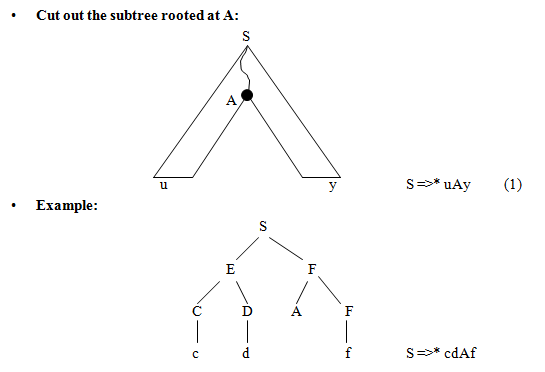
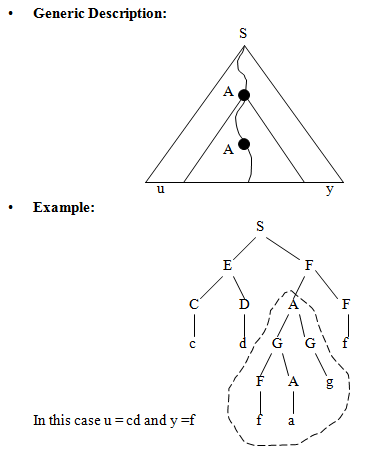
Such a path has:

* Length  k+1 (i.e., number of edges in the path is  k+1)
* At least k+2 nodes
* 1 terminal

At least k+1 non-terminals

* Since there are only k non-terminals in the grammar, and since k+1 appear on this long path, it follows that some non-terminal (and perhaps many) appears at least twice on this path.
* Consider the first non-terminal that is repeated, when traversing the path from the leaf to the root.

This path, and the non-terminal A will be used to break up the string z.



* In addition, (2) also tells us:

S =>\* uAy (1)

=>\* uvAxy (2)

=>\* uv2Ax2y (2)

=>\* uv2wx2y (3)

* More generally:

S =>\* uviwxiy for all i>=1

* And also:

S =>\* uAy (1)

=>\* uwy (3)

* Hence:

S =>\* uviwxiy for all i>=0

* Consider the statement of the Pumping Lemma:
  + *What is n?*

n = 2k, where k is the number of non-terminals in the grammar.

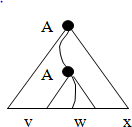
– *Why is |v| + |x|*  *1?*

Since the height of this subtree is  2, the first production is A->V1V2. Since no non- terminal derives the empty string (in CNF), either V1 or V2 must derive a non-empty v or x. More specifically, if w is generated by V1, then x contains at least one symbol, and if w is generated by V2, then v contains at least one symbol.

* + *Why is |vwx|*  *n?*

Observations:

* + - The repeated variable was the first repeated variable on the path from the bottom, and therefore (by the pigeon-hole principle) the path from the leaf to the second occurrence of the non-terminal has length at most k+1.
    - Since the path was the largest in the entire tree, this path is the longest in the subtree rooted at the second occurrence of the non-terminal. Therefore the subtree has height k+1. From the lemma, the yield of the subtree has length  2k=n.



CFL Closure Properties

* **Theorem#1:**

The context-free languages are closed under concatenation, union, and Kleene closure.

* Proof:

Start with 2 CFL *L*(*H1*) and *L*(*H2*) generated by *H1* = (*N1*,*T1*,*R1*,*s1*) and *H2* = (*N2*,*T2*,*R2*,*s2*).

Assume that the alphabets and rules are disjoint.

*Concatenation:*

Formed by *L*(*H1*)·*L*(*H2*) or a string in *L*(*H1*) followed by a string in *L*(*H2*) which can be generated by *L*(*H3*) generated by *H3* = (*N3*,*T3*,*R3*,*s3*). *N3* = *N1* ⋃ *N2*, *T3* = *T1* ⋃ *T2*, *R3*

= *R1* ⋃ *R2* ⋃ {*s3* -->*s1s2*} where *s3 s1s2* is a new rule introduced. The new rule generates a string of *L*(H1) then a string of *L*(*H2*). Then *L*(*H1*) ·*L*(*H2*) is context-free.

*Union:*

Formed by *L*(*H1*) ⋃ *L*(*H2*) or a string in *L*(*H1*) or a string in *L*(*H2*). It is generated by *L*(*H3*) generated by *H4* = (*N4,T4,R4,s4*) where *N4* = *N1* ⋃ *N2*, *T4* = *T1* ⋃ *T2*, and *R4* = *R1* ⋃ *R2* ⋃ {*s4*-->*s1*, *s4*  *s2*}, the new rules added will create a string of *L*(*H1*) or *L*(*H2*). Then *L*(*H1*) ⋃ *L*(*H2*) is context-free.

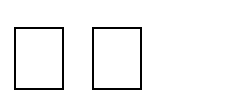
*Kleene:*

Formed by *L*(*H1*)\* is generated by the grammar *L*(*H5*) generated by *H5* = (*N1,T1,R5,s1*) with *R5* = *R1* ⋃ {*s1**e*, *s1**s1s1*}. *L*(*H5*) includes *e*, every string in *L*(*H1*), and through *i*-1 applications of *s1**s1s1*, every string in *L*(*H1*)*i*. Then *L*(*H1*)\* is generated by *H5* and is context-free.

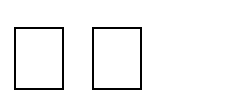
* Theorem#2:

The set of context-free languages is not closed under complementation or intersection.

* Proof:

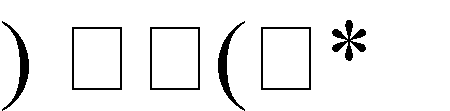
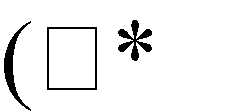
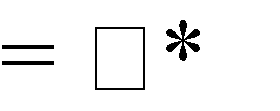
Intersections of two languages *L1 L2* can be defined in terms of the Complement and Union operations as follows:

-



*L1*

*L2*

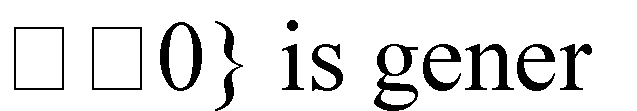


- *L1*

- *L2*)

Therefore if CFL are closed under intersection then it is closed under compliment and if closed under compliment then it is closed under intersection.

The proof is just showing two context-free languages that their intersection is not a context-free language.

Choose *L1* = {*anbncm* | *m*,*n* ated by grammar *H1* = {*N1,T1,R1,s1*}, where

*N1* = {*s*, *A*, *B*}

*T1* = {*a*, *b*, *c*} *R1* = {*s AB*,

*A aAb*,

1. *e*,
2. *Bc*,

*B e*}.

Choose *L2* = {*ambncn* | *m*,*n H2* = {*N2,T2,R2,s2*}, where

*N1* = {*s*, *A*, *B*}

*T1* = {*a*, *b*, *c*} *R2* = {*s AB*, *A aA*,

1. *e*,
2. *bBc*,

*B e*}.

Thus *L1* and *L2* are both context-free.

The intersection of the two languages is *L3* = {*anbncn* | *n*

already been proven earlier in this paper to be not context-free. Therefore CFL are not closed under intersections, which also means that it is not closed under complementation.

Pushdown Automata (PDA)

* Informally:
* A PDA is an NFA-ε with a stack.

–Transitions are modified to accommodate stack operations.

* Questions:

–What is a stack?

–How does a stack help?

* A DFA can “remember” only a finite amount of information, whereas a PDA can “remember” an infinite amount of (certain types of) information.
* Example:

{0n1n | 0=<n} Is *not* regular.

{0n1n | 0nk, for some fixed k} Is regular, for any fixed k.

* For k=3:

L = {ε, 01, 0011, 000111}

* In a DFA, each state remembers a finite amount of information.
* To get {0n1n | 0n} with a DFA would require an infinite number of states using the preceding technique.
* An infinite stack solves the problem for {0n1n | 0n} as follows:

–Read all 0’s and place them on a stack

–Read all 1’s and match with the corresponding 0’s on the stack

* Only need two states to do this in a PDA
* Similarly for {0n1m0n+m | n,m0}

Formal Definition of a PDA

* A pushdown automaton (PDA) is a seven-tuple: M = (Q, Σ, Г, δ, q0, z0, F)

Q A finite set of states

Σ A finite input alphabet

Г A finite stack alphabet

q0 The initial/starting state, q0 is in Q z0 A starting stack symbol, is in Г

F A set of final/accepting states, which is a subset of Q δ A transition function, where

δ: Q x (Σ U {ε}) x Г  finite subsets of Q x Г\*

* Consider the various parts of δ:

Q x (Σ U {ε}) x Г  finite subsets of Q x Г\*

–Q on the LHS means that at each step in a computation, a PDA must consider its’ current state.

–Г on the LHS means that at each step in a computation, a PDA must consider the symbol on top of its’ stack.

–Σ U {ε} on the LHS means that at each step in a computation, a PDA may or may not consider the current input symbol, i.e., it may have epsilon transitions.

–“Finite subsets” on the RHS means that at each step in a computation, a PDA will have several options.

–Q on the RHS means that each option specifies a new state.

–Г\* on the RHS means that each option specifies zero or more stack symbols that will replace the top stack symbol.

* Two types of PDA transitions #1:

δ(q, a, z) = {(p1,γ1), (p2,γ2),…, (pm,γm)}

–Current state is q

–Current input symbol is a

–Symbol currently on top of the stack z

–Move to state pi from q

–Replace z with γi on the stack (leftmost symbol on top)

–Move the input head to the next input symbol

* Two types of PDA transitions #2:

δ(q, ε, z) = {(p1,γ1), (p2,γ2),…, (pm,γm)}

–Current state is q

–Current input symbol is not considered

–Symbol currently on top of the stack z

–Move to state pi from q

–Replace z with γi on the stack (leftmost symbol on top)

–No input symbol is read

* **Example:** (balanced parentheses)

M = ({q1}, {“(“, “)”}, {L, #}, δ, q1, #, Ø)

δ:

(1) δ(q1, (, #) = {(q1, L#)}

(2) δ(q1, ), #) = Ø

(3) δ(q1, (, L) = {(q1, LL)}

(4) δ(q1, ), L) = {(q1, ε)}

(5) δ(q1, ε, #) = {(q1, ε)}

(6) δ(q1, ε, L) = Ø

* **Goal:** (acceptance)

–Terminate in a non-null state

–Read the entire input string

–Terminate with an empty stack

* Informally, a string is accepted if there exists a computation that uses up all the input and leaves

the stack empty.

* Transition Diagram:
* Example Computation:

Current Input Stack Transition (()) #

()) L# (1) - Could have applied rule

)) LL# (3) (5), but it would have

) L# (4) done no good

ε # (4)

ε - (5)

* **Example PDA #1:** For the language {x | x = wcwr and w in {0,1}\*} M = ({q1, q2}, {0, 1, c}, {R, B, G}, δ, q1, R, Ø)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| δ: | (1) | δ(q1, 0, R) = {(q1, BR)} | (9) | δ(q1, 1, R) = {(q1, GR)} |
|  | (2) | δ(q1, 0, B) = {(q1, BB)} | (10) | δ(q1, 1, B) = {(q1, GB)} |
|  | (3) | δ(q1, 0, G) = {(q1, BG)} | (11) | δ(q1, 1, G) = {(q1, GG)} |
|  | (4) | δ(q1, c, R) = {(q2, R)} |  |  |
|  | (5) | δ(q1, c, B) = {(q2, B)} |  |  |
|  | (6) | δ(q1, c, G) = {(q2, G)} |  |  |
|  | (7) | δ(q2, 0, B) = {(q2, ε)} | (12) | δ(q2, 1, G) = {(q2, ε)} |
|  | (8) | δ(q2, ε, R) = {(q2, ε)} |  |  |

* Notes:

–Only rule #8 is non-deterministic.

–Rule #8 is used to pop the final stack symbol off at the end of a computation.

* Example Computation:

(1) δ(q1, 0, R) = {(q1, BR)} (9) δ(q1, 1, R) = {(q1, GR)}

(2) δ(q1, 0, B) = {(q1, BB)} (10) δ(q1, 1, B) = {(q1, GB)}

(3) δ(q1, 0, G) = {(q1, BG)} (11) δ(q1, 1, G) = {(q1, GG)} (4) δ(q1, c, R) = {(q2, R)}

(5) δ(q1, c, B) = {(q2, B)}

(6) δ(q1, c, G) = {(q2, G)}

(7) δ(q2, 0, B) = {(q2, ε)} (12) δ(q2, 1, G) = {(q2, ε)} (8) δ(q2, ε, R) = {(q2, ε)}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| State | Input | Stack | Rule Applied | Rules Applicable |
| q1 | **0**1c10 | R | - | (1) |
| q1 | **1**c10 | BR | (1) | (10) |
| q1 | **c**10 | GBR | (10) | (6) |
| q2 | **1**0 | GBR | (6) | (12) |
| q2 | **0** | BR | (12) | (7) |
| q2 | **ε** | R | (7) | (8) |
| q2 | ε | ε | (8) | - |

* Example Computation:

(1) δ(q1, 0, R) = {(q1, BR)} (9) δ(q1, 1, R) = {(q1, GR)}

(2) δ(q1, 0, B) = {(q1, BB)} (10) δ(q1, 1, B) = {(q1, GB)}

(3) δ(q1, 0, G) = {(q1, BG)} (11) δ(q1, 1, G) = {(q1, GG)} (4) δ(q1, c, R) = {(q2, R)}

(5) δ(q1, c, B) = {(q2, B)}

(6) δ(q1, c, G) = {(q2, G)}

(7) δ(q2, 0, B) = {(q2, ε)} (12) δ(q2, 1, G) = {(q2, ε)} (8) δ(q2, ε, R) = {(q2, ε)}

|  |  |  |  |
| --- | --- | --- | --- |
| State | Input | Stack | Rule Applied |
| q1 | **1**c1 | R |  |
| q1 | **c**1 | GR | (9) |
| q2 | **1** | GR | (6) |
| q2 | **ε** | R | (12) |
| q2 | ε | ε | (8) |

* **Definition:** |—\* is the reflexive and transitive closure of |—.

–I |—\* I for each instantaneous description I

–If I |— J and J |—\* K then I |—\* K

* Intuitively, if I and J are instantaneous descriptions, then I |—\* J means that J follows from I by zero or more transitions.
* **Definition:** Let M = (Q, Σ, Г, δ, q0, z0, F) be a PDA. The *language accepted by empty stack*, denoted LE(M), is the set

{w | (q0, w, z0) |—\* (p, ε, ε) for some p in Q}

* **Definition:** Let M = (Q, Σ, Г, δ, q0, z0, F) be a PDA. The *language accepted by final state*, denoted LF(M), is the set

{w | (q0, w, z0) |—\* (p, ε, γ) for some p in F and γ in Г\*}

* **Definition:** Let M = (Q, Σ, Г, δ, q0, z0, F) be a PDA. The *language accepted by empty stack and final state*, denoted L(M), is the set

{w | (q0, w, z0) |—\* (p, ε, ε) for some p in F}

* **Lemma 1:** Let L = LE(M1) for some PDA M1. Then there exits a PDA M2 such that L = LF(M2).
* **Lemma 2:** Let L = LF(M1) for some PDA M1. Then there exits a PDA M2 such that L = LE(M2).
* **Theorem:** Let L be a language. Then there exits a PDA M1 such that L = LF(M1) if and only if there exists a PDA M2 such that L = LE(M2).
* **Corollary:** The PDAs that accept by empty stack and the PDAs that accept by final state define the same class of languages.
* **Note:** Similar lemmas and theorems could be stated for PDAs that accept by both final state and empty stack.

Greibach Normal Form (GNF)

* **Definition:** Let G = (V, T, P, S) be a CFL. If every production in P is of the form A –> aα

Where A is in V, a is in T, and α is in V\*, then G is said to be in Greibach Normal Form (GNF).

* Example:

S –> aAB | bB A –> aA | a

B –> bB | c

* **Theorem:** Let L be a CFL. Then L – {ε} is a CFL.
* **Theorem:** Let L be a CFL not containing {ε}. Then there exists a GNF grammar G such that L

= L(G).

* **Lemma 1:** Let L be a CFL. Then there exists a PDA M such that L = LE(M).
* **Proof:** Assume without loss of generality that ε is not in L. The construction can be modified to include ε later.

Let G = (V, T, P, S) be a CFG, and assume without loss of generality that G is in GNF. Construct M = (Q, Σ, Г, δ, q, z, Ø) where:

Q = {q} Σ = T

Г = V

z = S

δ: for all a in Σ and A in Г, δ(q, a, A) contains (q, γ) if A –> aγ is in P or rather: δ(q, a, A) = {(q, γ) | A –> aγ is in P and γ is in Г\*}, for all a in Σ and A in Г

* For a given string x in Σ\* , M will attempt to simulate a leftmost derivation of x with G.
* **Example #1:** Consider the following CFG in GNF. S  aS G is in GNF

S  a L(G) = a+

Construct M as:

Q = {q}

Σ = T = {a} Г = V = {S}

z = S

δ(q, a, S) = {(q, S), (q, ε)} δ(q, ε, S) = Ø

* **Example #2:** Consider the following CFG in GNF.

1. S –> aA
2. S –> aB
3. A –> aA G is in GNF
4. A –> aB L(G) = a+b+
5. B –> bB
6. B –> b

Construct M as:

Q = {q}

Σ = T = {a, b}

Г = V = {S, A, B}

z = S

1. δ(q, a, S) = {(q, A), (q, B)} From productions #1 and 2, S->aA, S->aB
2. δ(q, a, A) = {(q, A), (q, B)} From productions #3 and 4, A->aA, A->aB
3. δ(q, a, B) = Ø
4. δ(q, b, S) = Ø
5. δ(q, b, A) = Ø
6. δ(q, b, B) = {(q, B), (q, ε)} From productions #5 and 6, B->bB, B->b
7. δ(q, ε, S) = Ø
8. δ(q, ε, A) = Ø
9. δ(q, ε, B) = Ø Recall δ: Q x (Σ U {ε}) x Г –> finite

subsets of Q x Г\*

* For a string w in L(G) the PDA M will simulate a leftmost derivation of w.

–If w is in L(G) then (q, w, z0) |—\* (q, ε, ε)

–If (q, w, z0) |—\* (q, ε, ε) then w is in L(G)

* Consider generating a string using G. Since G is in GNF, each sentential form in a *leftmost*

derivation has form:

* And each step in the derivation (i.e., each application of a production) adds a terminal and some non-terminals.

A1 –> ti+1α

=> t1t2…ti ti+1 αA1A2…Am

* Each transition of the PDA simulates one derivation step. Thus, the ith step of the PDAs’ computation corresponds to the ith step in a corresponding leftmost derivation.
* After the ith step of the computation of the PDA, t1t2…ti+1 are the symbols that have already

been read by the PDA and αA1A2…Amare the stack contents.

* For each leftmost derivation of a string generated by the grammar, there is an equivalent accepting computation of that string by the PDA.
* Each sentential form in the leftmost derivation corresponds to an instantaneous description in the PDA’s corresponding computation.
* For example, the PDA instantaneous description corresponding to the sentential form:

=> t1t2…ti A1A2…Am

would be: (q, ti+1ti+2…tn , A1A2…Am)

* **Example:** Using the grammar from example #2:

|  |  |
| --- | --- |
| S => aA | (1) |
| => aaA | (3) |
| => aaaA | (3) |
| => aaaaB | (4) |
| => aaaabB | (5) |
| => aaaabb | (6) |

* The corresponding computation of the PDA:
* (q, aaaabb, S) |— (q, aaabb, A) (1)/1

|— (q, aabb, A) (2)/1

|— (q, abb, A) (2)/1

|— (q, bb, B) (2)/2

|— (q, b, B) (6)/1

|— (q, ε, ε) (6)/2

–String is read

–Stack is emptied

–Therefore the string is accepted by the PDA

* **Example #3:** Consider the following CFG in GNF.

1. S –> aABC
2. A –> a G is in GNF
3. B –> b
4. C –> cAB
5. C –> cC

Construct M as:

Q = {q}

Σ = T = {a, b, c}

Г = V = {S, A, B, C}

z = S

(1) δ(q, a, S) = {(q, ABC)} S->aABC (9) δ(q, c, S) = Ø

(2) δ(q, a, A) = {(q, ε)} A->a (10) δ(q, c, A) = Ø (3) δ(q, a, B) = Ø (11) δ(q, c, B) = Ø

(4) δ(q, a, C) = Ø C->cAB|cC (12) δ(q, c, C) = {(q, AB), (q, C))

(5) δ(q, b, S) = Ø (13) δ(q, ε, S) = Ø

(6) δ(q, b, A) = Ø (14) δ(q, ε, A) = Ø (7) δ(q, b, B) = {(q, ε)} B->b (15) δ(q, ε, B) = Ø (8) δ(q, b, C) = Ø (16) δ(q, ε, C) = Ø

* Notes:

–Recall that the grammar G was required to be in GNF before the construction could be applied.

–As a result, it was assumed at the start that ε was not in the context-free language L.

* Suppose ε is in L:

1. First, let L’ = L – {ε}

Fact: If L is a CFL, then L’ = L – {ε} is a CFL.

By an earlier theorem, there is GNF grammar G such that L’ = L(G).

1. Construct a PDA M such that L’ = LE(M) How do we modify M to accept ε?

Add δ(q, ε, S) = {(q, ε)}? No!

* Counter Example:

Consider L = {ε, b, ab, aab, aaab, …} Then L’ = {b, ab, aab, aaab, …}

* The GNF CFG for L’:

1. S –> aS
2. S –> b

* The PDA M Accepting L’:

Q = {q}

Σ = T = {a, b} Г = V = {S}

z = S

δ(q, a, S) = {(q, S)}

δ(q, b, S) = {(q, ε)} δ(q, ε, S) = Ø

* If δ(q, ε, S) = {(q, ε)} is added then:

L(M) = {ε, a, aa, aaa, …, b, ab, aab, aaab, …}

1. Instead, add a new *start* state q’ with transitions:

δ(q’, ε, S) = {(q’, ε), (q, S)}

* **Lemma 1:** Let L be a CFL. Then there exists a PDA M such that L = LE(M).
* **Lemma 2:** Let M be a PDA. Then there exists a CFG grammar G such that LE(M) = L(G).
* **Theorem:** Let L be a language. Then there exists a CFG G such that L = L(G) iff there exists a PDA M such that L = LE(M).
* **Corollary:** The PDAs define the CFLs.

Equivalence of CFG to PDAs

* + **Example:** Consider the grammar for arithmetic expressions we introduced earlier. It is reproduced below for convenience. G = ( {E, T, F}, {n, v, +, \*, ( , )}, P, E), where

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| E = { 1: | E |  |  | E |  | + | T, |
| 2: |  | E |  |  |  |  | T, |
| 3: | T |  |  |  |  | T | F, |
| 4: |  | T |  |  |  |  | F, |
| 5: |  | F |  |  |  |  | n, |
| 6: |  | F |  |  |  |  | v, |
| 7: | F |  |  | ( |  | E | ), |

}

Suppose the input to our parser is the expression, n\*(v+n\*v). Since G is unambiguous this expression has only one leftmost derivation, p = 2345712463456. We describe the behavior of the PDA in general, and then step through its moves using this derivation to guide the computation.

* + PDA Simulator:
    - Step 1: Initialize the stack with the start symbol (E in this case). The start symbol will serve as the bottom of stack marker (Z0).
    - Step 2: Ignoring the input, check the top symbol of the stack.
      * Case (a) Top of stack is a nonterminal, “X”: non-deterministically decide which

X-rule to use as the next step of the derivation. After selecting a rule, replace X in the stack with the rightpart of that rule. If the stack is non- empty, repeat step 2. Otherwise, halt (input may or may not be empty.)

* + - * Case(b) Top of stack is a terminal, “a”: Read the next input. If the input matches a, then pop the stack and repeat step 2.

Otherwise, halt (without popping “a” from the stack.)

* + - This parsing algorithm by showing the sequence of configurations the parser would assume in an accepting computation for the input, n\*(v+n\*v).

Assume “q0” is the one and only state of this PDA.

* + - p (leftmost derivation in G) = 2345712463456 (q0, n\*(v+n\*v), E)

2M (q0, n\*(v+n\*v), T)

3M (q0, n\*(v+n\*v), T\*F)

4M (q0, n\*(v+n\*v), F\*F)

5M (q0, n\*(v+n\*v), n\*F) readM (q0, \*(v+n\*v), \*F)

readM (q0, (v+n\*v), F) 7M (q0, (v+n\*v), (E) ) readM (q0, v+n\*v), E) ) 1M (q0, v+n\*v),E+T) )

2M (q0, v+n\*v), T+T) )

4M (q0, v+n\*v), F+T) )

6M (q0, v+n\*v), v+T) ) readM (q0, +n\*v), +T) )

readM (q0, n\*v), T) )

3M (q0, n\*v), T\*F) )

4M (q0, n\*v), F\*F) )

5M (q0, n\*v), n\*F) ) readM (q0, \*v), \*F) ) readM (q0, v), F) )

6M (q0, v), v) ) readM (q0, ), ) ) readM (q0, l, l ) accept!

Deterministic PDAs and DCFLs

* + **Definition:** A *Deterministic Pushdown Automaton* (DPDA) is a 7-tuple, M = (Q, , , , q0, Z0, A),

where

Q = finite set of states,

 = input alphabet,

 = stack alphabet,

q0  Q = the initial state,

Z0  = bottom of stack marker (or initial stack symbol), and

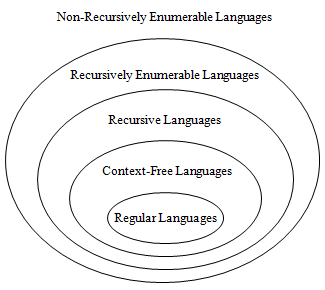
: Q  ( {L})    Q  \* = the transition function (not necessarily total). Specifically,

1. if d(q, a, Z) is defined for some a  and Z , then d(q, L, Z) =  and

d(q, a, Z)= 1.

1. Conversely, if d(q, L, Z)  , for some Z, then d(q, a, Z)  , for all a , and d(q, L, Z)= 1.
   * **NOTE:** DPDAs can accept their input either by final state or by empty stack – just as for the non-deterministic model. We therefore define *Dstk* and *Dste*, respectively, as the corresponding families of Deterministic Context-free Languages accepted by a DPDA by empty stack and final state.

UNIT IV:



**Turing Machines (TM)**

* + **Generalize the class of CFLs:**
* **Another Part of the Hierarchy:**
* Recursively enumerable languages are also known as *type 0* languages.
* Context-sensitive languages are also known as *type 1* languages.
* Context-free languages are also known as *type 2* languages.
* Regular languages are also known as *type 3* languages.
* TMs model the computing capability of a general purpose computer, which informally can be described as:
* Effective procedure
  + Finitely describable
  + Well defined, discrete, “mechanical” steps
  + Always terminates
* Computable function
  + A function computable by an effective procedure
* TMs formalize the above notion.

Deterministic Turing Machine (DTM)

* Two-way, infinite tape, broken into cells, each containing one symbol.
* Two-way, read/write tape head.
* Finite control, i.e., a program, containing the position of the read head, current symbol being scanned, and the current state.
* An input string is placed on the tape, padded to the left and right infinitely with blanks, read/write head is positioned at the left end of input string.
* In one move, depending on the current state and the current symbol being scanned, the TM 1) changes state, 2) prints a symbol over the cell being scanned, and 3) moves its’ tape head one cell left or right.
* Many modifications possible.

Formal Definition of a DTM

* A DTM is a seven-tuple: M = (Q, Σ, Γ, δ, q0, B, F)

Q A finite set of states

Γ A finite tape alphabet

B A distinguished blank symbol, which is in Γ

Σ A finite input alphabet, which is a subset of Γ– {B} q0 The initial/starting state, q0 is in Q

F A set of final/accepting states, which is a subset of Q δ A next-move function, which is a *mapping* from

Q x Γ –> Q x Γ x {L,R}

Intuitively, δ(q,s) specifies the next state, symbol to be written and the direction of tape head movement by M after reading symbol s while in

state q.

* **Example #1:** {0n1n | n >= 1}

0 1 X Y B

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| q0 | (q1, X, R) | - | - | (q3, Y, R) | - |
| q1 | (q1, 0, R) | (q2, Y, L) | - | (q1, Y, R) | - |
| q2 | (q2, 0, L) | - | (q0, X, R) | (q2, Y, L) | - |
| q3 | - | - | - | (q3, Y, R) | (q4, B, R) |
| q4 | - | - | - | - | - |

•

– **Example #1:** {0n1n | n >= 1}

0 1 X Y B q0 (q1, X, R) - - (q3, Y, R) -

q1 (q1, 0, R) (q2, Y, L) - (q1, Y, R) -

q2 (q2, 0, L) - (q0, X, R) (q2, Y, L) -

q3 - - - (q3, Y, R) (q4, B, R) q4 - - - - -

* The TM basically matches up 0’s and 1’s
* q1 is the “scan right” state
* q2 is the “scan left” state
* q4 is the final state
* **Example #2:** {w | w is in {0,1}\* and w ends with a 0} 0

00

10

10110

Not ε

Q = {q0, q1, q2}

Γ = {0, 1, B}

Σ = {0, 1}

F = {q2}

0 1 B

q0 (q0, 0, R) (q0, 1, R) (q1, B, L)

q1 (q2, 0, R) - -

q2 - - -

* + q0 is the “scan right” state
  + q1 is the verify 0 state
* **Definition:** Let M = (Q, Σ, Г, δ, q0, B, F) be a TM, and let w be a string in Σ\*. Then w is

*accepted* by M iff

q0w |—\* α1pα2

Where p is in F and α1 and α2 are in Г\*

* **Definition:** Let M = (Q, Σ, Г, δ, q0, B, F) be a TM. The *language accepted by M*, denoted L(M), is the set
* Notes:

{w | w is in Σ\* and w is accepted by M}

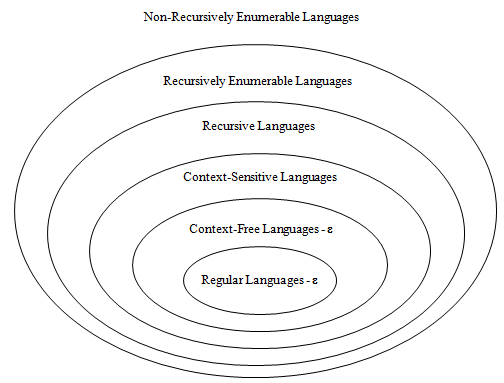
* In contrast to FA and PDAs, if a TM simply *passes through* a final state then the string is accepted.
* Given the above definition, no final state of an TM need have any exiting transitions.

*Henceforth, this is our assumption*.

* If x is not in L(M) then M may enter an infinite loop, or halt in a non-final state.
* Some TMs halt on all inputs, while others may not. In either case the language defined by TM is still well defined.
* **Definition:** Let L be a language. Then L is *recursively enumerable* if there exists a TM M such that L = L(M).
  + If L is r.e. then L = L(M) for some TM M, and
    - If x is in L then M halts in a final (accepting) state.
    - If x is not in L then M may halt in a non-final (non-accepting) state, or loop forever.
* **Definition:** Let L be a language. Then L is *recursive* if there exists a TM M such that L = L(M) and M halts on all inputs.
  + If L is recursive then L = L(M) for some TM M, and
    - If x is in L then M halts in a final (accepting) state.
    - If x is not in L then M halts a non-final (non-accepting) state.

Notes:

* + The set of all recursive languages is a subset of the set of all recursively enumerable languages
  + Terminology is easy to confuse: A *TM* is not recursive or recursively enumerable, rather a *language* is recursive or recursively enumerable.
* Recall the Hierarchy:



* **Observation:** Let L be an r.e. language. Then there is an infinite list M0, M1, … of TMs such that L = L(Mi).
* **Question:** Let L be a recursive language, and M0, M1, … a list of all TMs such that L = L(Mi), and choose any i>=0. Does Mi always halt?

**Answer:** Maybe, maybe not, but *at least one in the list does*.

* **Question:** Let L be a recursive enumerable language, and M0, M1, … a list of all TMs such that L = L(Mi), and choose any i>=0. Does Mi always halt?

**Answer:** Maybe, maybe not. Depending on L, none might halt or some may halt.

* + If L is also recursive then L is recursively enumerable.
* **Question:** Let L be a recursive enumerable language that is not recursive (L is in r.e. – r), and M0, M1, … a list of all TMs such that L = L(Mi), and choose any i>=0. Does Mi always halt?

**Answer:** No! If it did, then L would not be in r.e. – r, it would be recursive.

* Let M be a TM.
* Question: Is L(M) r.e.?

Answer: Yes! By definition it is!

* Question: Is L(M) recursive?

Answer: Don’t know, we don’t have enough information.

* Question: Is L(M) in r.e – r?

Answer: Don’t know, we don’t have enough information.

* Let M be a TM that halts on all inputs:
* Question: Is L(M) recursively enumerable? Answer: Yes! By definition it is!
* Question: Is L(M) recursive? Answer: Yes! By definition it is!
* Question: Is L(M) in r.e – r?

Answer: No! It can’t be. Since M always halts, L(M) is recursive.

* Let M be a TM.
* As noted previously, L(M) is recursively enumerable, but may or may not be recursive.
* Question: Suppose that L(M) is recursive. Does that mean that M always halts? Answer: Not necessarily. However, some TM M’ must exist such that L(M’) = L(M) and M’ always halts.
* Question: Suppose that L(M) is in r.e. – r. Does M always halt?

Answer: No! If it did then L(M) would be recursive and therefore not in r.e. – r.

* Let M be a TM, and suppose that M loops forever on some string x.
* Question: Is L(M) recursively enumerable? Answer: Yes! By definition it is.
* Question: Is L(M) recursive?

Answer: Don’t know. Although M doesn’t always halt, some other TM M’ may exist

such that L(M’) = L(M) and M’ always halts.

* Question: Is L(M) in r.e. – r? Answer: Don’t know.

Closure Properties for Recursive and Recursively Enumerable Languages

* **TMs Model General Purpose Computers:**
* If a TM can do it, so can a GP computer
* If a GP computer can do it, then so can a TM

*If you want to know if a TM can do X, then some equivalent question are:*

* *Can a general purpose computer do X?*
* *Can a C/C++/Java/etc. program be written to do X?*

*For example, is a language L recursive?*

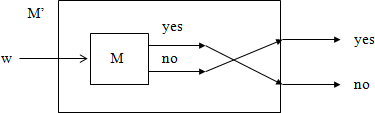
* *Can a C/C++/Java/etc. program be written that always halts and accepts L?*
* TM Block Diagrams:
* If L is a recursive language, then a TM M that accepts L and always halts can be

pictorially represented by a “chip” that has one input and two outputs.

* If L is a recursively enumerable language, then a TM M that accepts L can be pictorially represented by a “chip” that has one output.
* Conceivably, M could be provided with an output for “no,” but this output cannot be counted on. Consequently, we simply ignore it.
* **Theorem:** The recursive languages are closed with respect to complementation, i.e., if L is a recursive language, then so is

**Proof:** Let M be a TM such that L = L(M) and M always halts. Construct TM M’ as

follows:



* Note That:
  + M’ accepts iff M does not
  + M’ always halts since M always halts

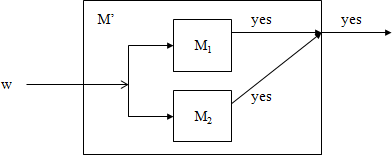
From this it follows that the complement of L is recursive. •

* **Theorem:** The recursive languages are closed with respect to union, i.e., if L1 and L2 are recursive languages, then so is

**Proof:** Let M1 and M2 be TMs such that L1 = L(M1) and L2 = L(M2) and M1 and M2 always halts. Construct TM M’ as follows:

* Note That:
* L(M’) = L(M1) U L(M2)
  + L(M’) is a subset of L(M1) U L(M2)
  + L(M1) U L(M2) is a subset of L(M’)
* M’ always halts since M1 and M2 always halt It follows from this that L3 = L1 U L2 is recursive.
* **Theorem:** The recursive enumerable languages are closed with respect to union, i.e., if L1 and L2 are recursively enumerable languages, then so is L3 = L1 U L2

**Proof:** Let M1 and M2 be TMs such that L1 = L(M1) and L2 = L(M2). Construct M’ as follows:



* Note That:

– L(M’) = L(M1) U L(M2)

* L(M’) is a subset of L(M1) U L(M2)
* L(M1) U L(M2) is a subset of L(M’)

– M’ halts and accepts iff M1 or M2 halts and accepts

It follows from this that is recursively enumerable.

The Halting Problem – Background

* **Definition:** A decision problem is a problem having a yes/no answer (that one presumably wants to solve with a computer). Typically, there is a list of parameters on which the problem is based.
* Given a list of numbers, is that list sorted?
* Given a number x, is x even?
* Given a C program, does that C program contain any syntax errors?
* Given a TM (or C program), does that TM contain an infinite loop?

From a practical perspective, many decision problems do not seem all that interesting. However, from a theoretical perspective they are for the following two reasons:

* Decision problems are more convenient/easier to work with when proving complexity results.
* Non-decision counter-parts are typically at least as difficult to solve.
* Notes:

– The following terms and phrases are analogous:

Algorithm - A halting TM program

Decision Problem - A language (un)Decidable - (non)Recursive

Statement of the Halting Problem

* **Practical Form:** (P1)

Input: Program P and input I. Question: Does P terminate on input I?

* Theoretical Form: (P2)

Input: Turing machine M with input alphabet Σ and string w in Σ\*. Question: Does M halt on w?

* A Related Problem We Will Consider First: (P3)

Input: Turing machine M with input alphabet Σ and one final state, and string w in Σ\*. Question: Is w in L(M)?

* Analogy:

Input: DFA M with input alphabet Σ and string w in Σ\*. Question: Is w in L(M)?

Is this problem decidable? Yes!

* Over-All Approach:
* We will show that a language *Ld* is not recursively enumerable
* From this it will follow that is not recursive
* Using this we will show that a language *Lu* is not recursive
* From this it will follow that the halting problem is undecidable.

The Universal Language

* Define the language Lu as follows:

Lu = {x | x is in {0, 1}\* and x = <M,w> where M is a TM encoding and w is in L(M)}

* Let x be in {0, 1}\*. Then either:

1. x doesn’t have a TM prefix, in which case x is **not** in Lu

1. x has a TM prefix, i.e., x = <M,w> and either:

a) w is not in L(M), in which case x is **not** in Lu

b) w is in L(M), in which case x is in Lu

* Compare P3 and Lu:

(P3):

Input: Turing machine M with input alphabet Σ and one final state, and string w in Σ\*.

* Notes:
* Lu is P3 expressed as a language
* Asking if Lu is recursive is the same as asking if P3 is decidable.
* We will show that Lu is not recursive, and from this it will follow that P3 is un- decidable.
* From this we can further show that the halting problem is un-decidable.
* Note that Lu is recursive if M is a DFA.

Church-Turing Thesis

* There is an effective procedure for solving a problem if and only if there is a TM that halts for all inputs and solves the problem.
* There are many other computing models, but all are equivalent to or subsumed by TMs.

*There is no more powerful machine* (Technically cannot be proved).

* DFAs and PDAs do not model all effective procedures or computable functions, but only a subset.
* If something can be “computed” it can be computed by a Turing machine.
* Note that this is called a ***Thesis***, not a theorem.
* It can’t be proved, because the term “can be computed” is too vague.
* But it is universally accepted as a true statement.
* Given the ***Church-Turing Thesis***:
  + What does this say about "computability"?
  + Are there things even a Turing machine can't do?
  + If there are, then there are things that simply can't be "computed."
    - Not with a Turing machine
    - Not with your laptop
    - Not with a supercomputer
  + There ARE things that a Turing machine can't do!!!
* The Church-Turing Thesis:
  + In other words, there is no problem for which we can describe an algorithm that can’t be done by a Turing machine.

The Universal Turing machine

* If Tm’s are so damned powerful, can’t we build one that simulates the behavior of any Tm on any tape that it is given?
* Yes. This machine is called the ***Universal Turing machine***.
* How would we build a Universal Turing machine?
  + We place an encoding of any Turing machine on the input tape of the Universal Tm.
  + The tape consists entirely of zeros and ones (and, of course, blanks)
  + Any Tm is represented by zeros and ones, using unary notation for elements and zeros as separators.
* Every Tm instruction consists of four parts, each a represented as a series of **1**'s and separated by **0**'s.
* Instructions are separated by **00**.
* We use unary notation to represent components of an instruction, with
* 0 = **1**,
* 1 = **11**,
* 2 = **111**,
* 3 = **1111**,
* ***n*** = **111...111** (***n***+1 1's).
* We encode ***qn*** as *n* + 1 1's
* We encode symbol ***an*** as *n* + 1 1's
* We encode move left as 1, and move right as 11 1111011101111101110100101101101101100

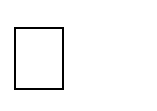
q3, a2, q4, a2, L q0, a1, q1, a1, R

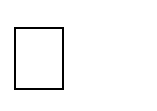
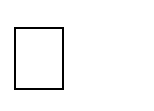
* Any Turing machine can be encoded as a unique long string of zeros and ones, beginning with a **1**.
* Let ***Tn*** be the Turing machine whose encoding is the number ***n****.*

Linear Bounded Automata

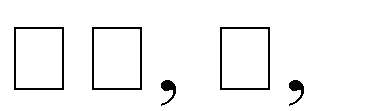
* A Turing machine that has the length of its tape limited to the length of the input string is called a linear-bounded automaton (LBA).
* A linear bounded automaton is a 7-tuple *nondeterministic* Turing machine M = (Q, S, G, d, q0,qaccept, qreject) except that:
  1. There are two extra tape symbols < and >, which are not elements of G.
  2. The TM begins in the configuration (q0<*x*>), with its tape head scanning the symbol < in cell 0. The > symbol is in the cell immediately to the right of the input string *x*.
  3. The TM cannot replace < or > with anything else, nor move the tape head left of < or right of >.

Context-Sensitivity

* *Context-sensitive production* any production satisfying | |  |



|.

* *Context-sensitive grammar* any generative grammar *G* =  , production in  context-sensitive.





* No empty productions.

such that every

Context-Sensitive Language

* Language *L context-sensitive* if there exists context-sensitive grammar *G* such that either

*L* = *L*(*G*) or *L* = *L*(*G*)  { }.

* Example:

The language L = {anbncn : n  1} is a C.S.L. the grammar is S  abc/ aAbc,

Ab  bA, AC  Bbcc, bB  Bb,

aB  aa/ aaA

The derivation tree of a3b3c3 is looking to be as following S ⇒ aAbc

⇒ abAc

⇒ abBbcc

⇒ aBbbcc ⇒ aaAbbcc

⇒ aabAbcc

⇒ aabbAcc ⇒ aabbBbccc

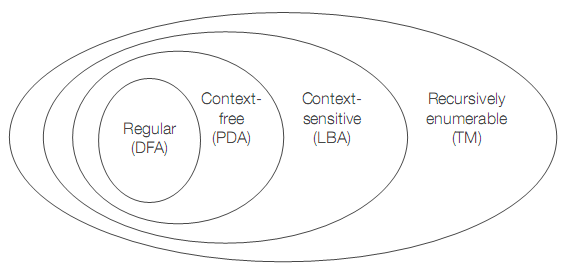
⇒ aabBbbccc

⇒ aaBbbbccc

⇒ aaabbbccc

CSG = LBA

* A language is accepted by an LBA iff it is generated by a CSG.
* Just like equivalence between CFG and PDA
* Given an x  CSG G, you can intuitively see that and LBA can start with S, and nondeterministically choose all derivations from S and see if they are equal to the input string x. Because CSL’s are non-contracting, the LBA only needs to generate derivations of length  |x|. This is because if it generates a derivation longer than |x|, it will never be able to shrink to the size of |x|.



UNIT V

**Chomsky Hierarchy of Languages**

* + A containment hierarchy (strictly nested sets) of classes of formal grammars

The Hierarchy

**Class Grammars Languages Automaton**

Type-0 Unrestricted Recursively enumerable Turing machine (Turing-recognizable)

none Recursive Decider

(Turing-decidable)

Type-1 Context-sensitive Context-sensitive Linear-bounded

|  |  |  |  |
| --- | --- | --- | --- |
| Type-2 | Context-free | Context-free | Pushdown |
| Type-3 | Regular | Regular | Finite |

*Type 0 Unrestricted*:

Languages defined by Type-0 grammars are accepted by Turing machines .

Rules are of the form: *α* → *β*, where *α* and *β* are arbitrary strings over a vocabulary *V* and

*α ≠ ε*

*Type 1 Context-sensitive:*

Languages defined by Type-1 grammars are accepted by linear-bounded automata. Syntax of some natural languages (Germanic)

*Rules are of the form:*

*αAβ → αBβ S → ε*

*where*

*A, S ∈ N*

*α, β, B ∈ (N ⋃ Σ)∗*

*B ≠ ε*

Type 2 Context-free:

Languages defined by Type-2 grammars are accepted by push-down automata. Natural language is almost entirely definable by type-2 tree structures

Rules are of the form:

*A* → *α*

Where

*A* ∈ *N*

*α* ∈ (*N* ⋃ Σ)∗

Type 3 Regular:

Languages defined by Type-3 grammars are accepted by finite state automata Most syntax of some informal spoken dialog

Rules are of the form:

*A* → *ε A* → *α A* → *αB*

where

*A, B* ∈ *N* and *α* ∈ Σ

The Universal Turing Machine

* If Tm’s are so damned powerful, can’t we build one that simulates the behavior of any Tm on any tape that it is given?
* Yes. This machine is called the ***Universal Turing machine***.
* How would we build a Universal Turing machine?
  + We place an encoding of any Turing machine on the input tape of the Universal Tm.
  + The tape consists entirely of zeros and ones (and, of course, blanks)
  + Any Tm is represented by zeros and ones, using unary notation for elements and zeros as separators.
* Every Tm instruction consists of four parts, each a represented as a series of **1**'s and separated by **0**'s.
* Instructions are separated by **00**.
* We use unary notation to represent components of an instruction, with
  + 0 = **1**,
  + 1 = **11**,
  + 2 = **111**,
  + 3 = **1111**,
  + ***n*** = **111...111** (***n***+1 1's).
* We encode ***qn*** as *n* + 1 1's
* We encode symbol ***an*** as *n* + 1 1's
* We encode move left as 1, and move right as 11

1111011101111101110100101101101101100

q3, a2, q4, a2, L q0, a1, q1, a1, R

* Any Turing machine can be encoded as a unique long string of zeros and ones, beginning with a **1**.
* Let ***Tn*** be the Turing machine whose encoding is the number ***n****.*

Turing Reducibility

* A language A is Turing reducible to a language B, written A T B, if A is decidable relative to B
* Below it is shown that ETM is Turing reducible to EQTM
* Whenever A is mapping reducible to B, then A is Turing reducible to B

– The function in the mapping reducibility could be replaced by an oracle

* An oracle Turing machine with an oracle for EQTM can decide ETM

TEQ-TM = “On input <M>

1. Create TM M1 such that L(M1) = 

M1 has a transition from start state to reject state for every element of 

1. Call the EQTM oracle on input <M,M2>
2. If it accepts, accept; if it rejects, reject”

* TEQ-TM decides ETM
* ETM is decidable relative to EQTM
* Applications
* If A T B and B is decidable, then A is decidable
* If A T B and A is undecidable, then B is undecidable
* If A T B and B is Turing-recognizable, then A is Turing-recognizable
* If A T B and A is non-Turing-recognizable, then B is non-Turing-recognizable

The class P

A decision problem *D* is *solvable in polynomial time* or *in the class P,* if there exists an algorithm *A* such that

* *A Takes instances* of *D* as inputs.
* *A* always outputs the correct answer “Yes” or “No”.
* There exists a polynomial *p* such that the execution of *A* on inputs of size *n* always terminates in *p(n)* or fewer steps.
* **EXAMPLE**: The Minimum Spanning Tree Problem is in the class P.

The class *P* is often considered as synonymous with the class of computationally feasible problems, although in practice this is somewhat unrealistic.

The class NP

A decision problem is *nondeterministically polynomial-time solvable* or *in the class NP* if there exists an algorithm *A* such that

* *A* takes as inputs potential witnesses for “yes” answers to problem *D.*
* *A* correctly distinguishes true witnesses from false witnesses.
* There exists a polynomial *p* such that for each potential witnesses of each instance of size *n* of *D,* the execution of the algorithm *A* takes at most *p(n)* steps.
* Think of a non-deterministic computer as a computer that magically “guesses” a solution, then has to verify that it is correct
  + If a solution exists, computer always guesses it
  + One way to imagine it: a parallel computer that can freely spawn an infinite number of processes
    - Have one processor work on each possible solution
    - All processors attempt to verify that their solution works
    - If a processor finds it has a working solution
  + So: **NP** = problems *verifiable* in polynomial time
  + Unknown whether **P = NP** (most suspect not)

NP-Complete Problems

* We will see that NP-Complete problems are the “hardest” problems in NP:
  + If any *one* NP-Complete problem can be solved in polynomial time.
  + Then *every* NP-Complete problem can be solved in polynomial time.
  + And in fact *every* problem in **NP** can be solved in polynomial time (which would show **P = NP**)
  + Thus: solve hamiltonian-cycle in O(*n*100) time, you’ve proved that **P = NP**. Retire rich & famous.
* The crux of NP-Completeness is *reducibility*
  + Informally, a problem P can be reduced to another problem Q if *any* instance of P can be “easily rephrased” as an instance of Q, the solution to which provides a solution to the instance of P
    - *What do you suppose “easily” means?*
    - This rephrasing is called *transformation*
  + Intuitively: If P reduces to Q, P is “no harder to solve” than Q
* An example:
  + P: Given a set of Booleans, is at least one TRUE?
  + Q: Given a set of integers, is their sum positive?
  + Transformation: (x1, x2, …, x*n*) = (y1, y2, …, y*n*) where y*i* = 1 if x*i* = TRUE, y*i* = 0 if x*i* = FALSE
* Another example:
  + Solving linear equations is reducible to solving quadratic equations
    - *How can we easily use a quadratic-equation solver to solve linear equations?*
* Given one NP-Complete problem, we can prove many interesting problems NP-Complete
  + Graph coloring (= register allocation)
  + Hamiltonian cycle
  + Hamiltonian path
  + Knapsack problem
  + Traveling salesman
  + Job scheduling with penalties, etc.

NP Hard

* **Definition:** Optimization problems whose decision versions are NP- complete are called *NP-hard.*
* **Theorem:** If there exists a polynomial-time algorithm for finding the optimum in any *NP-*hard problem, then *P = NP.*

**Proof:** Let *E* be an *NP-*hard optimization (let us say minimization) problem, and let *A* be a polynomial-time algorithm for solving it. Now an instance *J* of the corresponding decision problem *D* is of the form *(I, c),* where *I* is an instance of *E,* and *c* is a number. Then the answer to *D* for instance *J* can be obtained by running *A* on *I* and checking whether the cost of the optimal solution exceeds *c.* Thus there exists a polynomial-time algorithm for *D,* and *NP-*completeness of the latter implies *P= NP.*