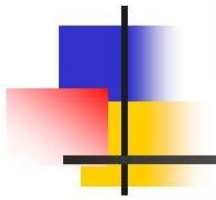
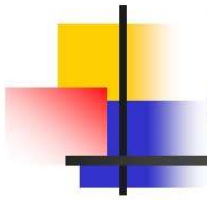
FORMAL LANGUAGES AND AUTOMATA THEORY

UNIT 2



Regular Expressions

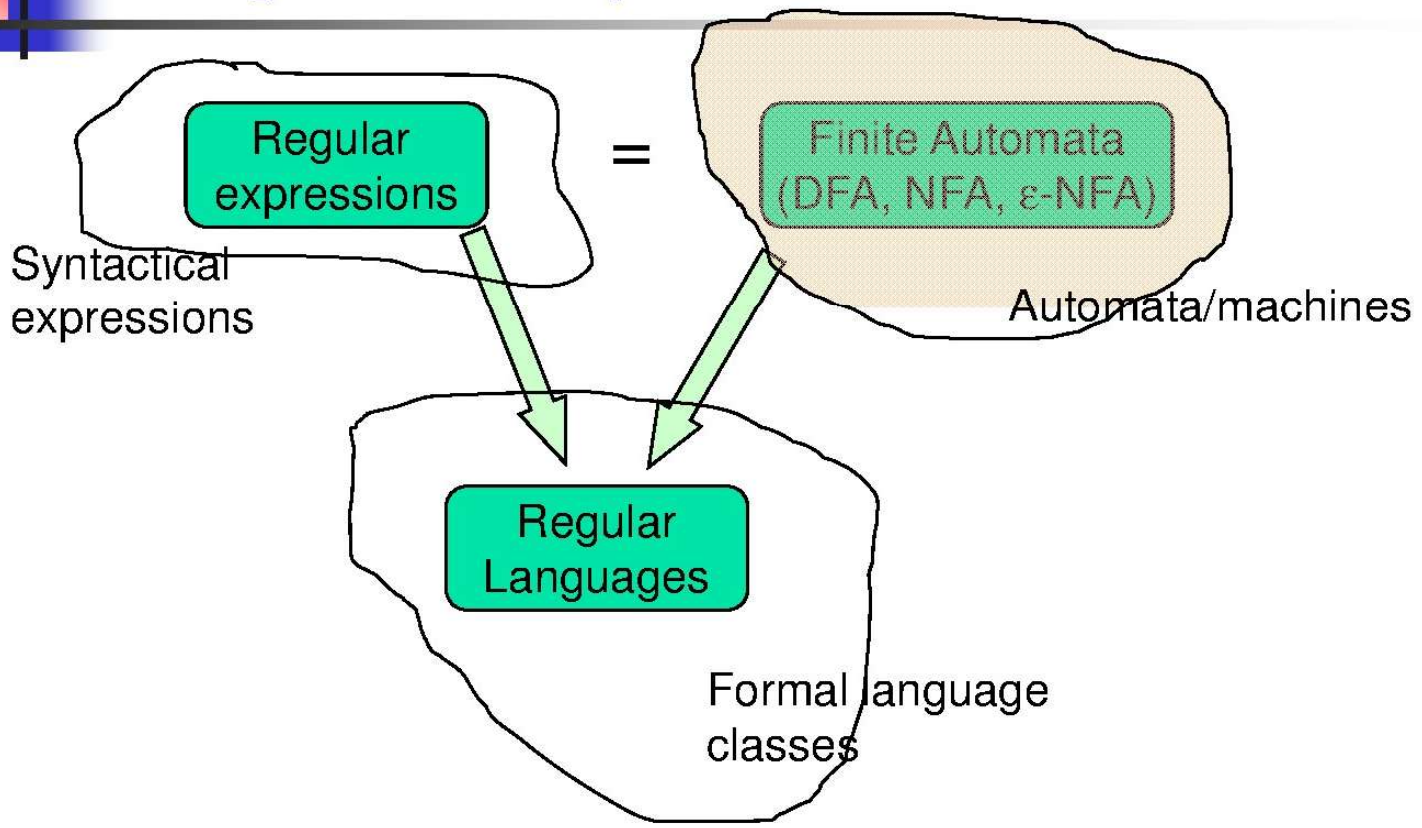


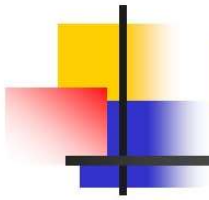
Regular Expressions vs. Finite Automata

- Offers a declarative way to express the pattern of any string we want to accept
  - E.g., 01*+ 10*
- Automata => more machine-like

  < input: string , output: [accept/reject] >
- Regular expressions => more program syntax-like
- Unix environments heavily use regular expressions
  - E.g., bash shell, grep, vi & other editors, sed
- Perl scripting – good for string processing
- Lexical analyzers such as Lex or Flex

Regular Expressions





Language Operators

- Union of two languages:
  - **L U M** = all strings that are either in L or M
  - Note: A union of two languages produces a third language

- Concatenation of two languages:
  - **L . M** = all strings that are of the form *xy* s.t., x ∈ L and y ∈ M
  - The *dot* operator is usually omitted
    - i.e., **LM** is same as L.M

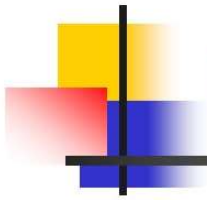
"i" here refers to how many strings to concatenate from the parent language L to produce strings in the language L^i

Kleene Closure (the * operator)

- Kleene Closure of a given language L:
  - $L^0 = \{\varepsilon\}$
  - $L^1 = \{w \mid \text{for some } w \in L\}$
  - $L^2 = \{ w_1w_2 \mid w_1 \in L, w_2 \in L \text{ (duplicates allowed)}\}$
  - $L^i = \{ w_1w_2\ldots w_i \mid \text{all w's chosen are} \in L \text{ (duplicates allowed)}\}$
  - (Note: the choice of each w_i is independent)
  - $L^* = \bigcup_{i\geq 0} L^i$ (arbitrary number of concatenations)

Example:

- Let L = { 1, 00}
  - $L^0 = \{\varepsilon\}$
  - $L^1 = \{1,00\}$
  - $L^2 = \{11,100,001,0000\}$
  - $L^3 = \{111,1100,1001,10000,000000,00001,00100,0011\}$
  - $L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$

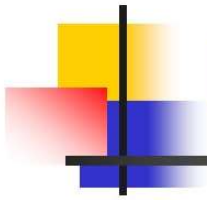


Kleene Closure (special notes)

- L^* is an infinite set iff $|L| \geq 1$ and $L \neq \{\varepsilon\}$
- If $L=\{\varepsilon\}$, then $L^* = \{\varepsilon\}$
- If $L = \Phi$, then $L^* = \{\varepsilon\}$

Σ^* denotes the set of all words over an alphabet Σ

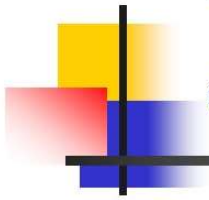
- Therefore, an abbreviated way of saying there is an arbitrary language L over an alphabet Σ is:
  - $L \subseteq \Sigma^*$



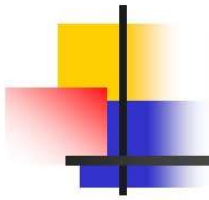
Building Regular Expressions

- Let E be a regular expression and the language represented by E is L(E)
- Then:
  - (E) = E
  - L(E + F) = L(E) U L(F)
  - L(E F) = L(E) L(F)
  - L(E*) = (L(E))*

Example: how to use these regular expression properties and language operators?



- L = { w | w is a binary string which does *not* contain two consecutive 0s or two consecutive 1s anywhere)
  - E.g., w = 01010101 is in L, while w = 10010 is not in L
- Goal: Build a regular expression for L
- Four cases for w:
  - Case A: w starts with 0 and |w| is even
  - Case B: w starts with 1 and |w| is even
  - Case C: w starts with 0 and |w| is odd
  - Case D: w starts with 1 and |w| is odd
- Regular expression for the four cases:
  - Case A: (01)*
  - Case B: (10)*
  - Case C: 0(10)*
  - Case D: 1(01)*
- Since L is the union of all 4 cases:
  - Reg Exp for L = (01)* + (10)* + 0(10)* + 1(01)*
- If we introduce ε then the regular expression can be simplified to:
  - Reg Exp for L = (ε +1)(01)*(ε +0)



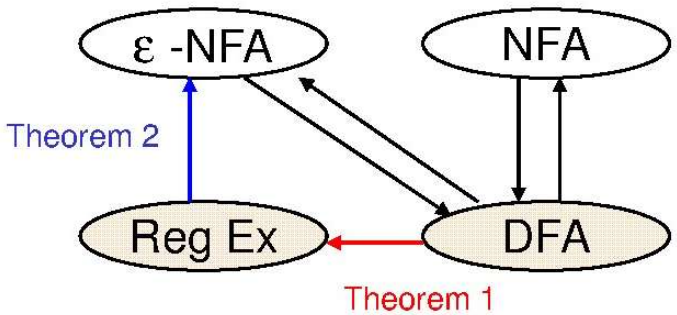
Precedence of Operators

- Highest to lowest
  - * operator (star)
  - . (concatenation)
  - + operator
- Example:
  - 01* + 1 = (0 . ((1)*)) + 1

Finite Automata (FA) & Regular Expressions (Reg Ex)

- To show that they are interchangeable, consider the following theorems:
  - *Theorem 1: For every DFA A there exists a regular expression R such that L(R)=L(A)*
  - *Theorem 2: For every regular expression R there exists an ε -NFA E such that L(E)=L(R)*

Proofs in the book

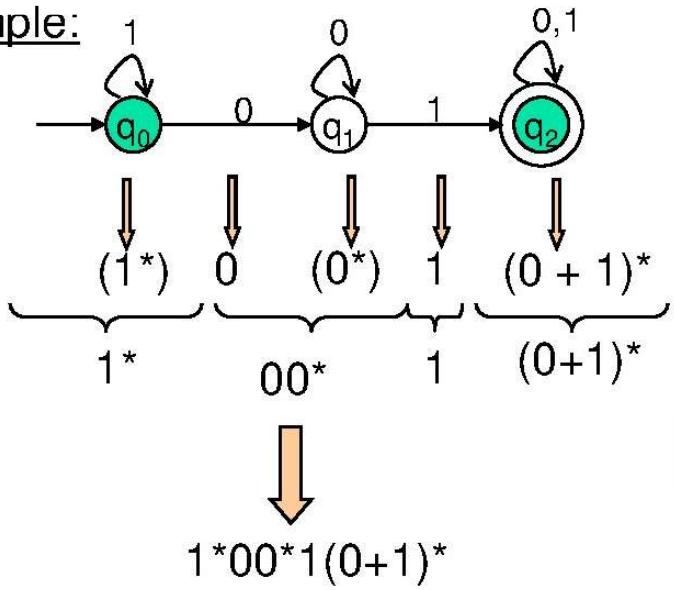


Kleene Theorem

DFA to RE construction

Informally, trace all distinct paths (traversing cycles only once)
from the start state to *each of the* final states
and enumerate all the expressions along the way

Example:

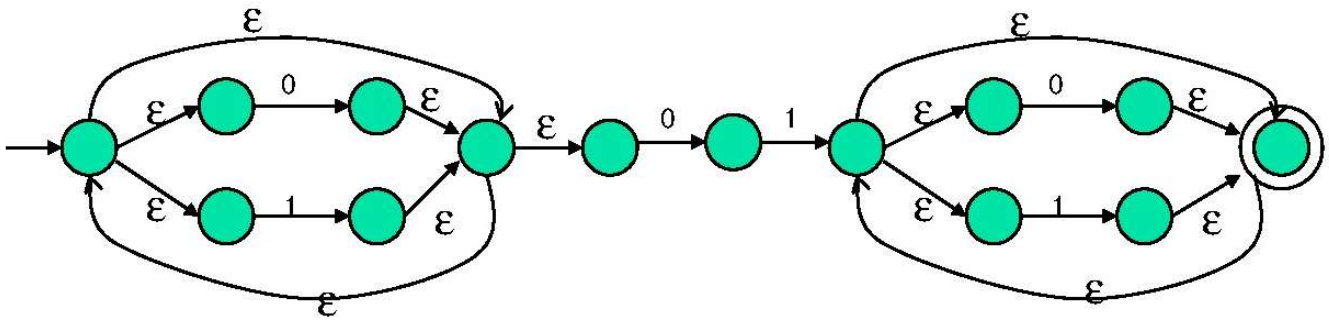


Q) What is the language?

RE to ε-NFA construction

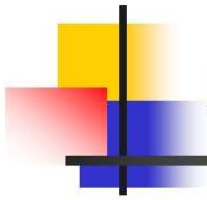
Example: (0+1)*01(0+1)*

(0+1)* 01 (0+1)*



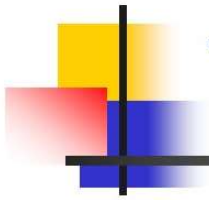
Algebraic Laws of Regular Expressions

- Commutative:
  - $E+F = F+E$
- Associative:
  - $(E+F)+G = E+(F+G)$
  - $(EF)G = E(FG)$
- Identity:
  - $E+\Phi = E$
  - $\varepsilon E = E \varepsilon = E$
- Annihilator:
  - $\Phi E = E\Phi = \Phi$



Algebraic Laws…

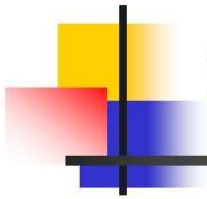
- Distributive:
  - $E(F+G) = EF + EG$
  - $(F+G)E = FE+GE$
- Idempotent: $E + E = E$
- Involving Kleene closures:
  - $(E^*)^* = E^*$
  - $\Phi^* = \varepsilon$
  - $\varepsilon^* = \varepsilon$
  - $E^+ = EE^*$
  - $E? = \varepsilon + E$



True or False?

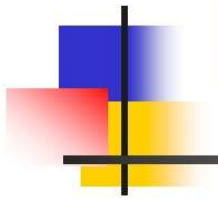
Let R and S be two regular expressions. Then:

1. $((R^*)^*)^* = R^*$?
2. $(R+S)^* = R^* + S^*$?
3. $(RS + R)^* RS = (RR^*S)^*$?

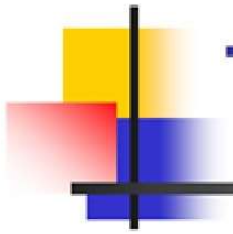


Summary

- Regular expressions
- Equivalence to finite automata
- DFA to regular expression conversion
- Regular expression to ε-NFA conversion
- Algebraic laws of regular expressions
- Unix regular expressions and Lexical Analyzer



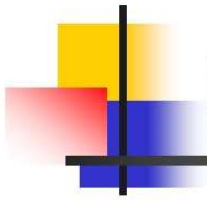
Properties of Regular Languages



Topics

1) How to prove whether a given language is regular or not?

2) Closure properties of regular languages



Some languages are *not* regular

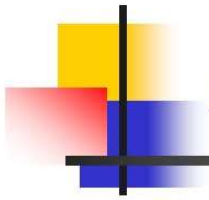
When is a language is regular?

if we are able to construct one of the following: DFA *or* NFA *or* ε -NFA *or* regular expression

When is it not?

If we can show that no FA can be built for a language

How to prove languages are ***not*** regular?



What if we cannot come up with any FA?

A) Can it be language that is not regular?

B) Or is it that we tried wrong approaches?

How do we *decisively* prove that a language is not regular?

"The hardest thing of all is to find a black cat in a dark room, especially if there is no cat!" -Confucius

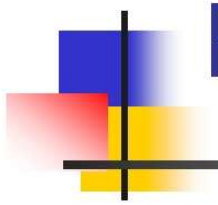
Example of a non-regular language

Let L = {w | w is of the form 0^n1^n , for all n≥0}

- *Hypothesis: L is not regular*
- Intuitive rationale: How do you keep track of a running count in an FA?
- A more formal rationale:
  - By contradition, if L is regular then there should exist a DFA for L.
  - Let k = number of states in that DFA.
  - Consider the special word w= 0^k1^k => w ∈ L
  - DFA is in some state p_i, after consuming the first i symbols in w

Rationale…

- Let $\{p_0, p_1, \ldots p_k\}$ be the sequence of states that the DFA should have visited after consuming the first k symbols in w which is 0^k
- But there are only k states in the DFA!
- ==> at least one state should repeat somewhere along the path (by  Principle)
- ==> Let the repeating state be $p_i = p_J$ for $i < j$
- ==> We can fool the DFA by inputing $0^{(k-(j-i))}1^k$ and still get it to accept (note: k-(j-i) is at most k-1).
- ==> DFA accepts strings w/ unequal number of 0s and 1s, implying that the DFA is wrong!



The Pumping Lemma for Regular Languages

A technique that is used to show that a given language is not regular

Pumping Lemma for Regular Languages

Let L be a regular language

Then *there exists* some constant ***N*** such that *for every* string $w \in L$ s.t. $|w| \geq N$, *there exists* a way to break w into three parts, $w=xyz$, such that:

1. $y \neq \varepsilon$
2. $|xy| \leq N$
3. For all $k \geq 0$, all strings of the form $xy^kz \in L$

This clause should hold for all regular languages.

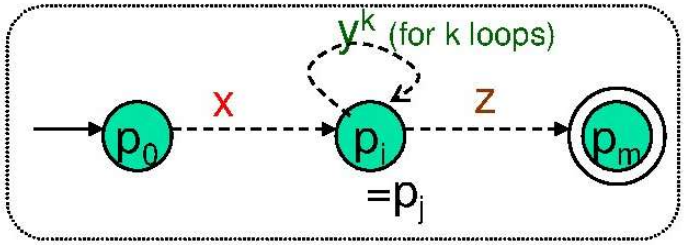
Definition: N is called the "Pumping Lemma Constant"

Pumping Lemma: Proof

- L is regular => it should have a DFA.
  - Set N := number of states in the DFA
- Any string $w \in L$, s.t. $|w| \geq N$, should have the form: $w=a_1a_2\ldots a_m$, where $m \geq N$
- Let the states traversed after reading the first N symbols be: $\{p_0, p_1, \ldots p_N\}$
  - ==> There are N+1 p-states, while there are only N DFA states
  - ==> at least one state has to repeat
    i.e, $p_i = p_J$ where $0 \leq i < j \leq N$ (by PHP)

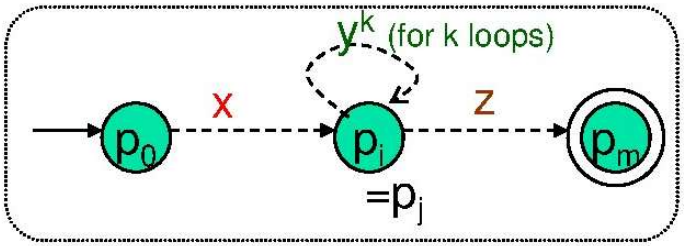
Pumping Lemma: Proof…

- => We should be able to break w=xyz as follows:
  - $x=a_1a_2..a_i$; $y=a_{i+1}a_{i+2}..a_J$; $z=a_{J+1}a_{J+2}..a_m$
  - x's path will be $p_0..p_i$
  - y's path will be $p_i\, p_{i+1}..p_J$ (but $p_i=p_J$ implying a loop)
  - z's path will be $p_Jp_{J+1}..p_m$
- Now consider another string $w_k=xy^kz$, where k≥0
- Case k=0
  - DFA will reach the accept state p_m
- Case k>0
  - DFA will loop for y^k, and finally reach the accept state p_m for z
- In either case, $w_k \in L$ This proves part (3) of the lemma



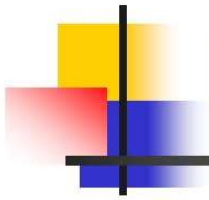
Pumping Lemma: Proof…

- For part (1):
  - Since i<j, y ≠ ε



- For part (2):
  - By PHP, the repetition of states has to occur within the first N symbols in w
  - ==> |xy|≤N

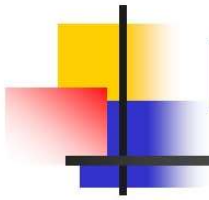
□



The Purpose of the Pumping Lemma for RL

- To prove that some languages *cannot be* regular.

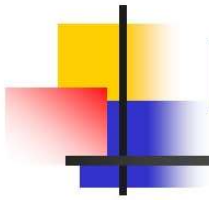
How to use the pumping lemma?



Think of playing a 2 person game

- Role 1: ***You*** claim that the language cannot be regular
- Role 2: An ***adversary*** who claims the language is regular
- You show that the adversary's statement will lead to a contradiction that implyies pumping lemma *cannot* hold for the language.
- You win!!

How to use the pumping lemma? (The Steps)



1. (you) L is not regular.
2. (adv.) Claims that L is regular and gives you a value for N as its P/L constant
3. (you) Using N, choose a string $w \in L$ s.t.,
   1. $|w| \geq N$,
   2. Using w as the template, construct other words w_k of the form xy^kz and show that at least one such $w_k \notin L$

      => this implies you have successfully broken the pumping lemma for the language, and hence that the adversary is wrong.

   (Note: In this process, you may have to try many values of k, starting with k=0, and then 2, 3, .. so on, until $w_k \notin L$)

Note: This N can be anything (need not necessarily be the #states in the DFA. It's the adversary's choice.)

Example of using the Pumping Lemma to prove that a language is not regular

Let L_{eq} = {w | w is a binary string with equal number of 1s and 0s}

- Your Claim: L_{eq} is not regular
- Proof:
  - By contradiction, let L_{eq} be regular → adv.
  - P/L constant should exist → adv.
    - Let *N* = that P/L constant
  - Consider input $w = 0^N1^N$ → you
    (your choice for the template string)
  - By pumping lemma, we should be able to break w=xyz, such that: →you
    1) $y \neq \varepsilon$
    2) $|xy| \leq N$
    3) For all $k \geq 0$, the string xy^kz is also in L

Template string $w = 0^N1^N$ = 00 …. 011 … 1 (N 0s, N 1s)

Proof…

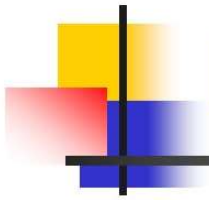
- Because $|xy| \leq N$, xy should contain only 0s
  - (This and because $y \neq \varepsilon$, implies $y=0^+$)
- Therefore x can contain *at most* N-1 0s
- Also, all the N 1s must be inside z
- By (3), any string of the form $xy^kz \in L_{eq}$ for all $k \geq 0$
- Case k=0: xz has at most N-1 0s but has N 1s
- Therefore, $xy^0z \notin L_{eq}$
- This violates the P/L (a contradiction)

→ you

Setting k=0 is referred to as "**pumping down**"

Setting k>1 is referred to as "**pumping up**"

Another way of proving this will be to show that if the #0s is arbitrarily pumped up (e.g., k=2), then the #0s will become exceed the #1s



Exercise 2

Prove L = {$0^n 10^n$ | n≥ 1} is not regular

Note: This n is not to be confused with the pumping lemma constant N. That *can* be different.

In other words, the above question is same as proving:

- L = {$0^m 10^m$ | m≥ 1} is not regular

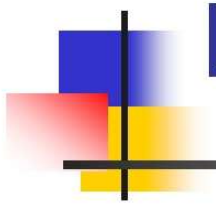
Example 3: Pumping Lemma

Claim: L = { 0^i | i is a perfect square} is not regular

- Proof:
  - By contradiction, let L be regular.
  - P/L should apply
  - Let *N* = P/L constant
  - Choose $w=0^{N^2}$
  - By pumping lemma, w=xyz satisfying all three rules
  - By rules (1) & (2), y has between 1 and N 0s
  - By rule (3), any string of the form xy^kz is also in L for all $k\geq 0$
  - Case k=0:
    - #zeros (xy^0z) = #zeros (xyz) - #zeros (y)
    - $N^2 - N \leq$ #zeros (xy^0z) $\leq N^2 - 1$
    - $\mathbf{(N-1)^2} < N^2 - N \leq$ #zeros (xy^0z) $\leq N^2 - 1 < \mathbf{N^2}$
    - $xy^0z \notin L$
    - But the above will complete the proof ONLY IF N>1.
    - … (proof contd.. Next slide)

Example 3: Pumping Lemma

- (proof contd…)
  - If the adversary pick N=1, then $\mathbf{(N-1)^2} \leq N^2 - N$, and therefore the #zeros(xy^0z) could end up being a perfect square!
  - This means that pumping down (i.e., setting k=0) is not giving us the proof!
  - So lets try pumping up next…
- Case k=2:
  - #zeros (xy^2z) = #zeros (xyz) + #zeros (y)
  - $N^2 + 1 \leq$ #zeros (xy^2z) $\leq N^2 + N$
  - $\mathbf{N^2} < N^2 + 1 \leq$ #zeros (xy^2z) $\leq N^2 + N < \mathbf{(N+1)^2}$
  - $xy^2z \notin L$ 
  - (Notice that the above should hold for all possible N values of N>0. Therefore, this completes the proof.)

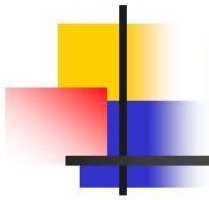


Closure properties of Regular Languages

Closure properties for Regular Languages (RL)

- *Closure property:*
  - If a set of regular languages are combined using an operator, then the resulting language is also regular
- Regular languages are *closed* under:
  - Union, intersection, complement, difference
  - Reversal
  - Kleene closure
  - Concatenation
  - Homomorphism
  - Inverse homomorphism

Now, lets prove all of this!



RLs are closed under union

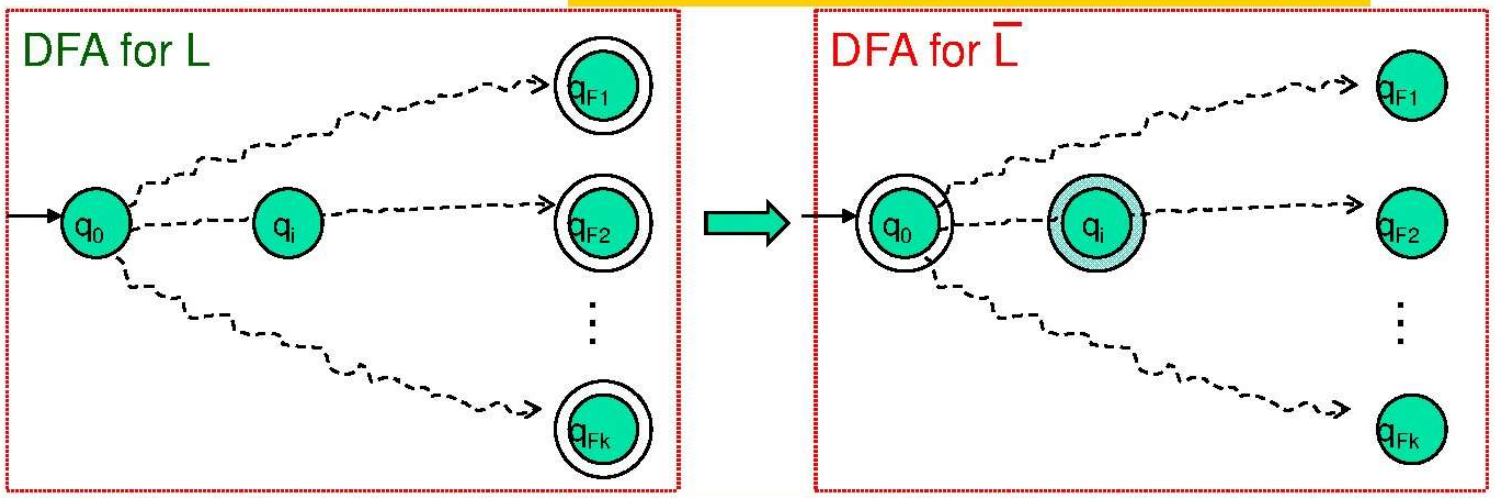
- IF L and M are two RLs THEN:
  - they both have two corresponding regular expressions, R and S respectively
  - (L U M) can be represented using the regular expression R+S
  - Therefore, (L U M) is also regular □

How can this be proved using FAs?

RLs are closed under complementation

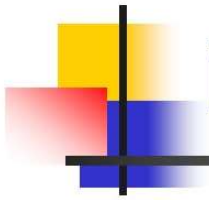
- If L is an RL over Σ, then $\overline{L}=\Sigma^*-L$
  - To show $\overline{L}$ is also regular, make the following construction

Convert every final state into non-final, and every non-final state into a final state

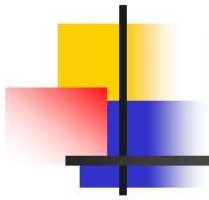


Assumes q0 is a non-final state. If not, do the opposite.

RLs are closed under intersection



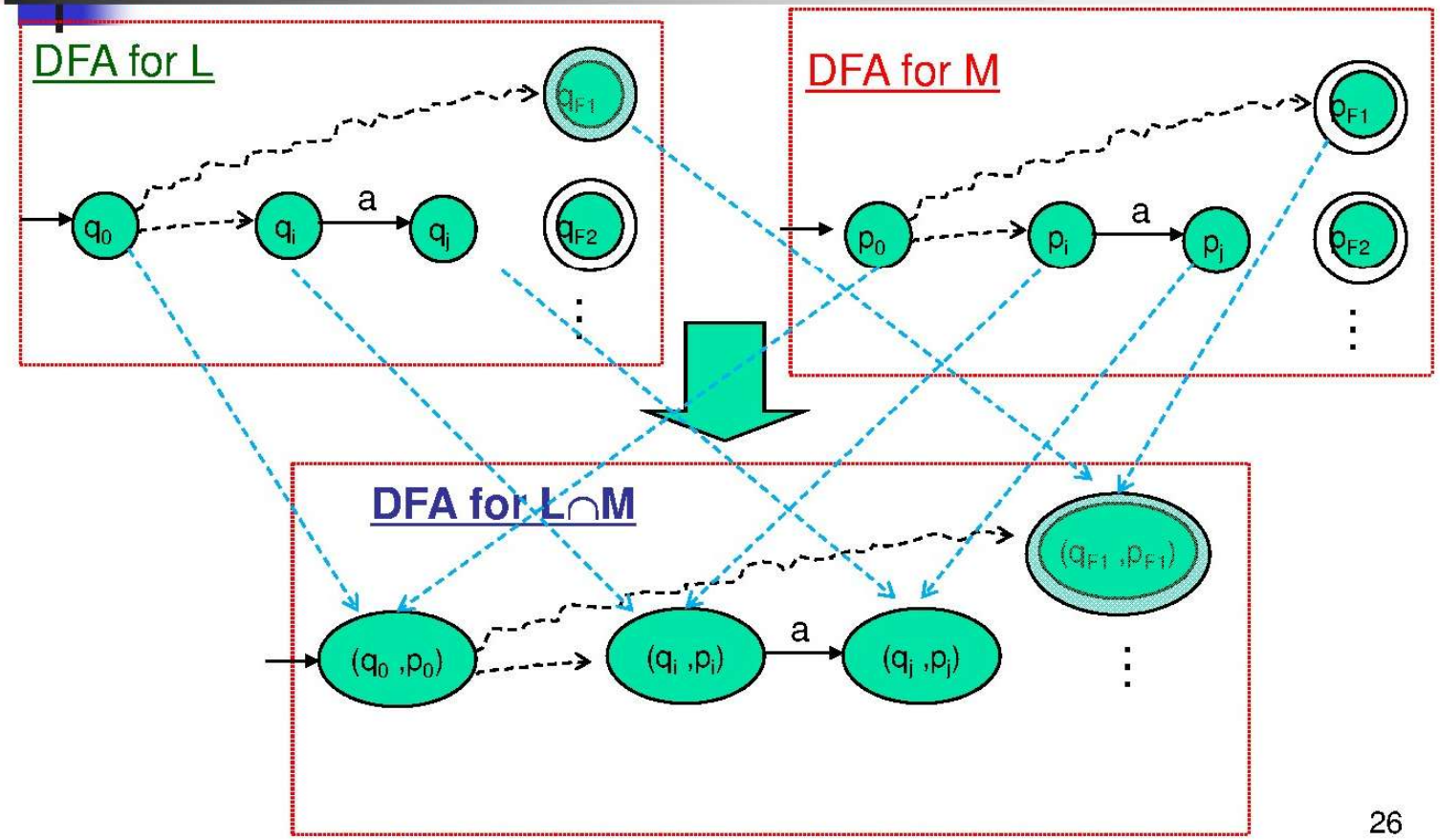
- A quick, indirect way to prove:
  - By DeMorgan's law:
  - $L \cap M = \overline{(\bar{L} \cup \bar{M})}$
  - Since we know RLs are closed under union and complementation, they are also closed under intersection
- A more direct way would be construct a finite automaton for $L \cap M$



DFA construction for L ∩ M

- A_L = DFA for L = $\{Q_L, \Sigma, q_L, F_L, \delta_L\}$
- A_M = DFA for M = $\{Q_M, \Sigma, q_M, F_M, \delta_M\}$
- Build $A_{L \cap M} = \{Q_L \times Q_M, \Sigma, (q_L, q_M), F_L \times F_M, \delta\}$ such that:
  - $\delta((p,q),a) = (\delta_L(p,a), \delta_M(q,a))$, where p in Q_L, and q in Q_M
- This construction ensures that a string w will be accepted if and only if w reaches an accepting state in <u>both</u> input DFAs.

DFA construction for L ∩ M



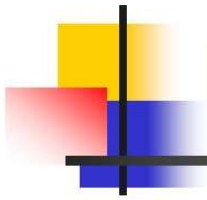
RLs are closed under set difference

- We observe:
  - $L - M = L \cap \overline{M}$

Closed under intersection

Closed under complementation

- Therefore, L - M is also regular



RLs are closed under reversal

Reversal of a string w is denoted by w^R

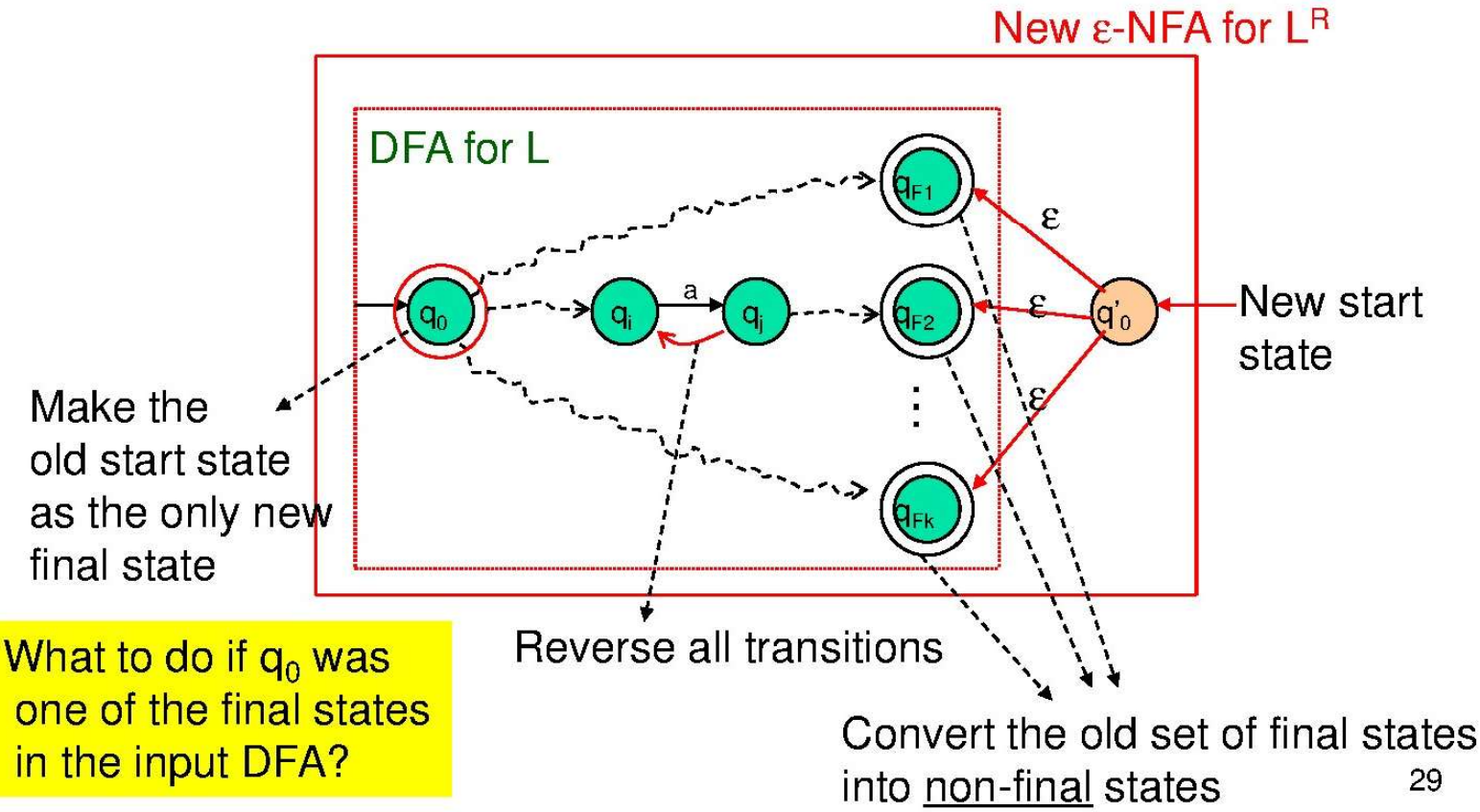
- E.g., w=00111, w^R=11100

Reversal of a language:

- L^R = The language generated by reversing all strings in L

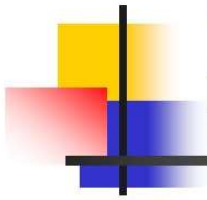
Theorem: If L is regular then L^R is also regular

ε -NFA Construction for L^R

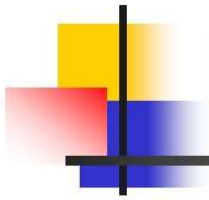


What to do if q_0 was one of the final states in the input DFA?

If L is regular, L^R is regular (proof using regular expressions)

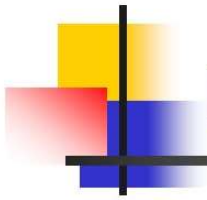


- Let E be a regular expression for L
- Given E, how to build E^R?
- Basis: If E= ε, Ø, or a, then E^R=E
- Induction: Every part of E (refer to the part as "F") can be in only *one* of the three following forms:
  1. $F = F_1+F_2$
     - $F^R = F_1^R+F_2^R$
  2. $F = F_1F_2$
     - $F^R = F_2^RF_1^R$
  3. $F = (F_1)^*$
     - $(F^R)^* = (F_1^R)^*$



Homomorphisms

- Substitute each symbol in ∑ (main alphabet) by a corresponding string in T (another alphabet)
  - h: ∑--->T*
- Example:
  - Let ∑={0,1} and T={a,b}
  - Let a homomorphic function h on ∑ be:
    - h(0)=ab, h(1)=ε
  - If w=10110, then h(w) = εabεεab = abab
- In general,
  - $h(w) = h(a_1)\, h(a_2) \ldots h(a_n)$



RLs are closed under homomorphisms

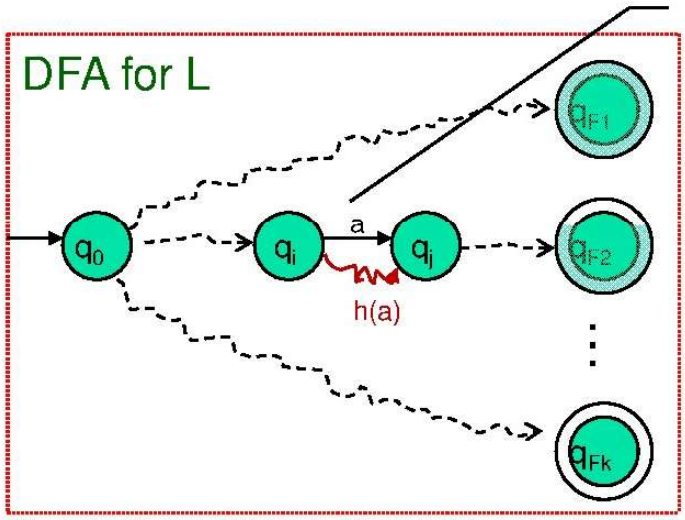
- Theorem: If L is regular, then so is h(L)
- Proof: If E is a RE for L, then show L(h(E)) = h(L(E))
- Basis: If E= ε, Ø, or a, then the claim holds.
- Induction: There are three forms of E:
  1. $E = E_1+E_2$
     - $L(h(E)) = L(h(E_1) + h(E_2)) = L(h(E_1)) \cup L(h(E_2))$ ----- (1)
     - $h(L(E)) = h(L(E_1) + L(E_2)) = h(L(E_1)) \cup h(L(E_2))$ ----- (2)
     - By inductive hypothesis, $L(h(E_1))= h(L(E_1))$ and $L(h(E_2))= h(L(E_2))$
     - Therefore, $L(h(E)= h(L(E)$
  2. $E = E_1E_2$
  3. $E = (E_1)^*$

  } Similar argument

Think of a DFA based construction

Given a DFA for L, how to convert it into an FA for h(L)?

FA Construction for h(L)

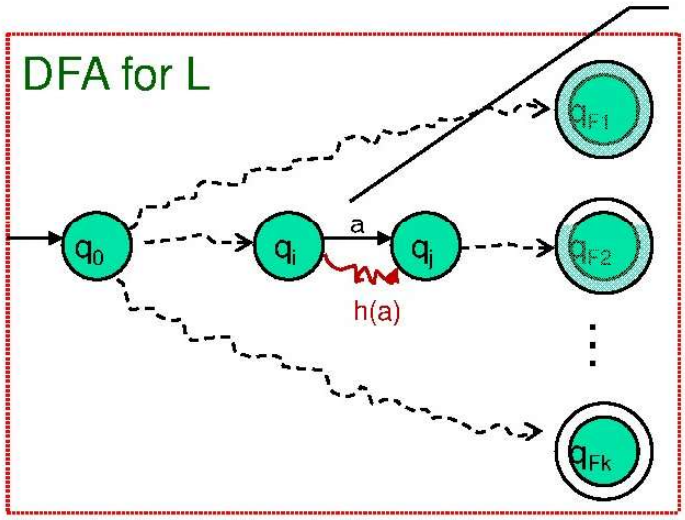


Replace every edge "a" by a path labeled h(a) in the new DFA

- Build a new FA that simulates h(a) for every symbol a transition in the above DFA
- The resulting FA (may or may not be a will be for h(L)

Given a DFA for L, how to convert it into an FA for h(L)?

FA Construction for h(L)



Replace every edge "a" by a path labeled h(a) in the new DFA

- Build a new FA that simulates h(a) for every symbol a transition in the above DFA
- The resulting FA may or may not be a DFA, but will be a FA for h(L)

Given a DFA for M, how to convert it into an FA for $h^{-1}(M)$?

The set of strings in Σ^* whose homomorphic translation results in the strings of M

Inverse homomorphism

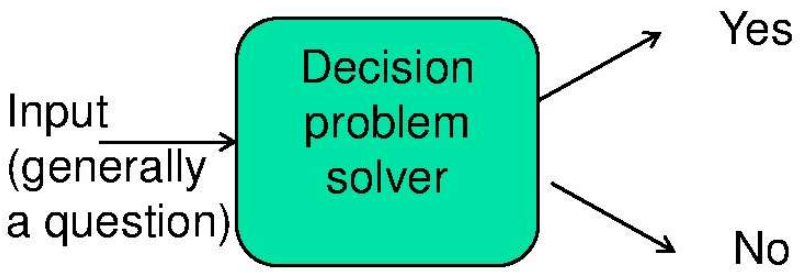
- Let h: Σ--->T*
- Let M be a language over alphabet T
- $h^{-1}(M) = \{w \mid w \in \Sigma^* \text{ s.t., } h(w) \in M\}$

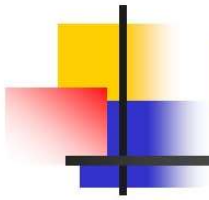
Claim: If M is regular, then so is $h^{-1}(M)$

- Proof:
  - Let A be a DFA for M
  - Construct another DFA A' which encodes $h^{-1}(M)$
  - A' is an exact replica of A, except that its transition functions are s.t. for any input symbol *a* in Σ, A' will simulate *h(a)* in A.
    - $\delta(p,a) = \hat{\delta}(p,h(a))$

Decision properties of regular languages

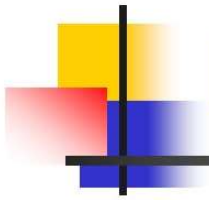
Any "decision problem" looks like this:





Membership question

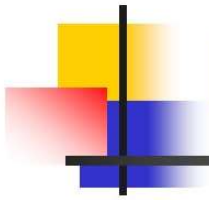
- Decision Problem: Given L, is w in L?
- Possible answers: Yes or No
- Approach:
  1. Build a DFA for L
  2. Input w to the DFA
  3. If the DFA ends in an accepting state, then yes; otherwise no.



Emptiness test

- Decision Problem: Is L=Ø ?
- Approach:
  1. Build a DFA for L
  2. From the start state, run a *reachability* test, which returns:
     1. success: if there is at least one final state that is reachable from the start state
     2. failure: otherwise
  3. L=Ø if and only if the reachability test fails

How to implement the reachability test?



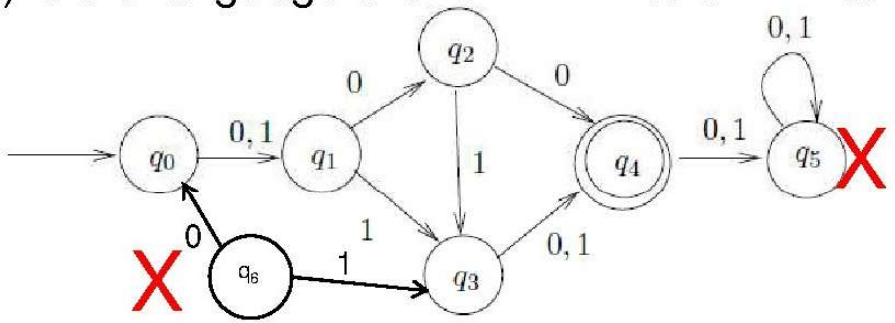
Finiteness

- Decision Problem: Is L finite or infinite?
- Approach:
  1. Build DFA for L
  2. Remove all states unreachable from the start state
  3. Remove all states that cannot lead to any accepting state.
  4. After removal, check for cycles in the resulting FA
  5. L is finite if there are no cycles; otherwise it is infinite
- Another approach
  - Build a regular expression and look for Kleene closure

How to implement steps 2 and 3?

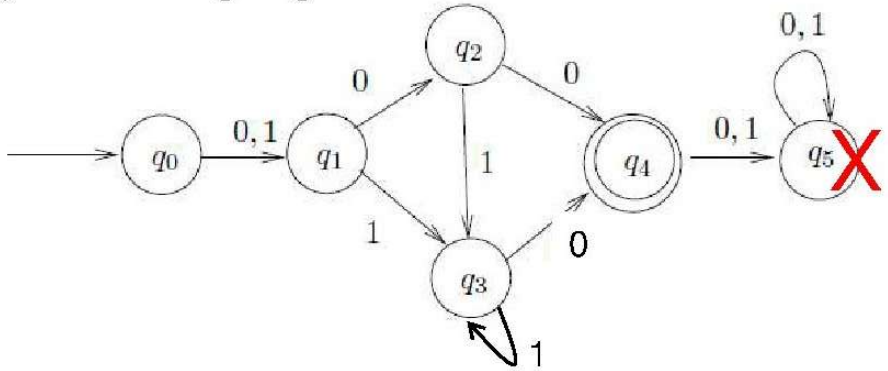
Finiteness test - examples

Ex 1) Is the language of this DFA finite or infinite?



FINITE

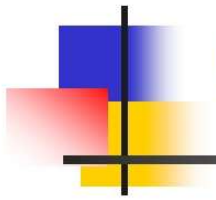
Ex 2) Is the language of this DFA finite or infinite?



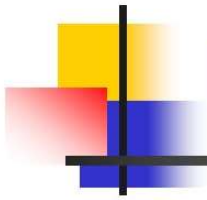
INFINITE

Summary

- How to prove languages are not regular?
  - Pumping lemma & its applications
- Closure properties of regular languages

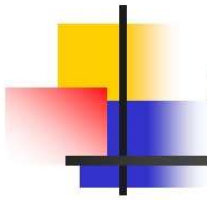


Context-Free Languages & Grammars (CFLs & CFGs)



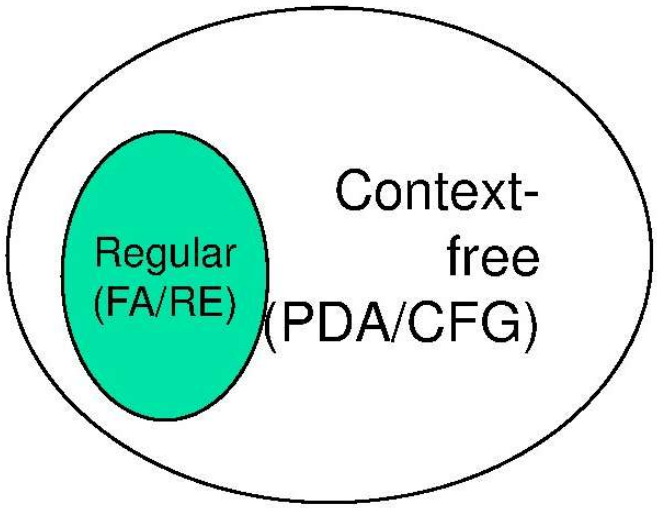
Not all languages are regular

- So what happens to the languages which are not regular?
- Can we still come up with a language recognizer?
  - i.e., something that will accept (or reject) strings that belong (or do not belong) to the language?



Context-Free Languages

- A language class larger than the class of regular languages
- Supports natural, recursive notation called "context-free grammar"
- Applications:
  - Parse trees, compilers
  - XML



An Example

- A palindrome is a word that reads identical from both ends
  - E.g., madam, redivider, malayalam, 010010010
- Let L = { w | w is a binary palindrome}
- Is L regular?
  - No.
  - Proof:
    - Let $w=0^N10^N$ (assuming N to be the p/l constant)
    - By Pumping lemma, w can be rewritten as xyz, such that xy^kz is also L (for any $k\geq 0$)
    - But $|xy|\leq N$ and $y\neq\varepsilon$
    - ==> $y=0^+$
    - ==> xy^kz *will NOT* be in L for k=0
    - ==> Contradiction

But the language of palindromes…

is a CFL, because it supports recursive substitution (in the form of a CFG)

- This is because we can construct a "*grammar*" like this:

1. A ==> ε
2. A ==> 0
3. A ==> 1
4. A ==> 0A0
5. A ==> 1A1

Terminal

Variable or non-terminal

Productions

Same as:
A => 0A0 | 1A1 | 0 | 1 | ε

How does this grammar work?

How does the CFG for palindromes work?

An input string belongs to the language (i.e., accepted) iff it can be generated by the CFG

G:
A => 0A0 | 1A1 | 0 | 1 | ε

- Example: w=01110
- G can generate w as follows:

1. A => 0A0
2. => 01A10
3. => 01110

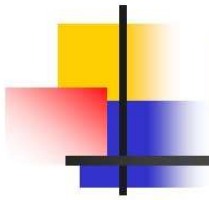
Generating a string from a grammar:

1. Pick and choose a sequence of productions that would allow us to generate the string.
2. At every step, substitute one variable with one of its productions.

Context-Free Grammar: Definition

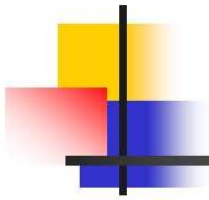
- A context-free grammar G=(V,T,P,S), where:
  - V: set of variables or non-terminals
  - T: set of terminals (= alphabet U {ε})
  - P: set of *productions,* each of which is of the form
    V ==> α_1 | α_2 | …
    - Where each α_i is an arbitrary string of variables and terminals
  - S ==> start variable

CFG for the language of binary palindromes:
G=({A},{0,1},P,A)
P: A ==> 0 A 0 | 1 A 1 | 0 | 1 | ε



More examples

- Parenthesis matching in code
- Syntax checking
- In scenarios where there is a general need for:
  - Matching a symbol with another symbol, or
  - Matching a count of one symbol with that of another symbol, or
  - Recursively substituting one symbol with a string of other symbols



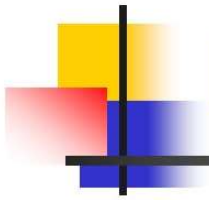
Example #2

- Language of balanced paranthesis

  e.g., ()((((()))))((()))….
- CFG?

G:
S => (S) | SS | ε

How would you "interpret" the string "((((()))()())" using this grammar?

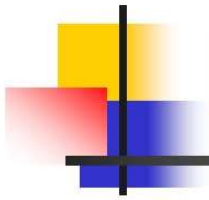


Example #3

- A grammar for L = $\{0^m 1^n \mid m \geq n\}$
- CFG?

G:
S => 0S1 | A
A => 0A | ε

How would you interpret the string "00000111" using this grammar?



Example #4

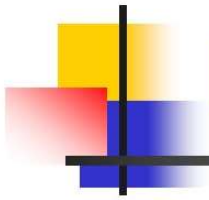
A program containing **if-then(-else)** statements

if *Condition* **then** *Statement* **else** *Statement*

(Or)

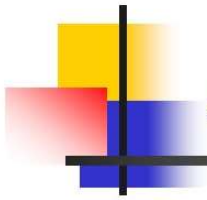
if *Condition* **then** *Statement*

CFG?



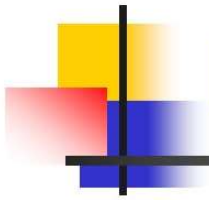
More examples

- $L_1 = \{0^n \mid n \geq 0\}$
- $L_2 = \{0^n \mid n \geq 1\}$
- $L_3 = \{0^i 1^j 2^k \mid i=j \text{ or } j=k, \text{ where } i,j,k \geq 0\}$
- $L_4 = \{0^i 1^j 2^k \mid i=j \text{ or } i=k, \text{ where } i,j,k \geq 1\}$



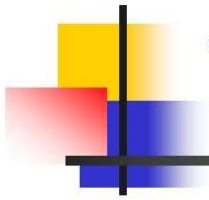
Applications of CFLs & CFGs

- Compilers use parsers for syntactic checking
- Parsers can be expressed as CFGs
  1. Balancing paranthesis:
     - B ==> BB | (B) | *Statement*
     - *Statement ==> …*
  2. If-then-else:
     - S ==> SS | *if Condition then Statement else Statement* | *if Condition then Statement* | *Statement*
     - *Condition ==> …*
     - *Statement ==> …*
  3. C paranthesis matching { … }
  4. Pascal *begin-end* matching
  5. YACC (Yet Another Compiler-Compiler)



More applications

- Markup languages
  - Nested Tag Matching
    - HTML
      - <html> … … … … </html>
    - XML
      - <PC> … <MODEL> … </MODEL> .. <RAM> … </RAM> … </PC>



Tag-Markup Languages

Roll ==> <ROLL> Class Students </ROLL>

Class ==> <CLASS> Text </CLASS>

Text ==> Char Text | Char

Char ==> a | b | … | z | A | B | .. | Z

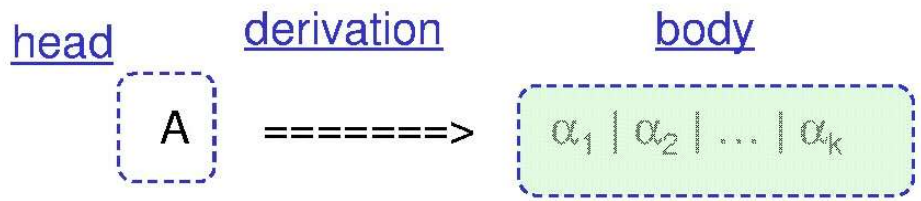
Students ==> Student Students | ε

Student ==> <STUD> Text </STUD>

Here, the left hand side of each production denotes one non-terminals (e.g., “Roll”, “Class”, etc.)

Those symbols on the right hand side for which no productions (i.e., substitutions) are defined are terminals (e.g., ‘a’, ‘b’, ‘|’, ‘<‘, ‘>’, “ROLL”, etc.)

Structure of a production

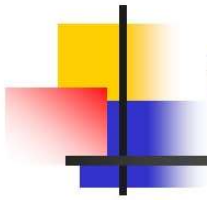


The above is same as:

1. A ==> α_1
2. A ==> α_2
3. A ==> α_3

…

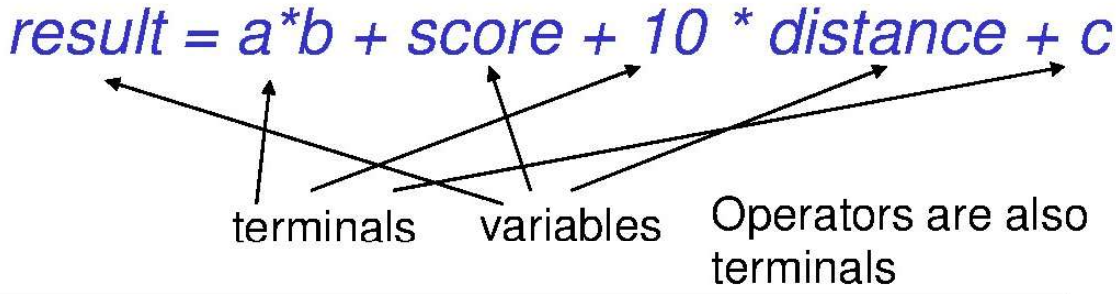
K. A ==> α_k



CFG conventions

- Terminal symbols <== a, b, c…
- Non-terminal symbols <== A,B,C, …
- Terminal or non-terminal symbols <== X,Y,Z
- Terminal strings <== w, x, y, z
- Arbitrary strings of terminals and non-terminals <== α, β, γ, ..

Syntactic Expressions in Programming Languages



Regular languages have only terminals

- Reg expression = [a-z][a-z0-1]*
- If we allow only letters a & b, and 0 & 1 for constants (for simplification)
  - Regular expression = (a+b)(a+b+0+1)*

String membership

How to say if a string belong to the language defined by a CFG?

1. Derivation
   - Head to body
2. Recursive inference
   - Body to head

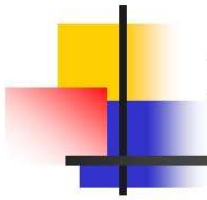
Both are equivalent forms

<u>Example:</u>

- w = 01110
- Is w a palindrome?

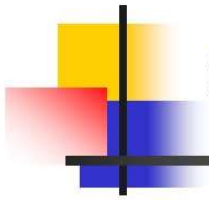
<u>G:</u>
A => 0A0 | 1A1 | 0 | 1 | ε

A => 0A0
=> 01A10
=> 01110



Simple Expressions…

- We can write a CFG for accepting simple expressions
- G = (V,T,P,S)
  - V = {E,F}
  - T = {0,1,a,b,+,*,(,)}
  - S = {E}
  - P:
    - E ==> E+E | E*E | (E) | F
    - F ==> aF | bF | 0F | 1F | a | b | 0 | 1



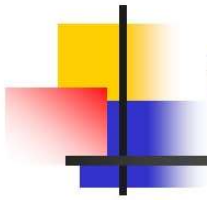
Generalization of derivation

- Derivation is *head ==> body*

- A==>X (A derives X in a single step)
- A ==>*$_G$ X (A derives X in a multiple steps)

- <u>Transitivity:</u>

  IFA ==>*$_G$B, and B ==>*$_G$C, THEN A ==>*$_G$ C



Context-Free Language

- The language of a CFG, G=(V,T,P,S), denoted by L(G), is the set of terminal strings that have a derivation from the start variable S.
  - $L(G) = \{ w \text{ in } T^* \mid S \Rightarrow^*_G w \}$

Left-most & Right-most Derivation Styles

G:
E => E+E | E*E | (E) | F
F => aF | bF | 0F | 1F | ε

Derive the string a*(ab+10) from G: $E =^{*}\Rightarrow_{G} a*(ab+10)$

Left-most derivation:

Always substitute leftmost variable

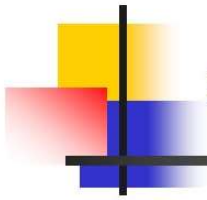
- E
- ==> E * E
- ==> F * E
- ==> a * E
- ==> a * (E)
- ==> a * (E + E)
- ==> a * (F + E)
- ==> a * (aF + E)
- ==> a * (abF + E)
- ==> a * (ab + E)
- ==> a * (ab + F)
- ==> a * (ab + 1F)
- ==> a * (ab + 10F)
- ==> a * (ab + 10)

Right-most derivation:

Always substitute rightmost variable

- E
- ==> E * E
- ==> E * (E)
- ==> E * (E + E)
- ==> E * (E + F)
- ==> E * (E + 1F)
- ==> E * (E + 10F)
- ==> E * (E + 10)
- ==> E * (F + 10)
- ==> E * (aF + 10)
- ==> E * (abF + 0)
- ==> E * (ab + 10)
- ==> F * (ab + 10)
- ==> aF * (ab + 10)
- ==> a * (ab + 10)

Leftmost vs. Rightmost derivations



Q1) For every leftmost derivation, there is a rightmost derivation, and vice versa. True or False?

True - will use parse trees to prove this

Q2) Does every word generated by a CFG have a leftmost and a rightmost derivation?

Yes – easy to prove (reverse direction)

Q3) Could there be words which have more than one leftmost (or rightmost) derivation?

Yes – depending on the grammar

How to prove that your CFGs are correct?

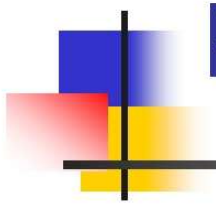
(using induction)

CFG & CFL

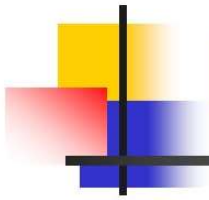
G_{pal}:
A => 0A0 | 1A1 | 0 | 1 | ε

- Theorem: A string w in (0+1)* is in $L(G_{pal})$, if and only if, w is a palindrome.

- Proof:
  - Use induction
    - on string length for the IF part
    - On length of derivation for the ONLY IF part



Parse trees



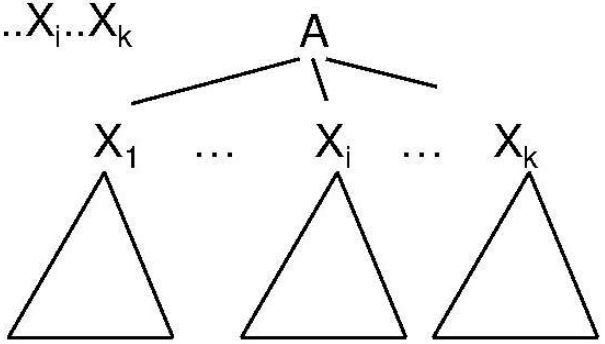
Parse Trees

- Each CFG can be represented using a *parse tree:*
  - Each internal node is labeled by a variable in V
  - Each leaf is terminal symbol
  - For a production, $A ==> X_1 X_2 \ldots X_k$, then any internal node labeled A has k children which are labeled from $X_1, X_2, \ldots X_k$ from left to right

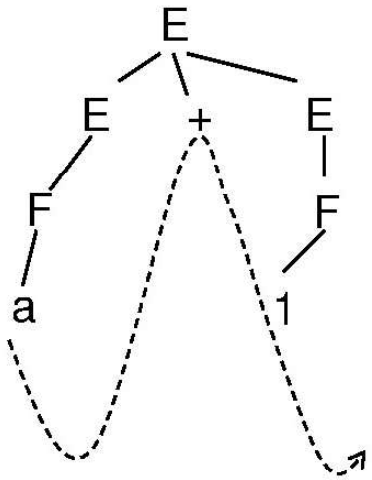
---

Parse tree for production and all other subsequent productions:

$A ==> X_1 .. X_i .. X_k$



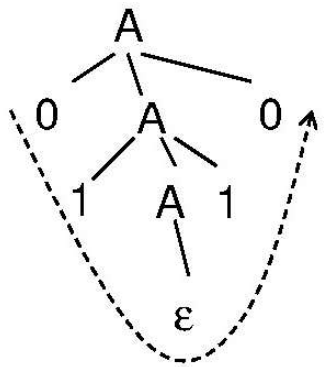
Examples



Parse tree for *a + 1*

G:
E => E+E | E*E | (E) | F
F => aF | bF | 0F | 1F | 0 | 1 | a | b

Recursive inference

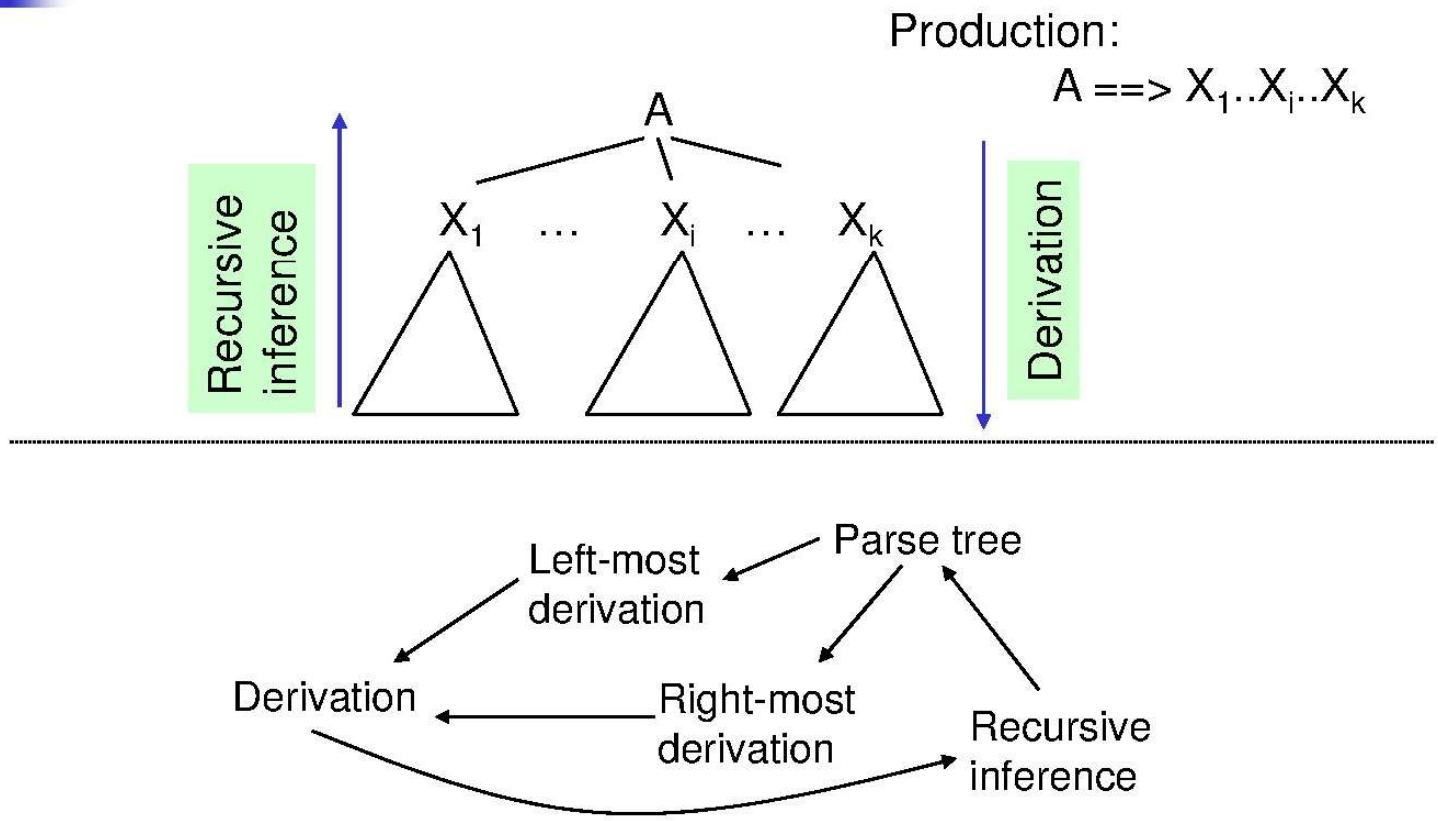


Derivation

Parse tree for *0110*

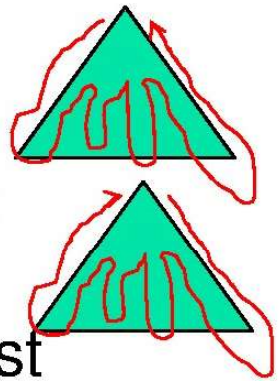
G:
A => 0A0 | 1A1 | 0 | 1 | ε

Parse Trees, Derivations, and Recursive Inferences



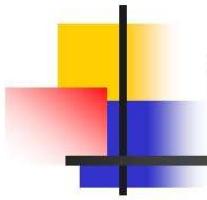
Interchangeability of different CFG representations

- Parse tree ==> left-most derivation
  - DFS left to right
- Parse tree ==> right-most derivation
  - DFS right to left
- ==> left-most derivation == right-most derivation
- Derivation ==> Recursive inference
  - Reverse the order of productions
- Recursive inference ==> Parse trees
  - bottom-up traversal of parse tree



Connection between CFLs and RLs

What kind of grammars result for regular languages?



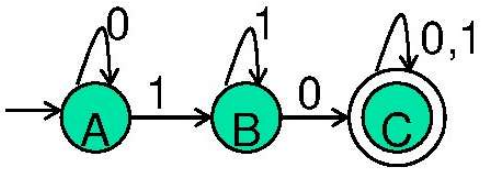
CFLs & Regular Languages

- A CFG is said to be *right-linear* if all the productions are one of the following two forms: *A ==> wB (or) A ==> w*

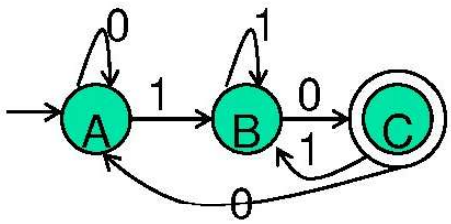
  Where:
  - A & B are variables,
  - w is a string of terminals

- Theorem 1: Every right-linear CFG generates a regular language
- Theorem 2: Every regular language has a right-linear grammar
- Theorem 3: Left-linear CFGs also represent RLs

Some Examples



Right linear CFG?



Right linear CFG?

- A => 01B | C
  B => 11B | 0C | 1A
  C => 1A | 0 | 1

Finite Automaton?

Summary

- Context-free grammars
- Context-free languages
- Productions, derivations, recursive inference, parse trees
- Left-most & right-most derivations