CSc 135 Course overview
Computing Theory and Programming Languages

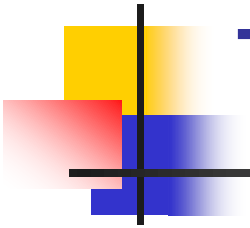
Course overview
Three basic concepts
Set theory: a brief review
Languages: set notation and operations

Course Goals

Course Goals

To provide students with:

1. familiarity with the theoretical foundations of Computer Science.
2. facility with the concepts, notations, and techniques of the theories of automata, formal languages, and Turing machines.
3. understanding of selected programming language features and their implementation.
4. experience using and implementing a recursive-descent parser.
5. experience writing programs using functional and logic language paradigms.



Tests and Exam

- **3 tests** – 25 - 40 min. each, focus on computing theory and key concepts in programming languages

- **exam** – combining Computing Theory (CT) and Programming Languages (PL)

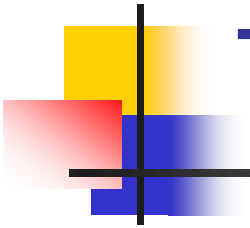
CT Home works and PL Assignments

- **Computing Theory Home works**
  - Each CT homework will be discussed and presented in class activity session
- **Programming Language Assignments**
  - Each PL assignment contains a set problems on:
    - (1) Parsing and Recursive-descent parser algorithm
    - (2) functional programming
    - (3) logic programming
    - (4) Web programming Recursive-descent parser
  - **Subset of CT hw and key concepts of PL assignments will be included in tests/exam**

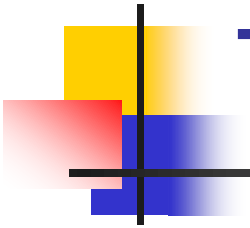
Flipped Classroom Idea -- Students do the learning

- **Students are responsible to:** read textbook, lecture notes, and other course related materials, HW
  - 1 required CT exercises presentation
  - Attend class regularly
  - Participate group discussion
- **Instructor will support** class activities with:
  - Tests
  - Study Guideline and examples solutions of problems
  - Lecture and notes

Computing theory: Three Major Concepts - 1

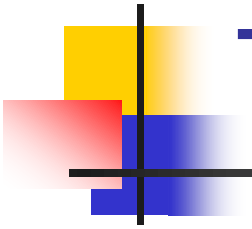


- Languages
  - 4 formal languages
- Language generators
  - 4 grammars (language generators) for 4 languages
- Languages recognizers
  - 4 automata (language recognizers) for 4 languages



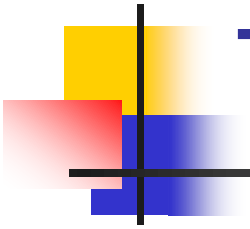
Three Major Concepts - 2

- 4 formal languages
  - Communication vehicle between man and machine
    - Regular Language (RL)
    - Context-Free Language (CFL)
    - Context-Sensitive Language (CSL)
    - Recursively Enumerable Language (REL)



Three Major Concepts - 3

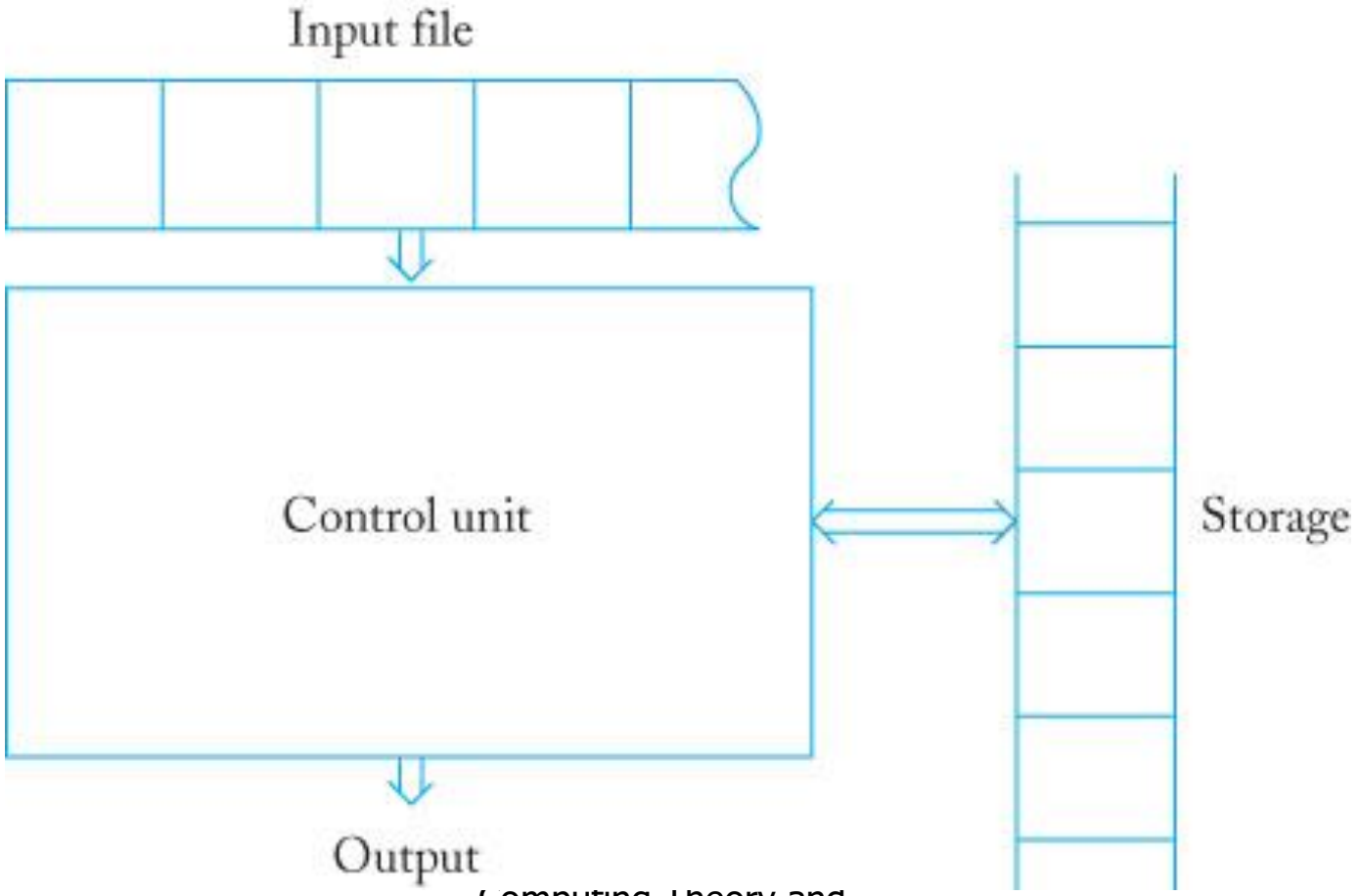
- Language generators – 4 Grammars for 4 languages
  - RG for RL
  - CFG for CFL
  - CSG for CSL
  - Unrestricted Grammar for REL

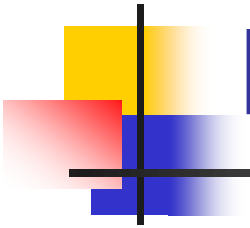


Three Major Concepts - 4

- Language recognizers – 4 automata for 4 languages
  - Finite Automata (FA) for RL
  - Push Down Automata (PDA) for CFL
  - Bounded TM for CSL
  - Turing Machine (TM) for REL

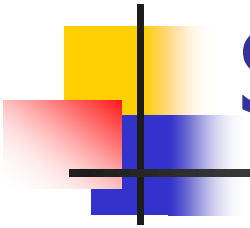
Automata: an abstract model of a digital computer





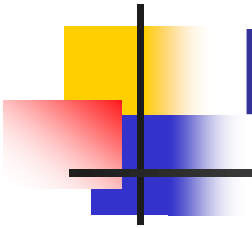
Languages and Abstractions

- Why do we need to study formal languages?
  - Formal language is a powerful abstraction that we can use it to design and implement new programming languages and study other computation problems
- What is an abstraction?
  - A method/language for working with one aspect of a design
  - All abstractions have limitations



Set Theory – a brief review

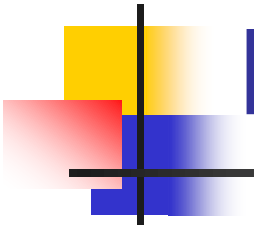
- Sets, subsets, empty set, multiset
- Cartesian product of A and B
- Power set of A:
  - 2^A -- set of all subsets of A
- Cardinality of A: size of set A
- Sets operations:
  - Union, intersection, difference, universal set



Languages in Set Notation - 1

- **Alphabet**: a finite, nonempty set S of symbols
- **Strings**: finite sequences of symbols from S
- **Language**: a set of strings from S

Languages in Set Notation - 2



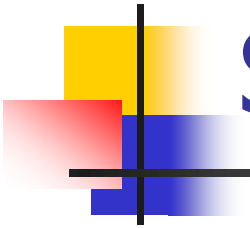
- **S^*** - set of strings obtained by concatenating zero or more symbols from S
- Empty string λ and empty set
- **S^+** = S^* - {λ}
- **Language**: a subset of S^*
- Example: S = {a, b}. What are S^* and S^+

Language Operations - 1

- A language L is a set
- set operations => language operations
- Complement of L, S^* - L
- Concatenation (product) of languages
  - $L_1 L_2 = \{xy: x \text{ from } L_1, y \text{ from } L_2\}$
  - Example of product when
    - $L_1 = \{0, 1\}$ and $L_2 = \{ a, b\}$
    - $L_1 L_2$ =?

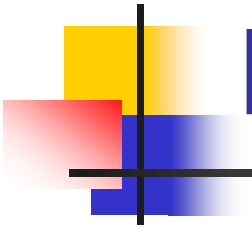
Language Operations - 2

- Power of L
  - $L^0 = \{\lambda\}$
  - $L^n = L \ \ L^{n-1}$
  - Example of power: given $L = \{a^n b^n : n \geq o?\}$
  - What is L^2?



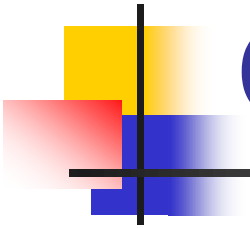
Summary

- See Class website for latest details:
  - http://athena.ecs.csus.edu/~mei/135/index.html
- To do well in this class, you need to
  - Read before/after class and attend class
  - Review lecture notes and HW before activities and tests
  - Do homework and participate in discussions
  - Ask questions in class and office hours



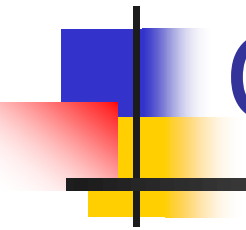
Homework

- Review set theory,
  - a quiz on set theory **next class**
- Read chapter 1 and 2 of (text or notes)



Quiz scope on set theory

- Subsets, notation and concepts
- Member of a set
- Empty set
- Power set
- Set operations: Product, union, intersection, difference



Chapter 2 Finite Automata

Syntax and Semantics

DFA – Deterministic FA

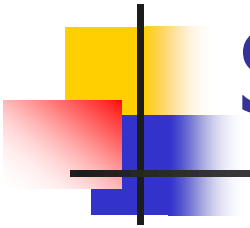
NFA – Nondeterministic FA

Equivalence of DFA and NFA

Syntax and Semantics -1

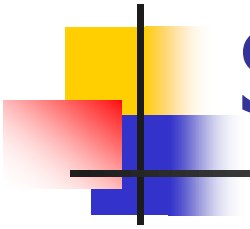
a statement from **Terence Parr**

- A ***language*** ***is*** *a set of valid sentences. What makes a sentence valid?*
- *You can break validity down into two things: syntax and semantics.*
  - ***syntax*** *refers to grammatical structure*
  - ***semantics*** *refers to the meaning of the vocabulary symbols arranged with that structure.*



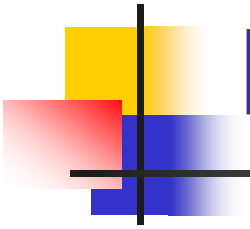
Syntax and Semantics - 2

- *Grammatical (**syntactically valid**) does not imply sensible (**semantically valid**)*
- *For example, the grammatical sentence "cows flow supremely" is grammatically ok (subject verb adverb) in English, but makes no sense.*
- *Similarly, in a programming language, your grammar (syntax rules) may allow but the language may only allow the meaningful sentence (a semantic rule).*



Syntax and Semantics - 3

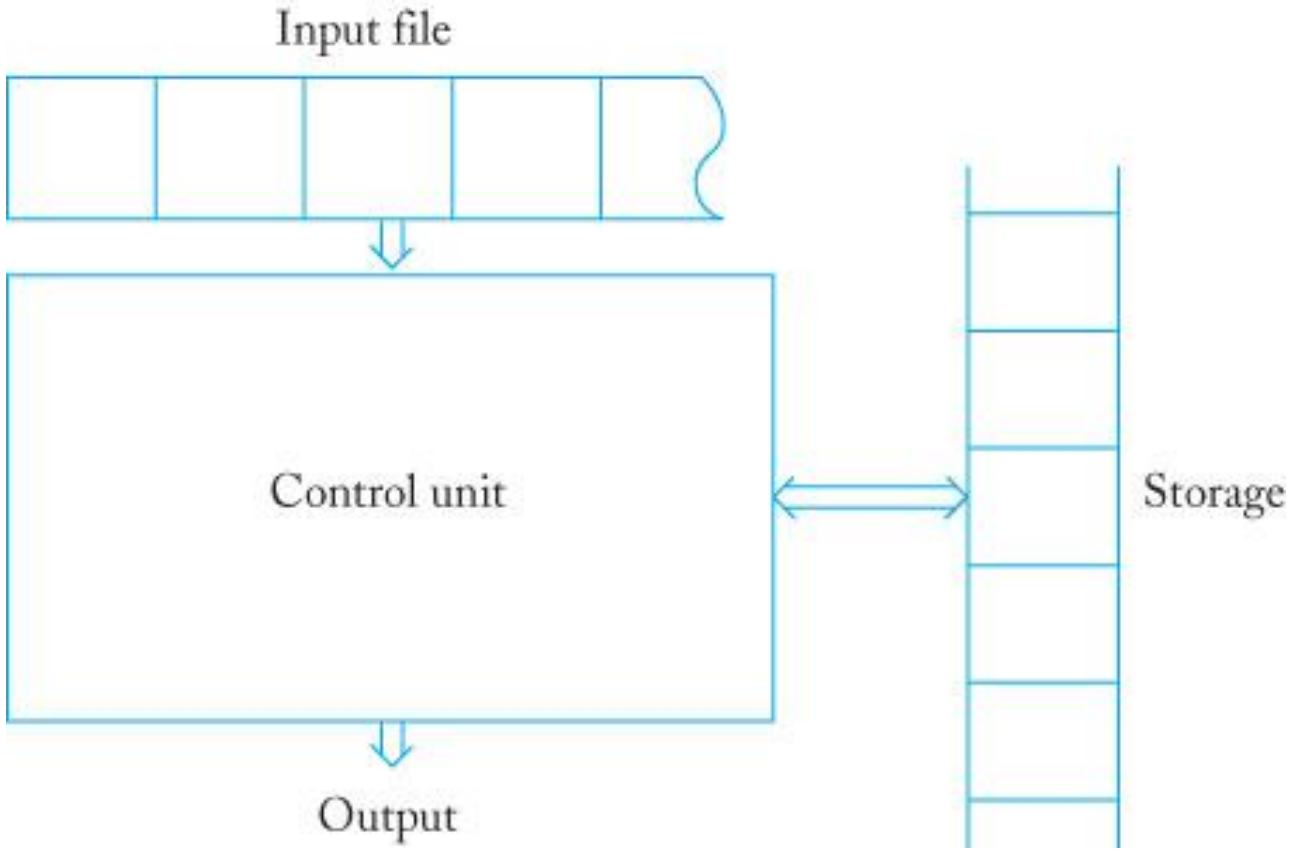
- *When you write a grammar, you are specifying the set of syntax rules obeyed by your language.*
- *When you use this to generate a recognizer for sentences in that language. To apply semantic rules, you must add actions or semantic predicates to the grammar.*
- *The actions test the "value" of the various tokens and their relationships to determine semantic validity.*
  - *For example, if you look up a type name in a symbol table to ensure it's a type not a variable, you are applying a semantic rule.*

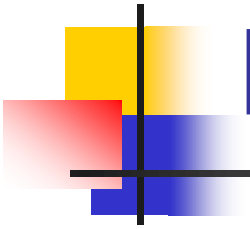


FA: a Representation of RL

- 4 representations of Regular Language (RL)
  - RE – regular expression
  - **FA – finite Accepters/Automata**
  - TG – a graph-based way of specifying patterns that can be represented by FA
  - RG – regular grammar

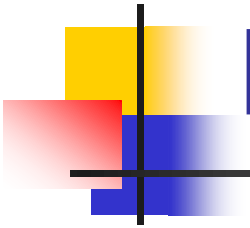
Automata:
FA does not have storage





DFA and NFA

- DFA – a deterministic FA is convertible in a simple way into a program that can serve as a recognizer for RL
- NFA – a nondeterministic FA is more powerful in design stage of a recognizer
- Every NFA can be converted into a DFA (see sec. 2.3)

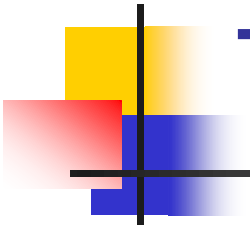


DFA

- Definition 2.1

  A **deterministic finite accepter** or **dfa** is defined by the quintuple $M = (Q, \Sigma, \delta, q_0, F)$ where

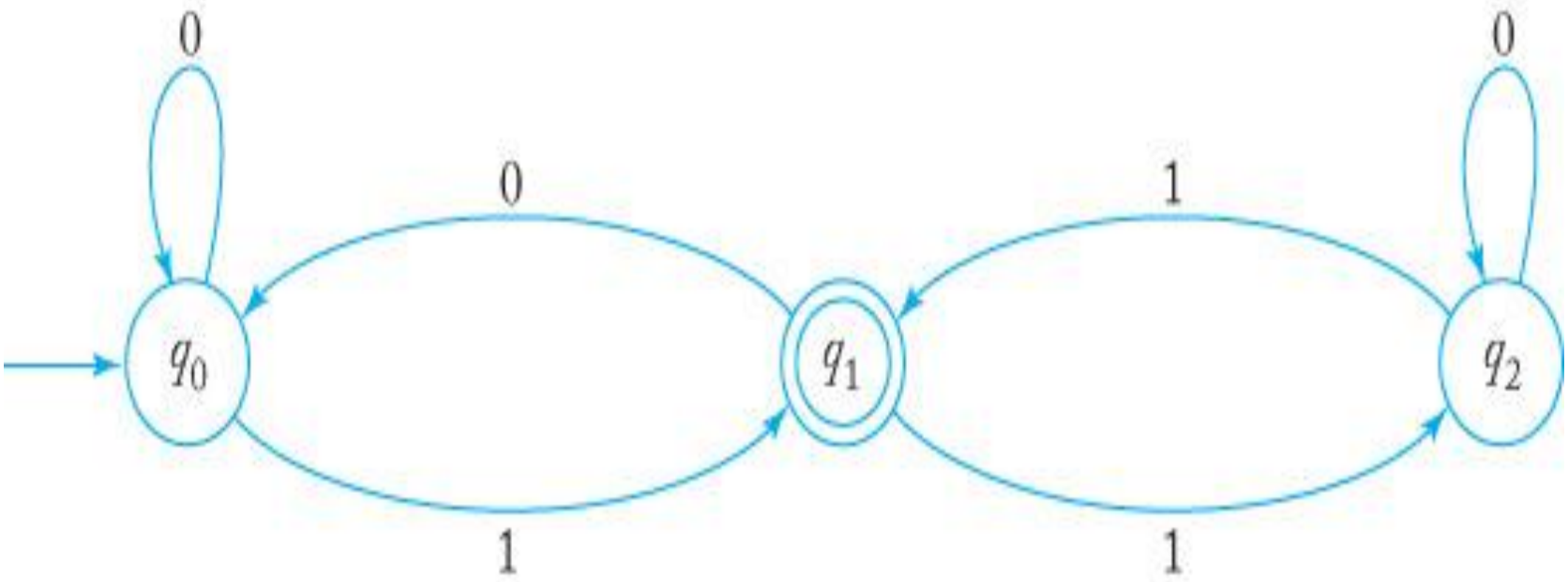
  - Q is a finite set of **internal states**
  - Σ is a finite set of symbols called the **input alphabet**
  - δ: $Q \times \Sigma \rightarrow Q$ is a total function called the **transition function**
  - $q_0 \in Q$ is the **initial state**
  - $F \subseteq Q$ is a set of **final states**

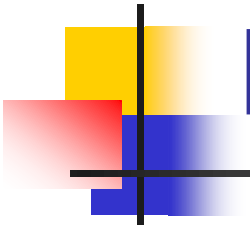


Transition Graph

- TG – a graphic notation for FA
  - **Example 2.1** and Figure 2.1 (page 39)
  - $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\})$
  - δ is given by
    - $\delta(q_0, 0) = q_0$, $\delta(q_0, 1) = q_1$,
    - $\delta(q_1, 0) = q_0$, $\delta(q_1, 1) = q_2$,
    - $\delta(q_2, 0) = q_2$, $\delta(q_2, 1) = q_1$,

Figure 2.1
TG for example 2.1





Languages and DFA's

- Definition 2.2 (page 40)
  - The language accepted by a dfa $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on alphabet accepted by M. In formal notation
    - $L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F \}$

Figure 2.2
A dfa with trap state, multiple label edge

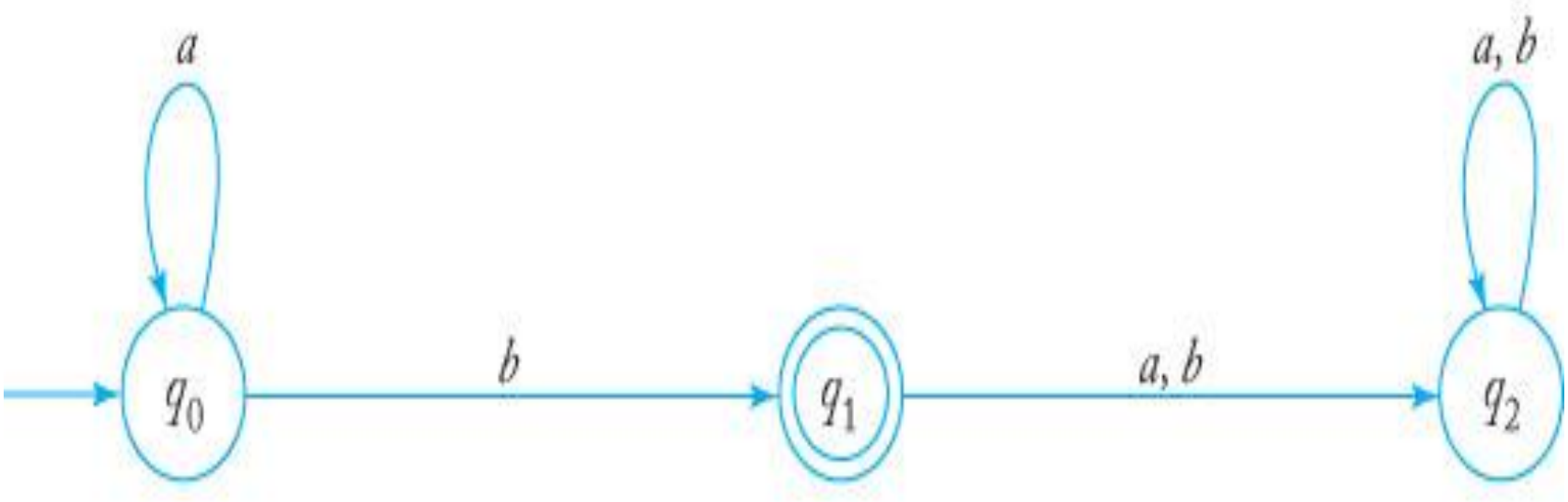
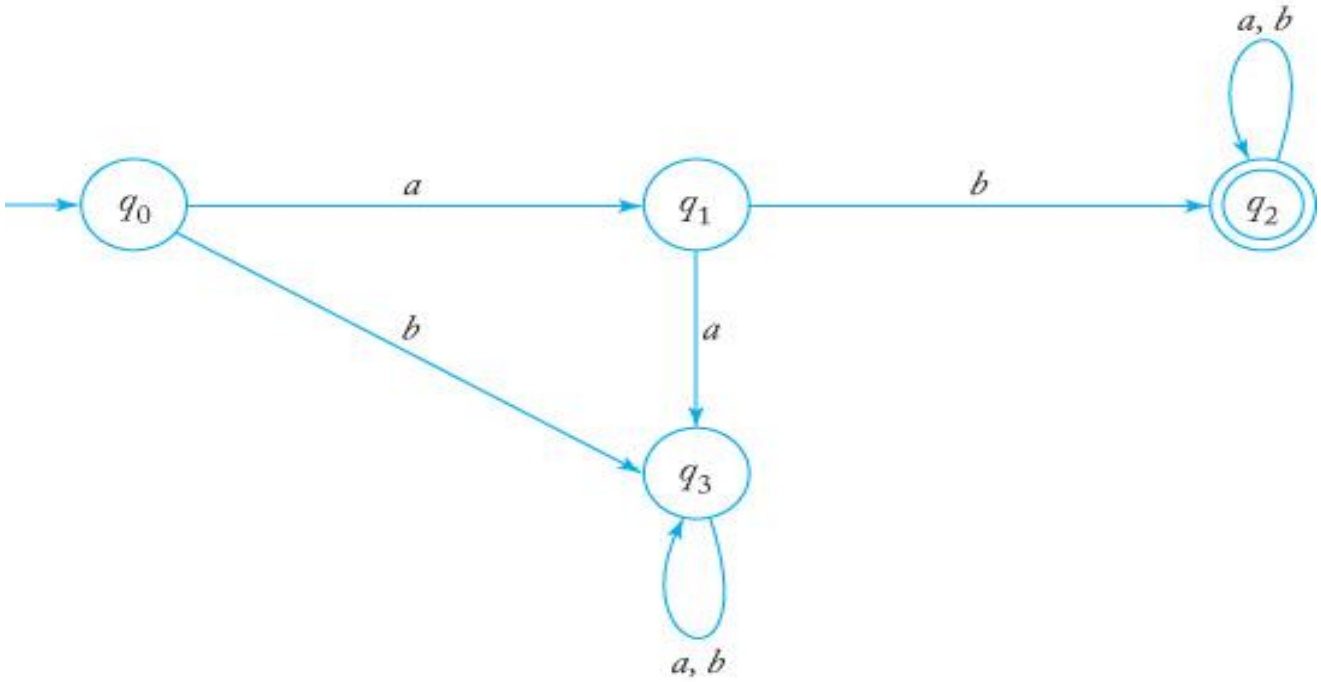


Figure 2.3
Table representation of dfa in Figure 2.2

	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

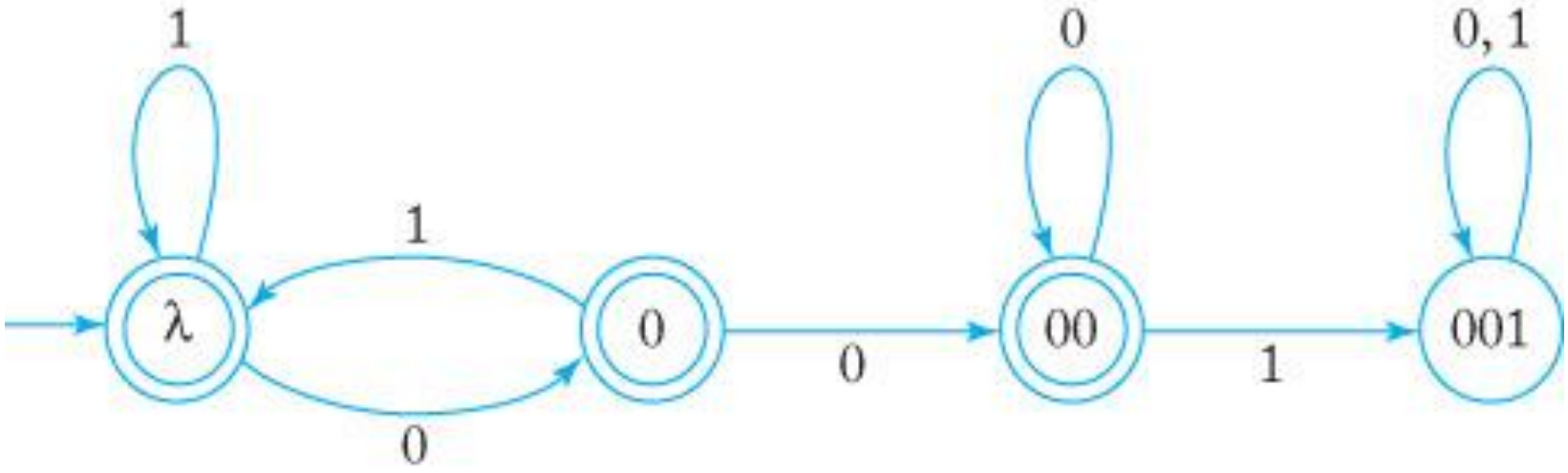
Example 2.3
dfa design technique

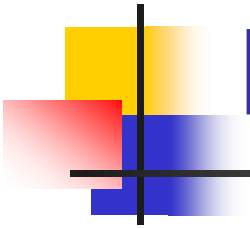
- Find a dfa that recognizes all strings starting with ab.



Example 2.4

- Find a dfa that accepts all the string s on {0, 1}, except those containing the substring 001.



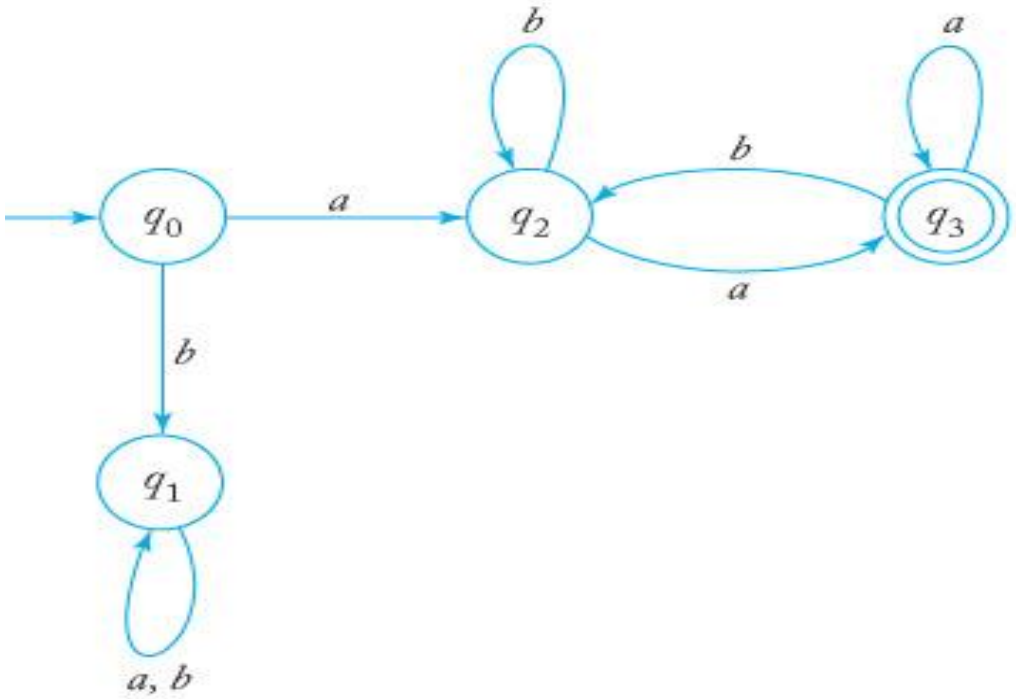


Regular Languages

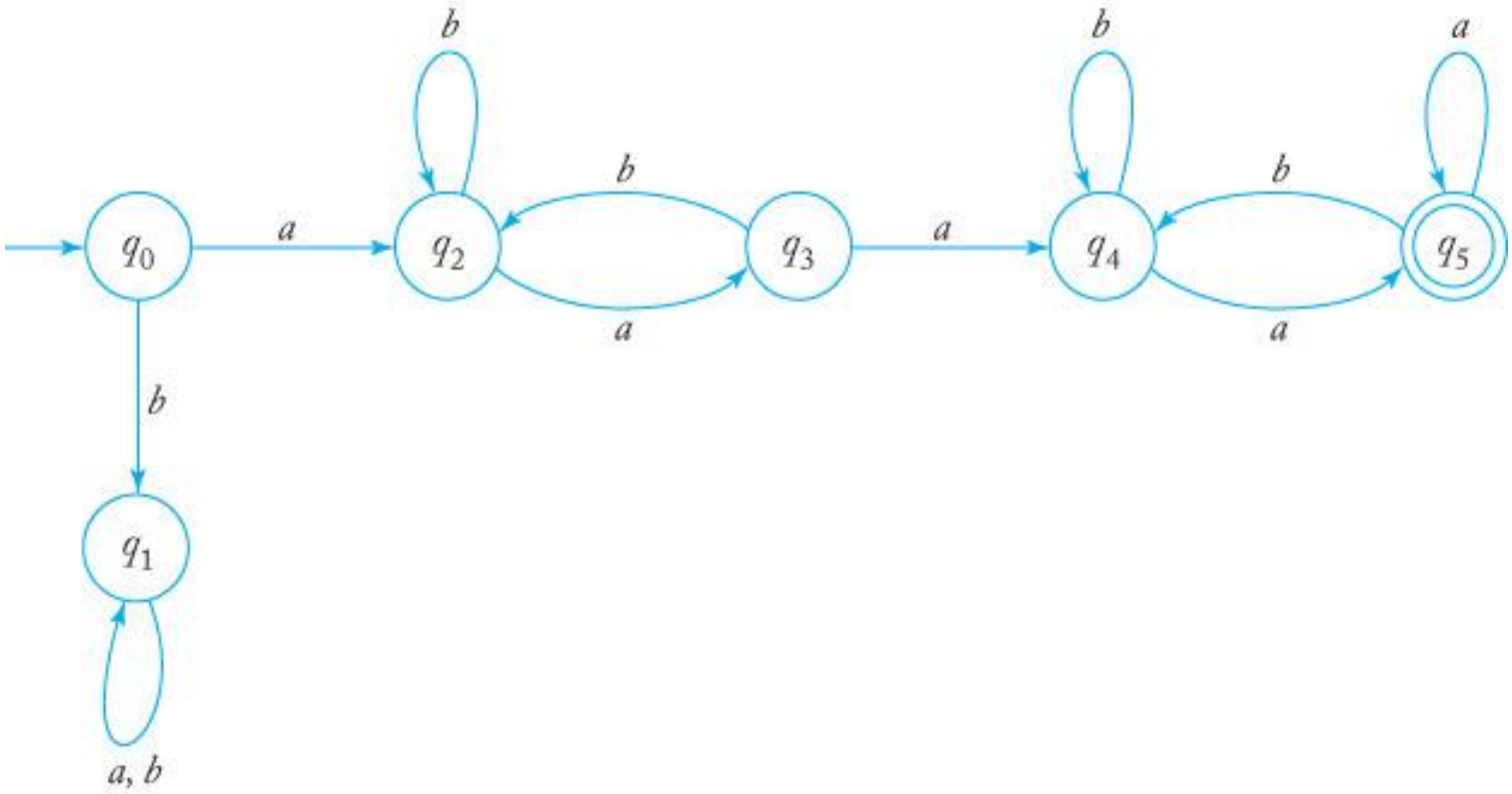
- The family of languages that is accepted by DFA is quite limited
- Definition 2.3
  - **A language L is called regular** if and only if there exists some dfa M such that
    - L = L(M)
- In class exercise to show a language is regular (see example 2.5)

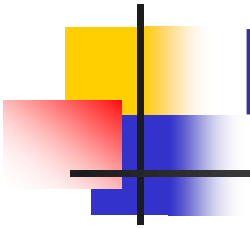
Example 2.5
Show that the language L is regular

- Show that L = {awa: w $\in$ {a, b}* } is regular



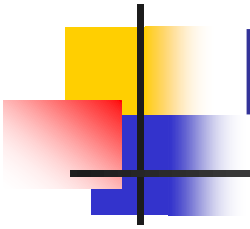
Example 2.6
If L is regular, show L^2 is regular too





Nondeterministic FA

- Nondeterminism is an effective mechanism for describing some complicated languages concisely
- Definition 2.4 NFA (page 49)
  - NFA is defined by the quintuple $M = (Q, \Sigma, \delta, q_0, F)$
  - Almost same like DFA with the only difference in transition function
  - $\delta : Q \times(\Sigma \cup \{\lambda\}) \rightarrow 2^Q$

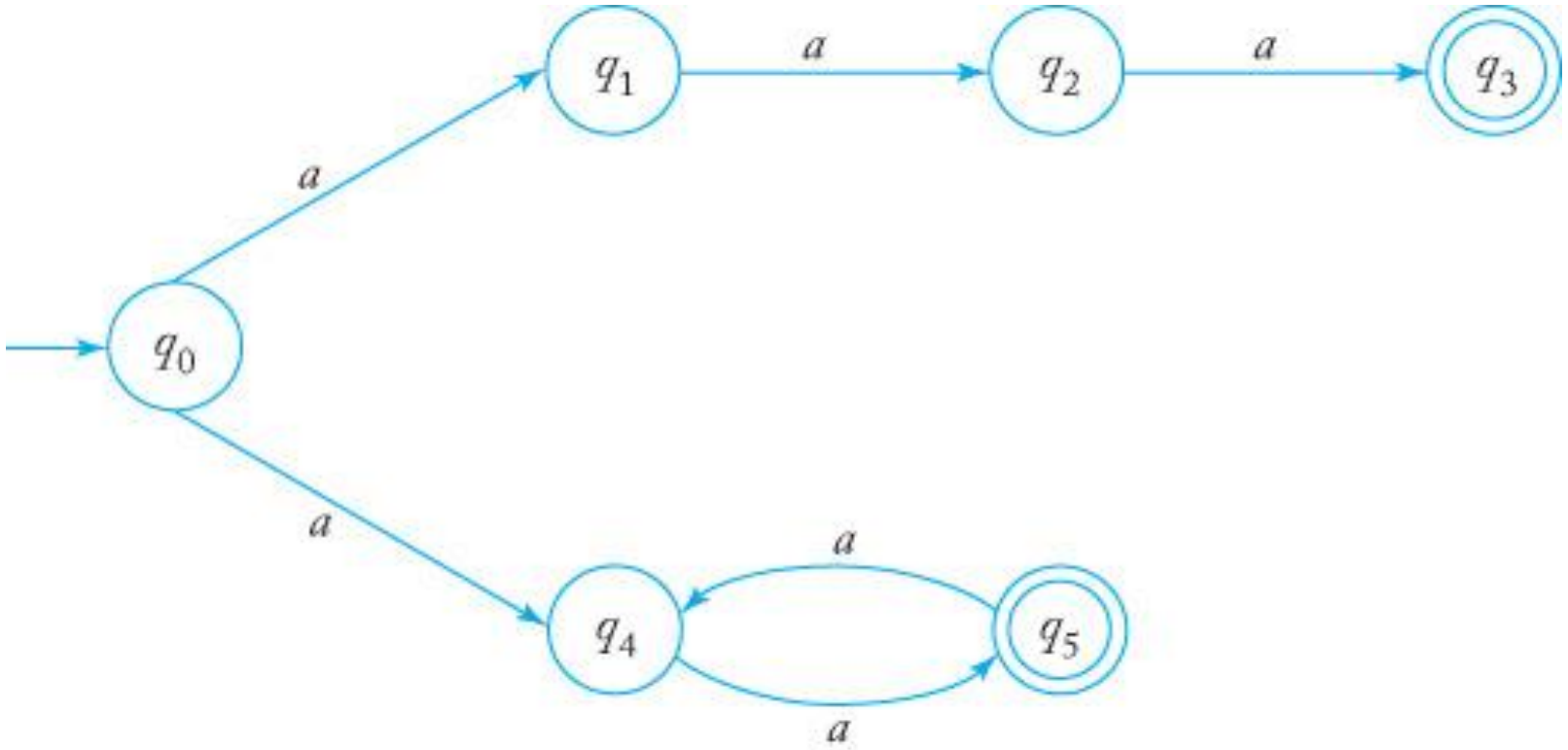


NFA and its TG notation

- In class exercise – show the definition difference of DFA and NFA
  - 2.2 #2 (page 55)
- TG – Transition graph
- Fig 2.8 and Fig 2.9 (page 50 - 51)

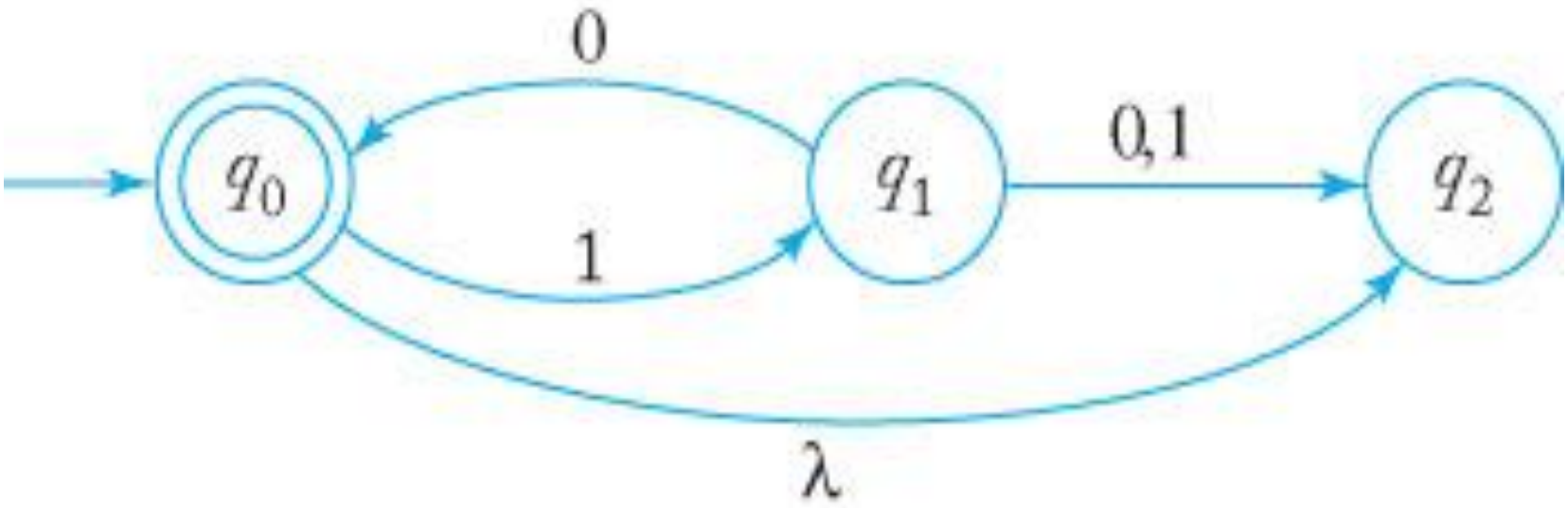
Example 2.7, Fig. 2.8

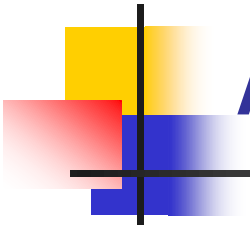
nfa with 2 transitions labeled the same



Example 2.8, Fig. 2.9

nfa: unspecified transition (to empty set) and λ transition

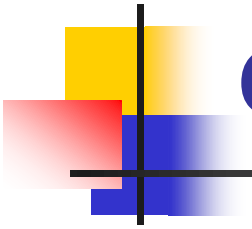




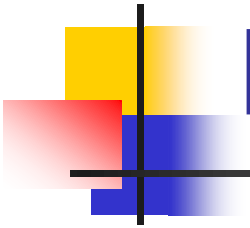
An Application of NFA

- Example 1: NFA that is designed to accept only strings in the form of a signed or unsigned decimal number
  - Accept strings:
    - -15.
    - 75.38
    - +.oo2
  - Reject strings:
    - 3..14
    - +17-56

Examples 2.9 extended notation *

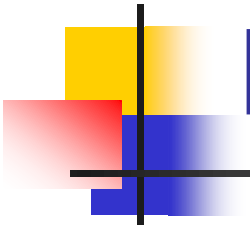


- Extended transition function notation * to allow you describe several applications of transition function in one compact expression
- Example 2.9 (page 51-52)
  - Write out the extended transition function for a new word, say aaa



Languages and NFA's

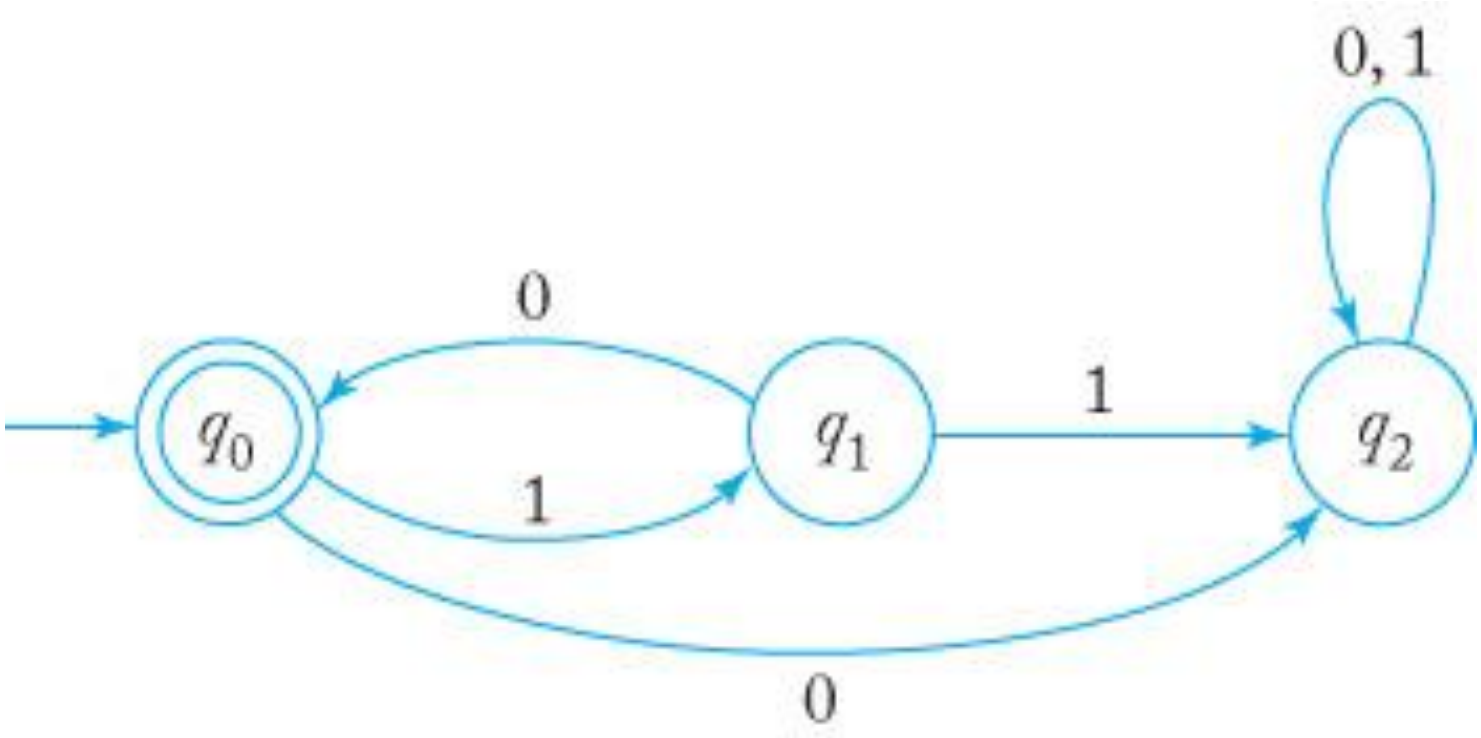
- Definition 2.6 (page 53)
  - The language accepted by an **nfa** M = (…) is defined as the set of all strings accepted by M. formally,
  - $L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing \}$
  - The language consists of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.
  - What is complement of L(M)?

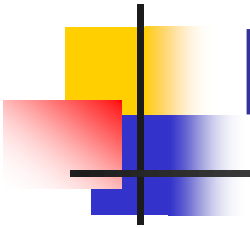


Equivalence of DFA and NFA

- Definition 2.7 (page 56)
  - Two finite accepters M_1 and M_2 are said to be equivalent if $L(M_1) = L(M_2)$, that is, if they both accept the same language
- Example 2.11
  - $L = \{(10)^n : n \geq 0\}$ (page 57)
  - There are many dfa or nfa for a given language
  - Fig 2.11 dfa for L
  - Can you draw nfa for L?

Example 2.11, Fig 2.11



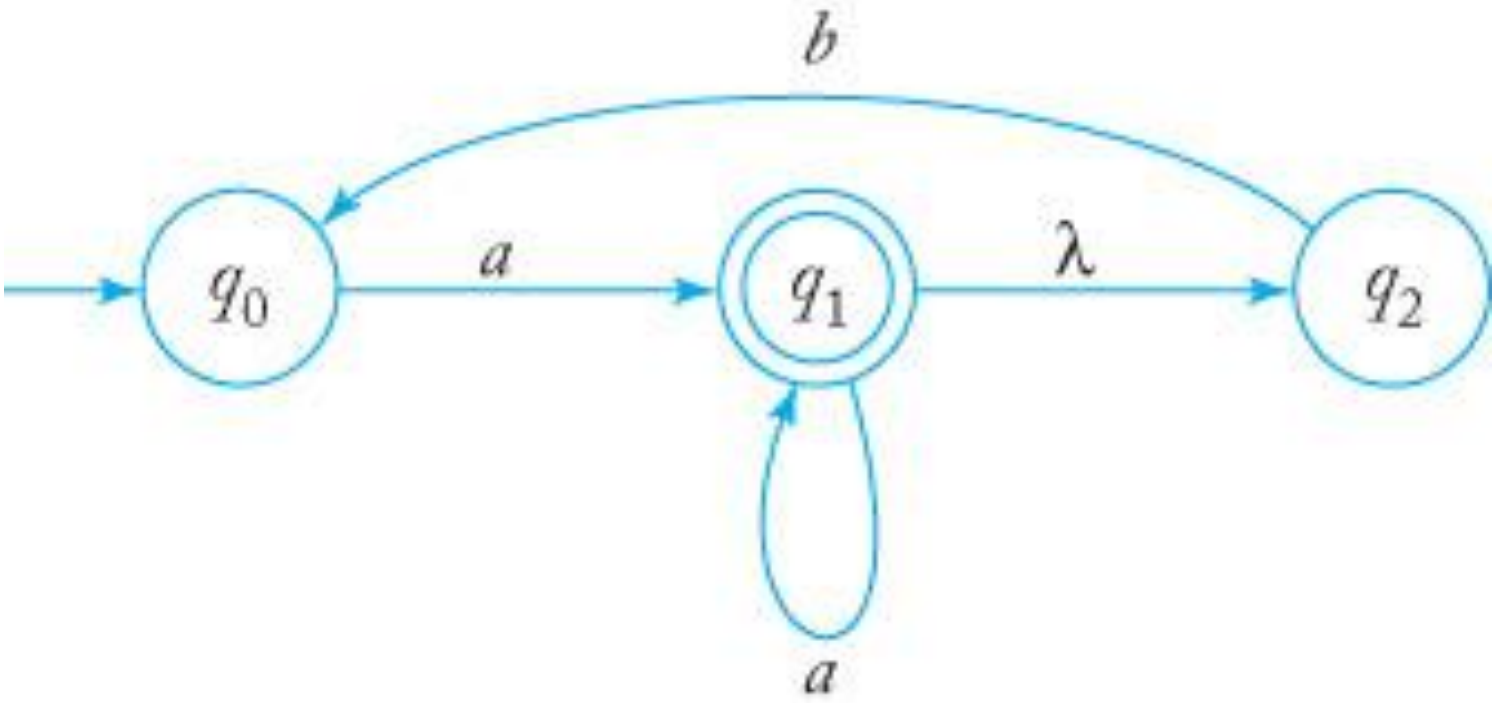


Review of Power Set

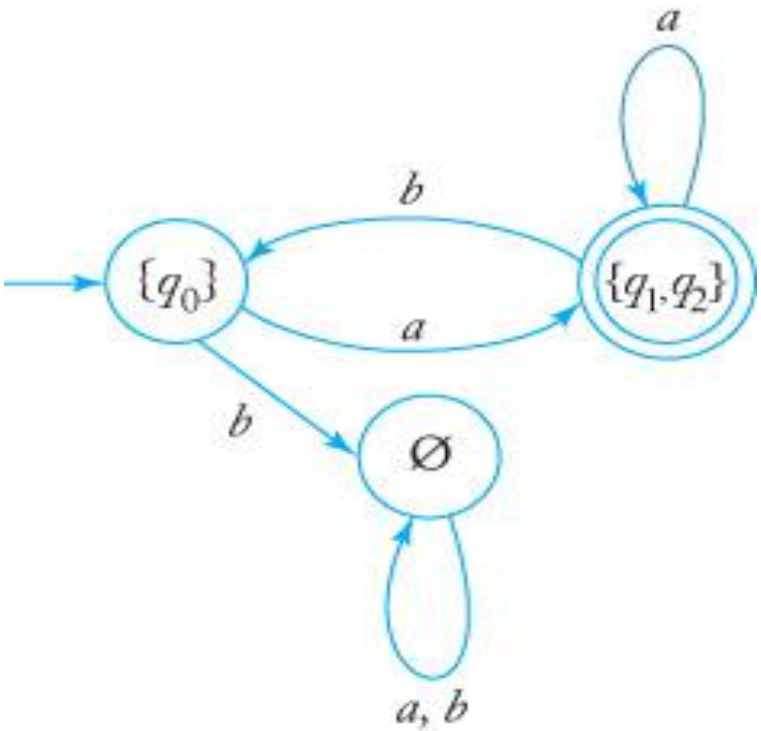
- For A = {0, 1}, 2^A =?
- For B = {0, 1, 2}, 2^B =?
- For C = {q_0, q_1, q_2}, 2^C =?

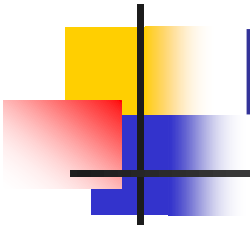
Example 2.12, Fig. 2.12

Convert the nfa to an equivalent dfa



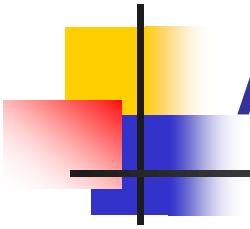
Example 2.12, Fig 2.13
The resulted dfa





Procedure: nfa-to-dfa

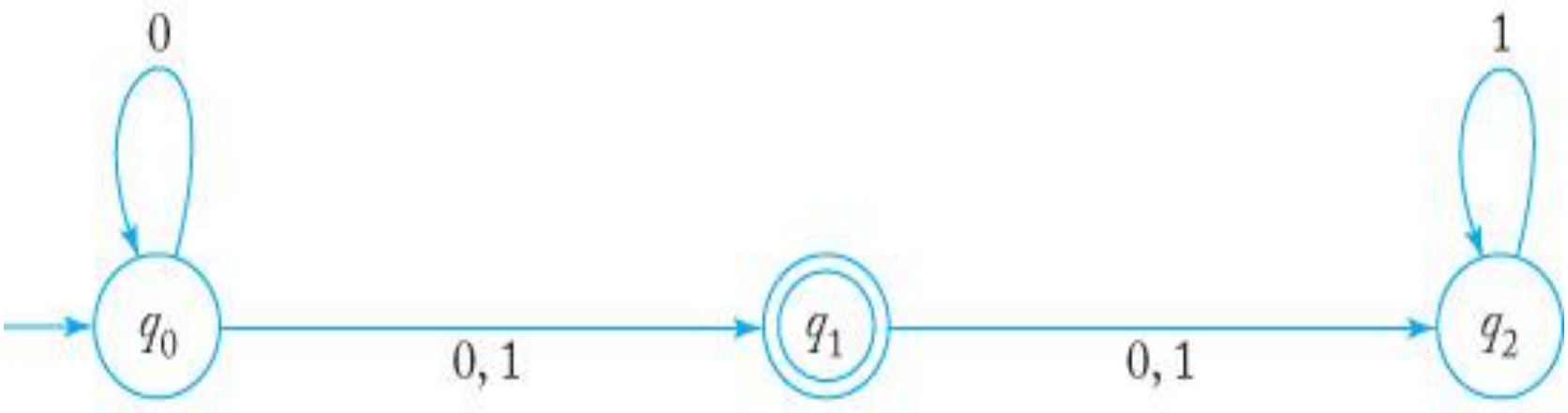
- Theorem 2.2 (page 59)
  - Let L be the language accepted by a nfa M_N. Then there exists a dfa M_D such that
    - $L = L(M_D)$
  - Proof of Theorem 2.2 is a constructive proof which shows the procedure of transforming nfa to dfa step by step
  - The key of the procedure is to use the power set of Q_N to be Q_D in construction of the new dfa M_D



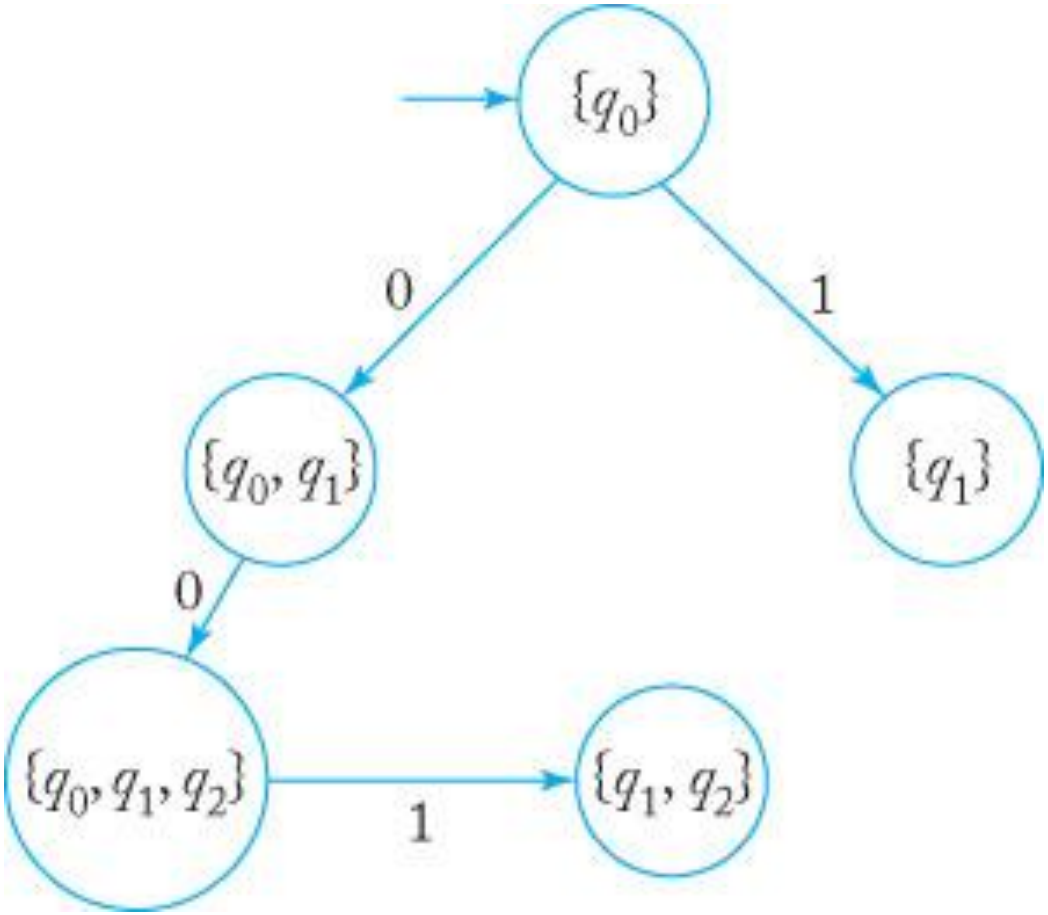
Another example of nfa-to-dfa

- Example 2.13 (page 60 – 61)
  - If you are still not sure about how the procedure nfa-to-dfa works, check out this example
  - It is important to know that every language accepted by an nfa is regular
  - What is a regular language?

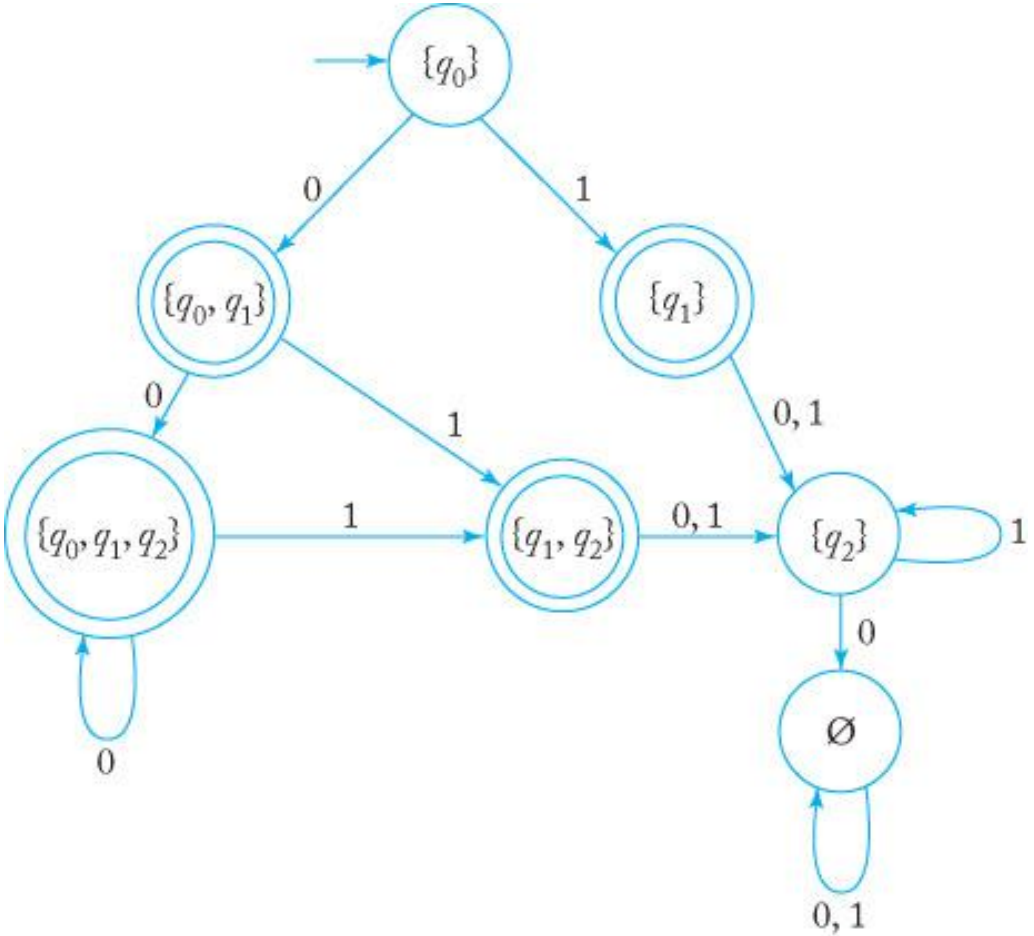
Example 2.13, Fig. 2.14

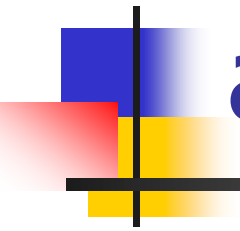


Example 2.13, Fig. 2.15 applying procedure nfa-to-dfa



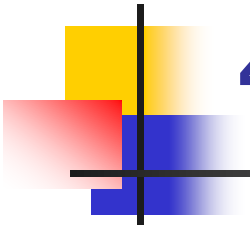
Example 2.13, Fig. 2.16 complete solution





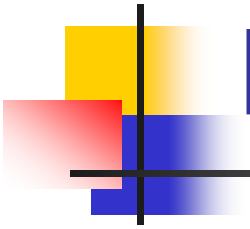
Chapter 3 Regular Languages and Regular Grammar

Recursive Definitions
Regular Expressions
RE and RL
Regular Grammars



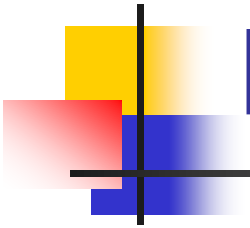
4 Representations of RL

- FA: DFA, NFA
- TG: Transition Graph
- **RE: Regular Expression**
- **RG: Regular Grammar**



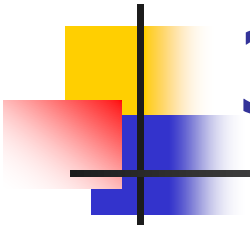
Recursive Definition: 3 Steps

- Rule 1. Specify basic objects in the set.
- Rule 2. Give rules for constructing more objects in the set.
- Declare that no other object in the set
- Example: EVEN in recursive definition
  - Rule 1. 2 is in EVEN
  - Rule 2. If x is in EVEN then so is x + 2.
  - Rule 3. Those are the only elements in EVEN.



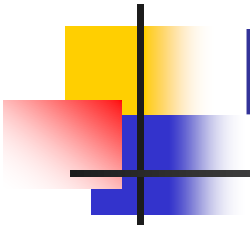
Recursive definition

- Give recursive definition for the following languages
  - ODD PALINDROM
  - EVEN PALINDROM
  - PALINDROM
  - Set ODD = {1, 3, 5, …}



3.1 Regular Expression

- A RE is a concise way of expressing a pattern in a series of characters.
- Regular Language (RL) – a language that can be defined by regular expression.
- RE is one of 4 representations for RL:
  - **RE – Regular Expression**
  - FA – Finite Automata
  - TG – Transition Graph
  - RG – Regular Gramma



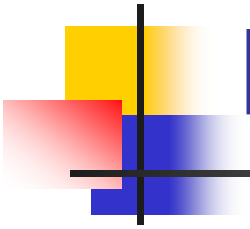
Formal Definition of RE

- Definition 3.1 (page 72)
  - Let Σ be a give alphabet, then
    - 1. $\varnothing$, λ, $a \in \Sigma$ are regular expressions (RE). These are called primitive RE.
    - 2. If r_1 and r_2 are RE, so are $r_1 + r_2$, $r_1 . r_2$, r_1^* and (r_1).
    - 3. A string is a RE if and only if it can be derived from rule 1 and rule 2.

Languages Associated with RE

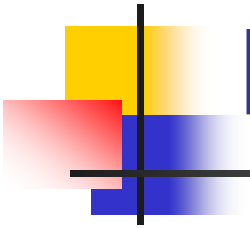
- Definition 3.2 (page 72)
  - The language L (r) denoted by any RE r is defined by the following rules.

1. $\varnothing$ is a regular expression denoting the empty set
2. λ is a RE denoting $\{\lambda\}$
3. For every $a \in \Sigma$, a is a RE denoting {a}
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
5. $L(r_1 . r_2) = L(r_1)\, L(r_2)$
6. $L((r_1)) = L(r_1)$
7. $L(r_1^{*}) = (L(r_1))^{*}$



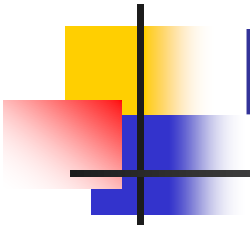
RE Examples - 1

- Example 3.2 (page 73)
  - L(a*.(a+b)) = L (a*)L(a+b)
  - = {λ, a, aa, aaa, …} {a, b}
  - = {a, aa, aaa, …, b, ab, aab,…}
- In-class exercise: try exhibit the following RL in set notation and NFA
  - L ((a+b).a*)
  - L (aa*)
  - L ((a+b)*)



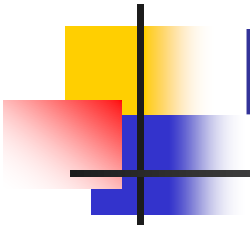
RE Examples - 2

- Example 3.3 r = (a+b)*(a+bb), L =?
- Example 3.4 r = (aa)* (bb)*b, L =?
- For Σ = {0, 1}, give a RE such that
  - L(r) = {w$\in\Sigma$*: w has at least one pair of consecutive zeros}
  - L(r) = {w$\in\Sigma$*: w has no pair of consecutive zeros}



RE Examples - 3

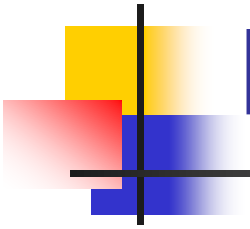
- Example 3.5
  - r = (0+1)* 00(0+1)*
- Example 3.6
  - r = (1 +01)*(0+λ)



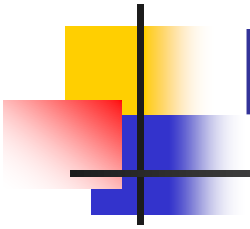
RE Examples - 4

- Exhibit L = { ? } and nfa
  - X*
  - XX*
  - a*b*
  - (ab)*
  - X(XX)*
  - b*ab*a(a+b)*
  - (a+b)(a+b)(a+b)

RE Examples - 5



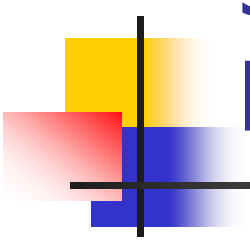
- Informal notations: a^+, a^3
- RE with * always denoting a language that contains λ
- RE to English descriptions:
  - (a+b)*
  - (a+b)*a(a+b)*
  - (a+b)*aaa
  - a(a+b)*b



RE Examples - 6

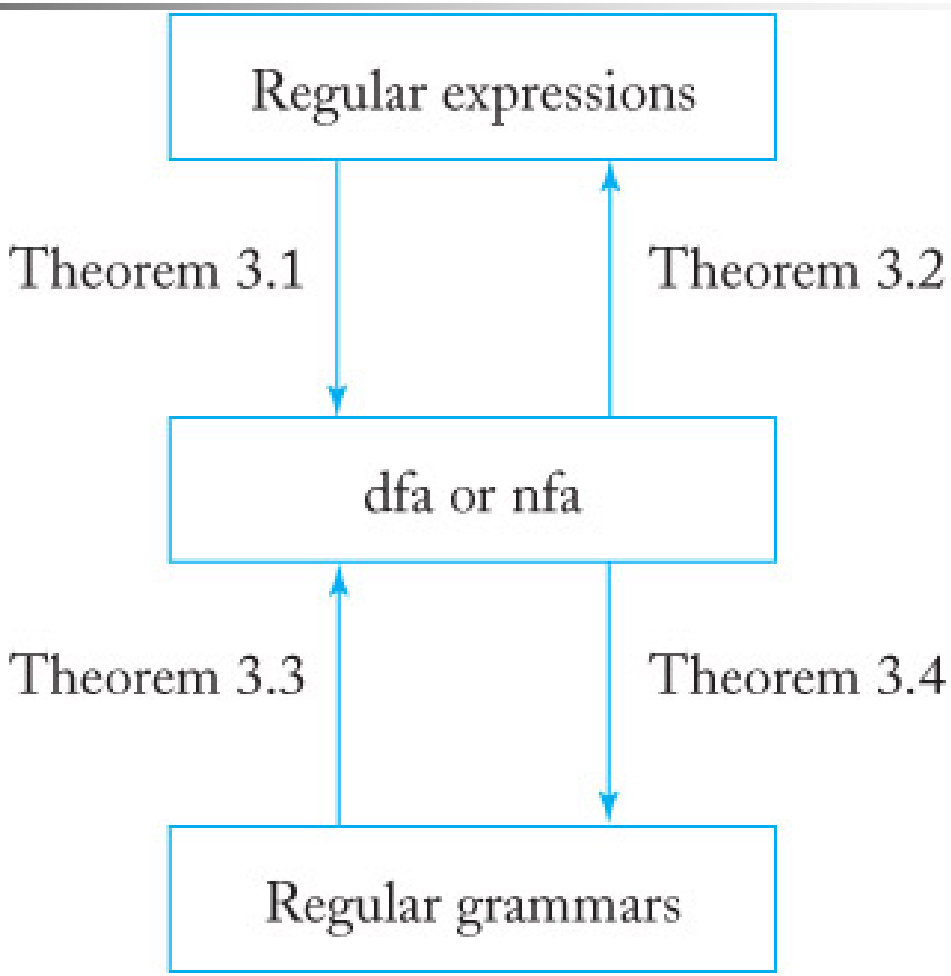
- An application of RE:
  - Letter(Letter +Digit)*
    - A set of strings that beginning with a letter
  - Letter = A+B+…+Z+a+b+…+z
  - Digit = 0+1+2+…+9

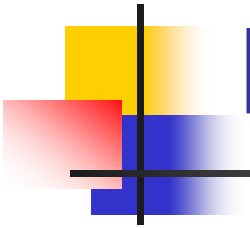
3.2 Connection Between Regular Expressions and Regular Languages



- RE denote Regular languages
- Kleen's Theorem:
  - L(FA) = L(TG) = L(RE)
- Theorem 3.1
  - $L(RE) \subset L(TG)$
- Theorem 3.2
  - $L(FA) \subset L(RE)$

Four Theorems in this Chapter

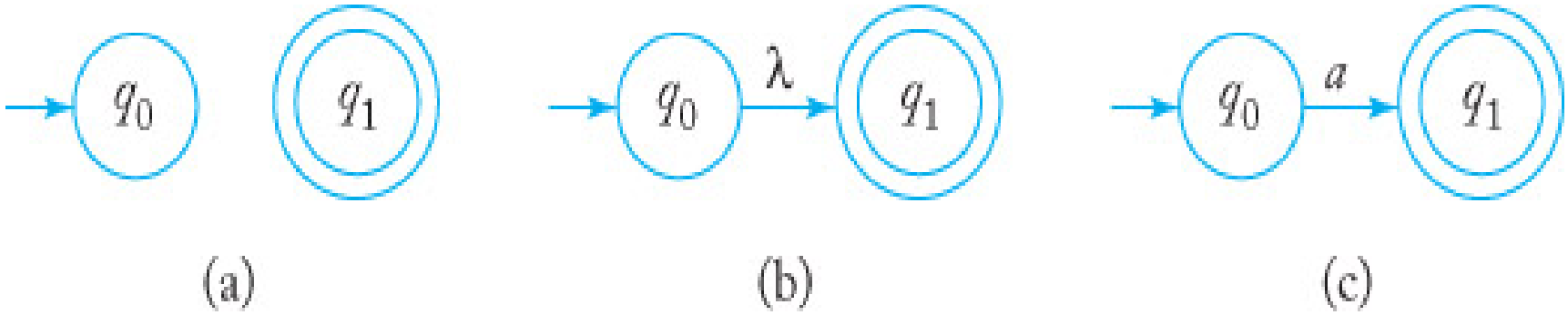




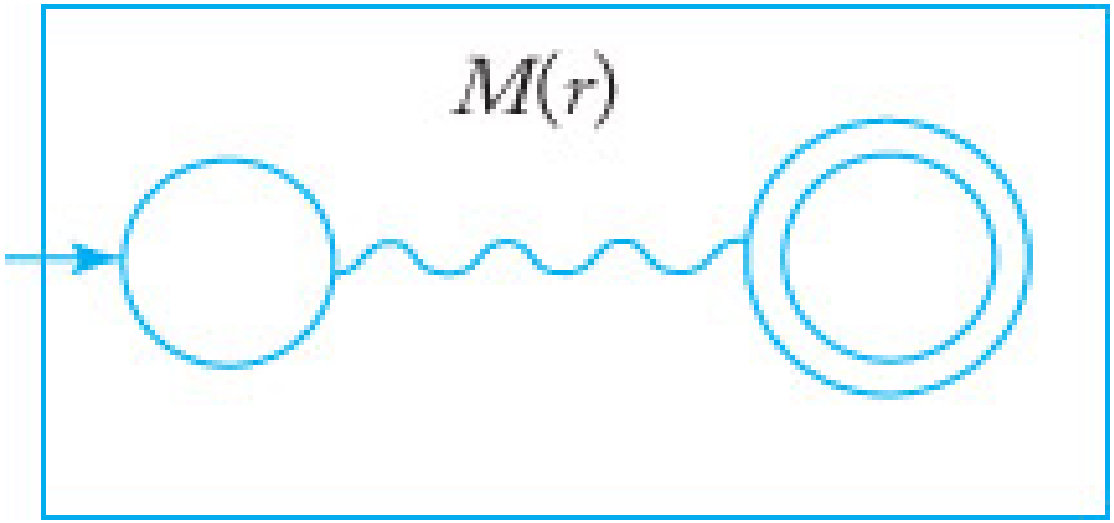
RE $\Rightarrow$ FA

- Theorem 3.1 RE $\Rightarrow$ NFA
- Proof – constructive proof based on recursive definition of RE
  - Step 1: construct nfa for each basic element defined in rule 1
  - Step 2: construct nfa for each operation defined in rule 2

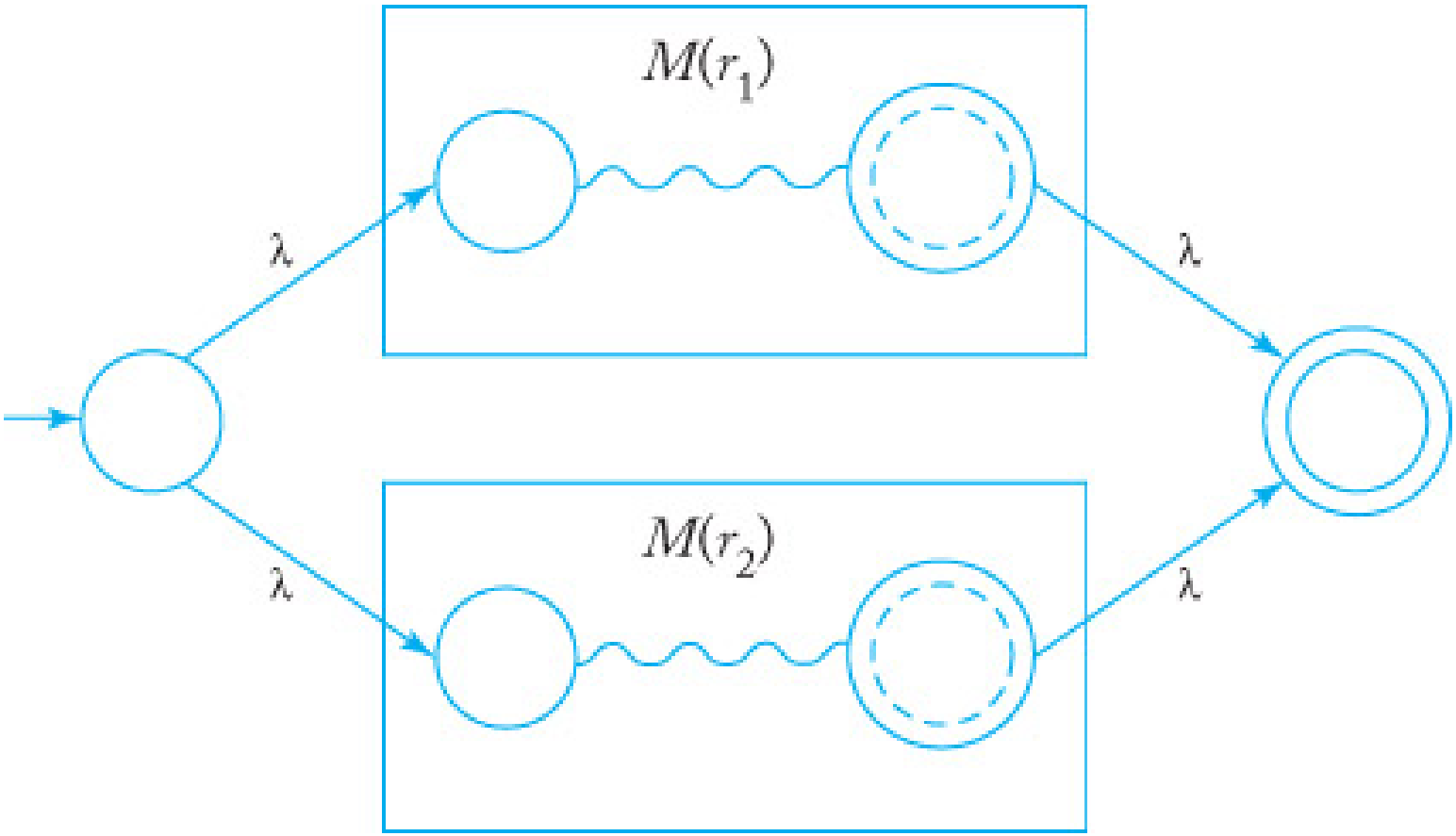
RE $\Rightarrow$ FA or TG proof step 1: an nfa for basic elements of RE



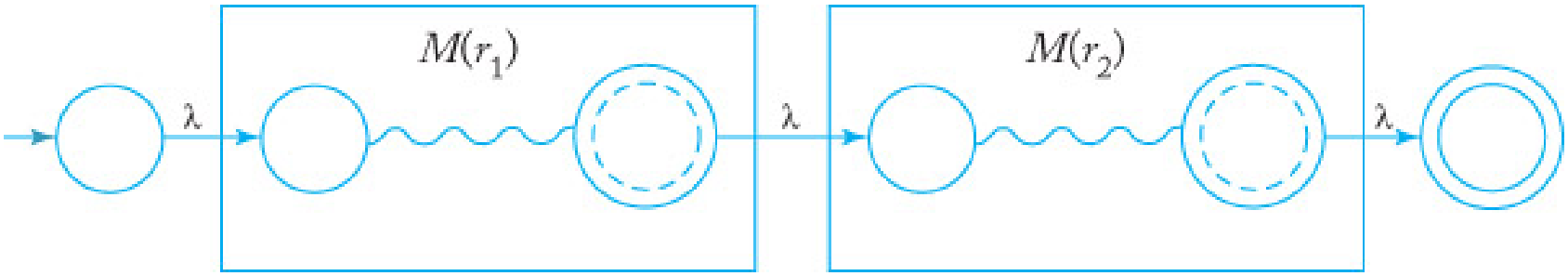
Schematic representation of an nfa accepting L(r)



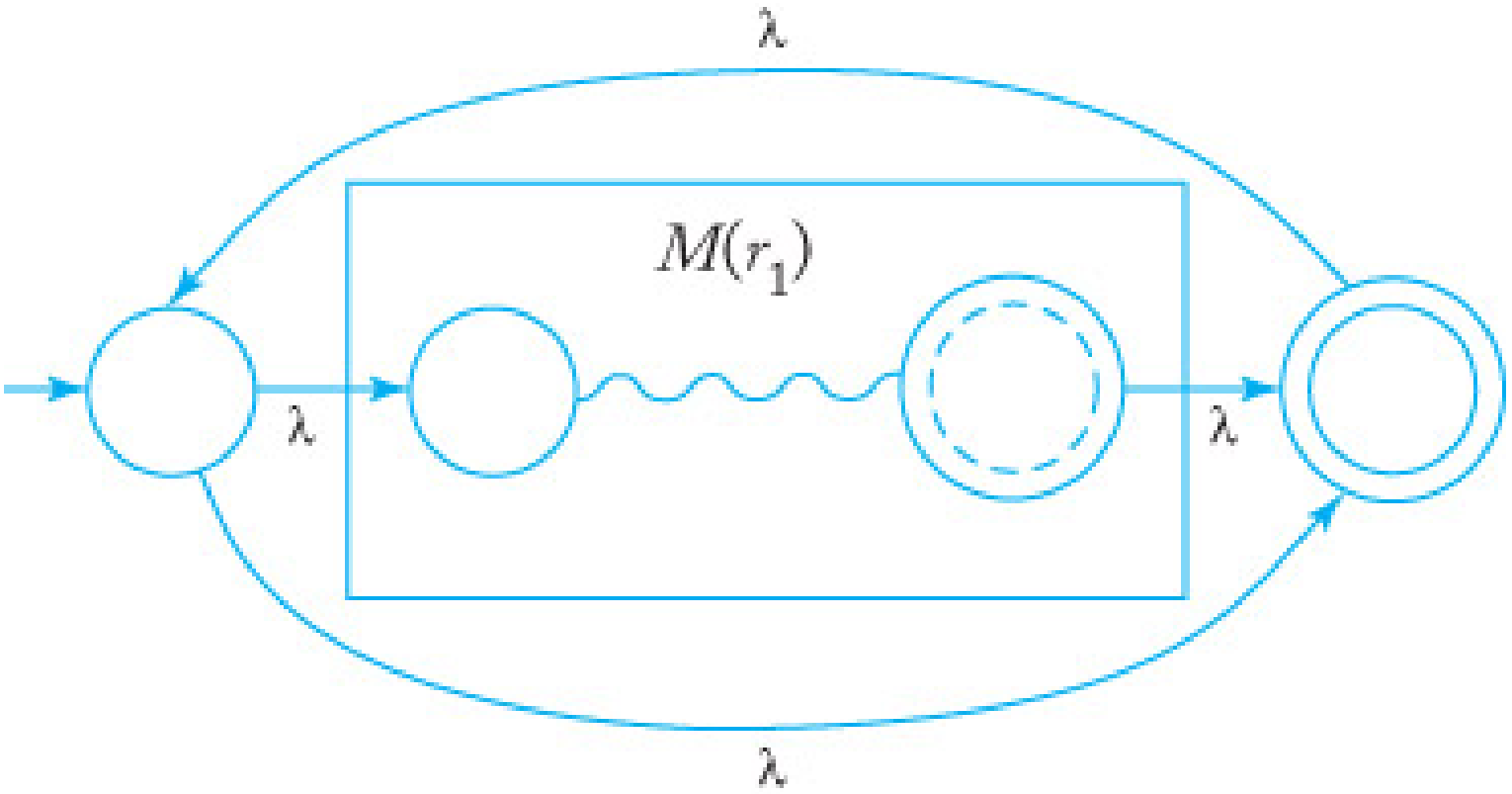
RE ⇒ FA proof Step 2-1: nfa for $L(r_1 + r_2)$



RE $\Rightarrow$ FA proof Step 2-2: nfa for L(r_1 r_2)



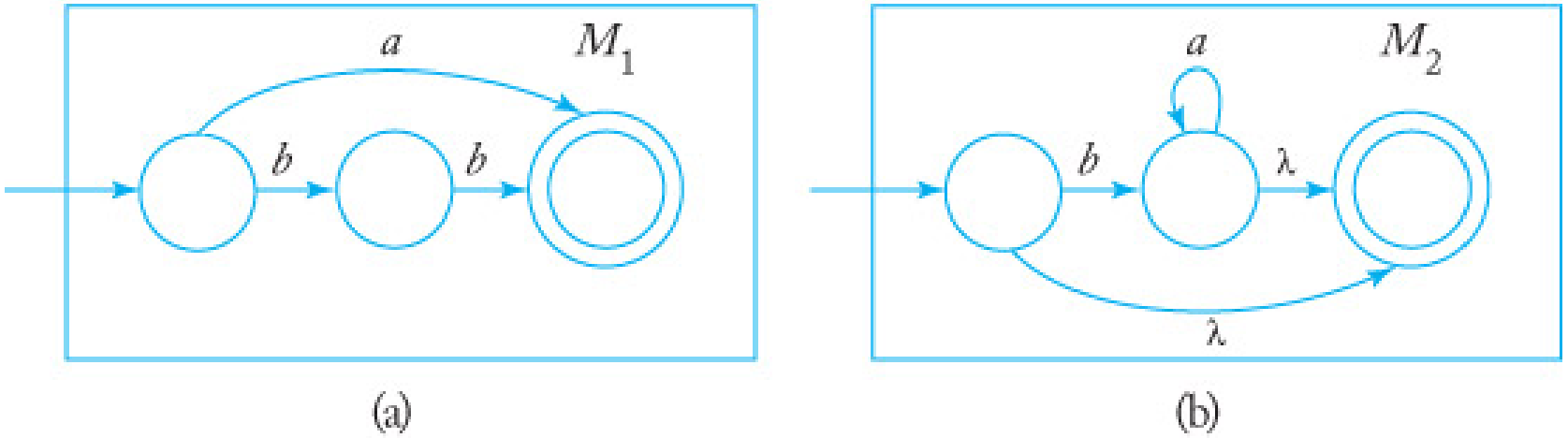
RE $\Rightarrow$ FA proof Step 2-3: nfa for L(r_1*)



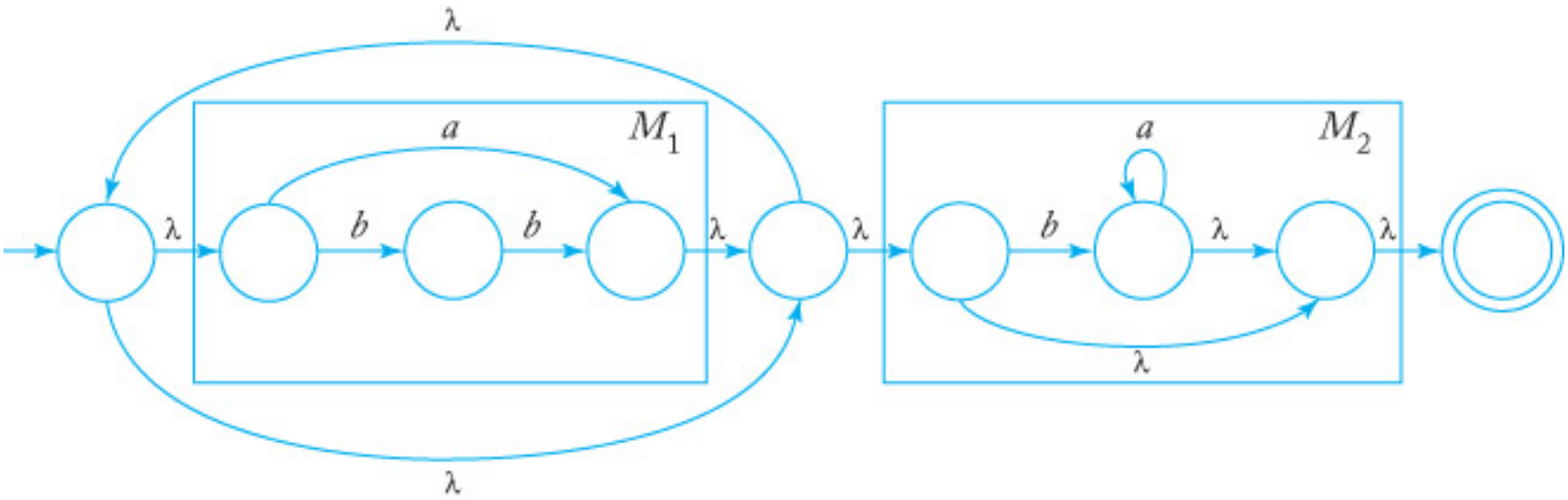
RE $\Rightarrow$ FA Example 3.7

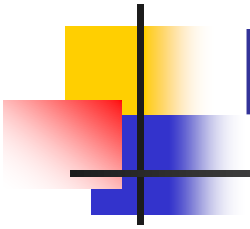
- Find an nfa that accepts L(r), where
  - r = (a + bb)*(ba* + λ)
- We first construct nfa for
  - r_1 = (a+bb) and
  - r_2 = (ba* + λ)
- Then putting them together according to Theorem 3.1 for r
  - r_1* r_2

(a+bb) and (ba* + λ)



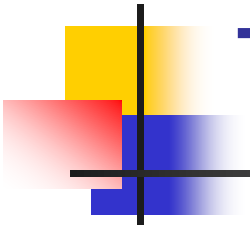
Putting them together (a + bb)*(ba* + λ)





RE to TG In-class exercises

- Convert RE to TG:
  - (a+b)*
  - (a+b)*a(a+b)*
  - (a+b)*aaa
  - a(a+b)*b
  - Even-Even: even number of a's and even number of b's

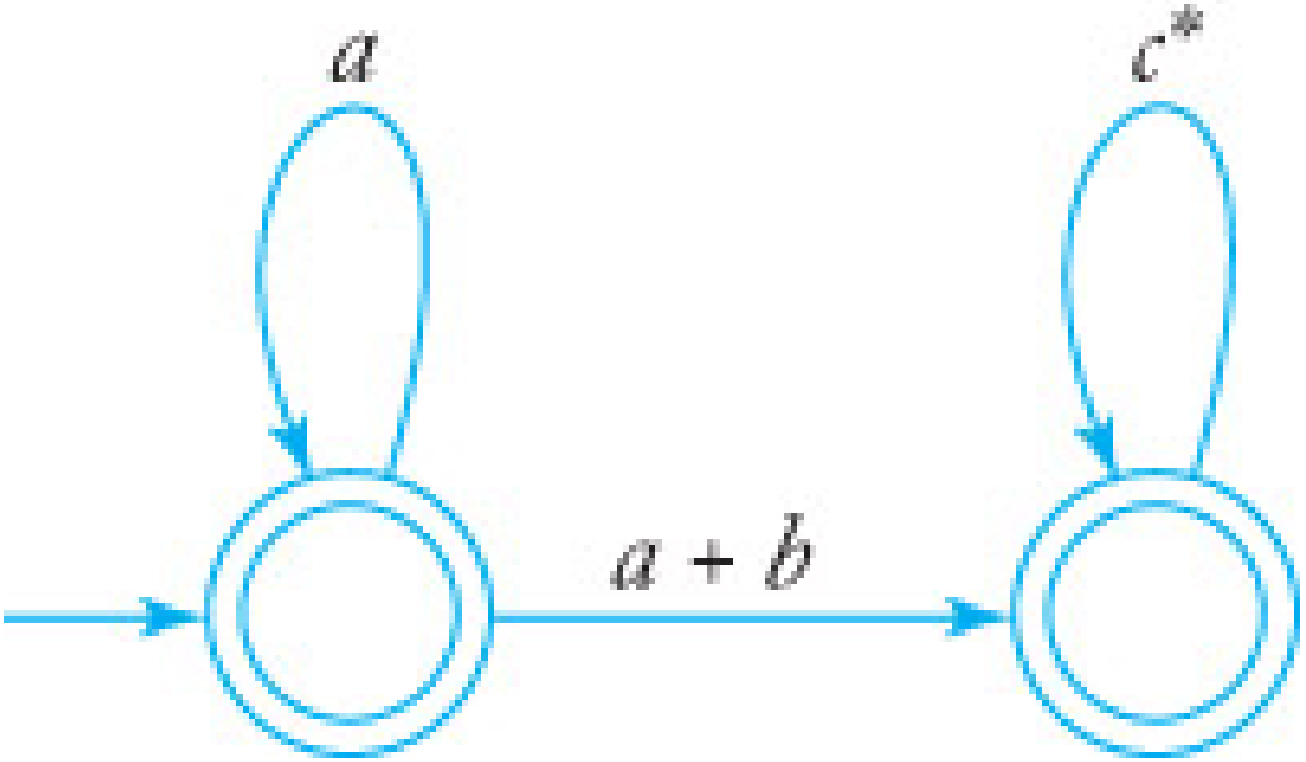


TG $\Rightarrow$ RE

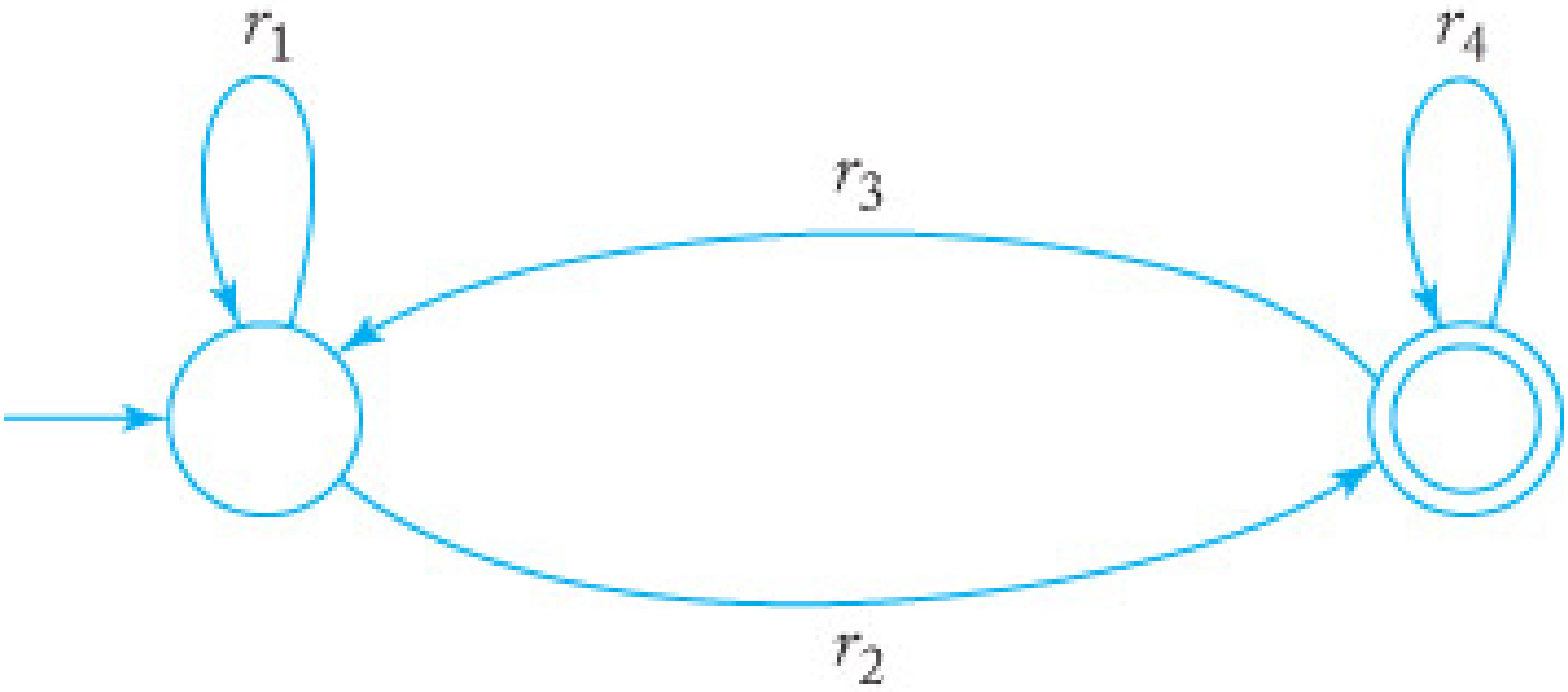
- Theorem 3.2
  - Let L be a regular language. Then there exists a RE r such that L = L(r).
  - Proof: If L is regular, there exists an nfa.
    - Convert nfa to a two-state complete GTG
    - Apply the procedure nfa-to-rex $\Rightarrow$ RE
  - GTG – generalized transition graphs

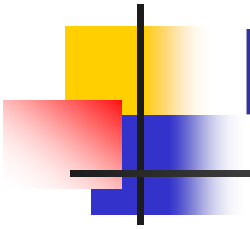
An Example of GTG $\Rightarrow$ RE

L(a* +a*(a+b)c*)



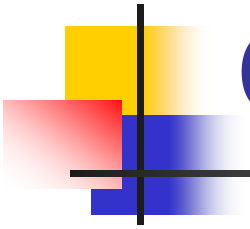
Two-state complete GTG to $r_1^* r_2(r_4 + r_3 r_1^* r_2)^*$





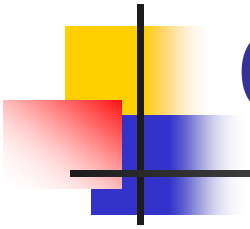
Procedure: nfa-to-rex

- Page 83-84
- Step 1 to step 6
- This is a reading assignment



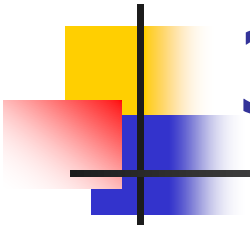
Grammars -- 1

- **Grammar**
  - a scheme for specifying the sentences allowed in the language, indicating the syntactic rules for combining words into well-formed phrases and clauses
  - Defined by Feighenbaum et al.
  - *Natural language understanding* has long been a goal of AI researchers
  - Example: google cross-language info retrieval



Grammars -- 2

- MIT linguist *Noam Chomskey* did the seminal work in the systematic and mathematical study of language syntax →computational linguistics
- **Formal language** – a set of strings composed of a vocabulary of symbols according to rules of grammar



3.3 Regular Grammar

- One way of describing regular language: right- and left-linear grammar
- Definition 3.3
  - A grammar G = (V, T, S, P) is said to be right-linear if all productions are of the form
    - $A \rightarrow xB$,
    - $A \rightarrow x$,
    - where $A, B \in V$, and $x \in T^*$.
- A regular grammar is one that is either right-linear or left-linear

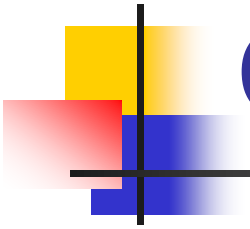
Example 3.13
right- and left-linear grammars

- $G_1 = (\{S\}, \{a, b\}, S, P_1)$, with P_1 given as
  - $S \rightarrow abS \mid a$ is right-linear
  - A derivation with G_1: $S \Rightarrow abS \Rightarrow ababS \Rightarrow ababa$
  - $L(G_1)$ is L ((ab)*a)
- $G_2 = (\{S, S_1, S_2\}, \{a, b\}, S, P_2)$, with P_2 as
  - $S \rightarrow S_1ab$,
  - $S_1 \rightarrow S_1ab \mid S_2$,
  - $S_2 \rightarrow a$ is left-linear
  - $L(G_2)$ = ?
- Both G_1 and G_2 are regular grammars.

Example 3.14
a non-Regular Grammar

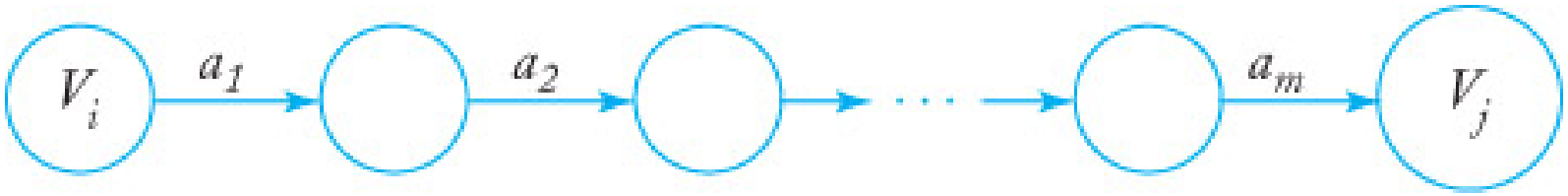
- $S \rightarrow A$
- $A \rightarrow aB \mid \lambda$
- $B \rightarrow Ab$

Right-Linear Grammars Generate RL - 1

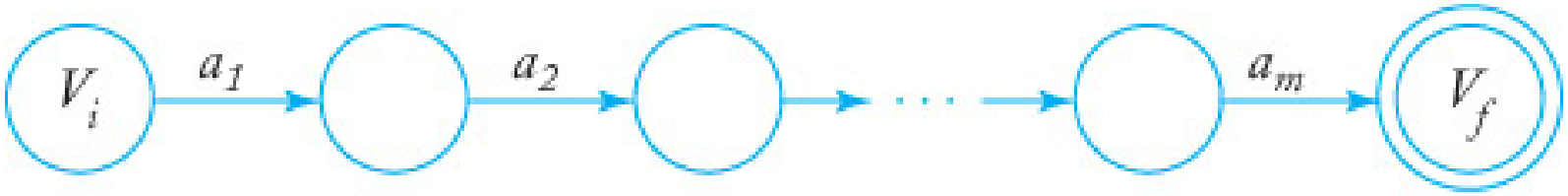


- Theorem 3.3 RG $\Rightarrow$ FA
  - Let G = (V, T, S, P) be a right-linear grammar. Then L(G) is a regular language.
  - Proof. Assume that V = $\{V_0, V_1, \ldots\}$, S = V_0
    - For each production, we convert it into a corresponding transition
    - We only have two possible forms of production in a right-linear grammar
    - Do you know which two?

Right-Linear Grammars Generate RL - 2



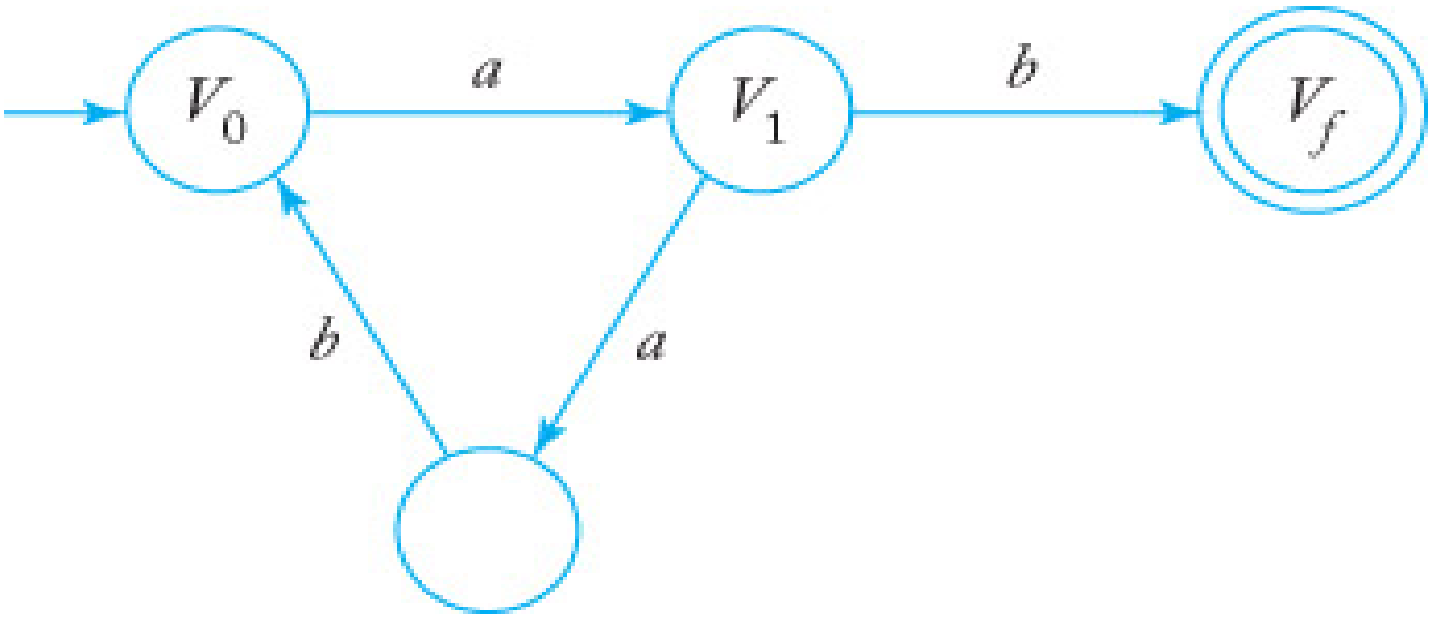
Represents $V_i \rightarrow a_1 a_2 \ldots a_m V_j$



Represents $V_i \rightarrow a_1 a_2 \ldots a_m$

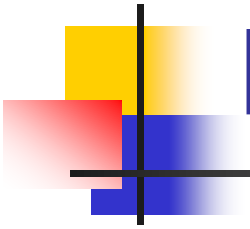
Example 3.15
Construct a FA that accepts the language generated by a RG

- $V_0 \rightarrow a\ V_1$
- $V_1 \rightarrow ab\ V_0 | b$



Right-Linear Grammars for Regular Language

- Theorem 3.4 FA $\Rightarrow$ RG
  - If L is a regular language on the alphabet Σ, then there exists a right-linear grammar G = (V, T, S, P) such that L = L(G).
  - Proof – constructive
    - Let M = (Q, Σ, δ, q_0, F) be a dfa that accepts L
    - Construct the right-linear grammar G with
      - V =$\{q_0, q_1, \ldots, q_n\}$
      - S = q_0
      - Construct a production for each transition



Equivalence of RL and RG

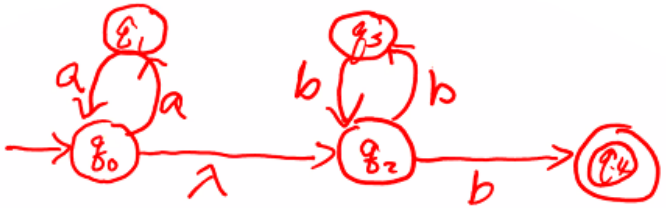
- Putting Theorems 3.4 and 3.5 together, we arrive at the equivalence of RL and RG
- Theorem 3.6:
  - A language L is regular if and only if there exists a regular grammar G such that L = L(G)

In-class Exercises

- Represent the following RL in 4 representations of RL (RE, TG, FA, RG):
  - Even number of a's followed by odd number of b's.

RE $(aa)^*(bb)^*b$

TG

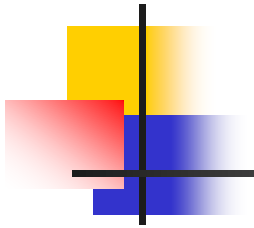


RG

$S \rightarrow aaA \mid B$

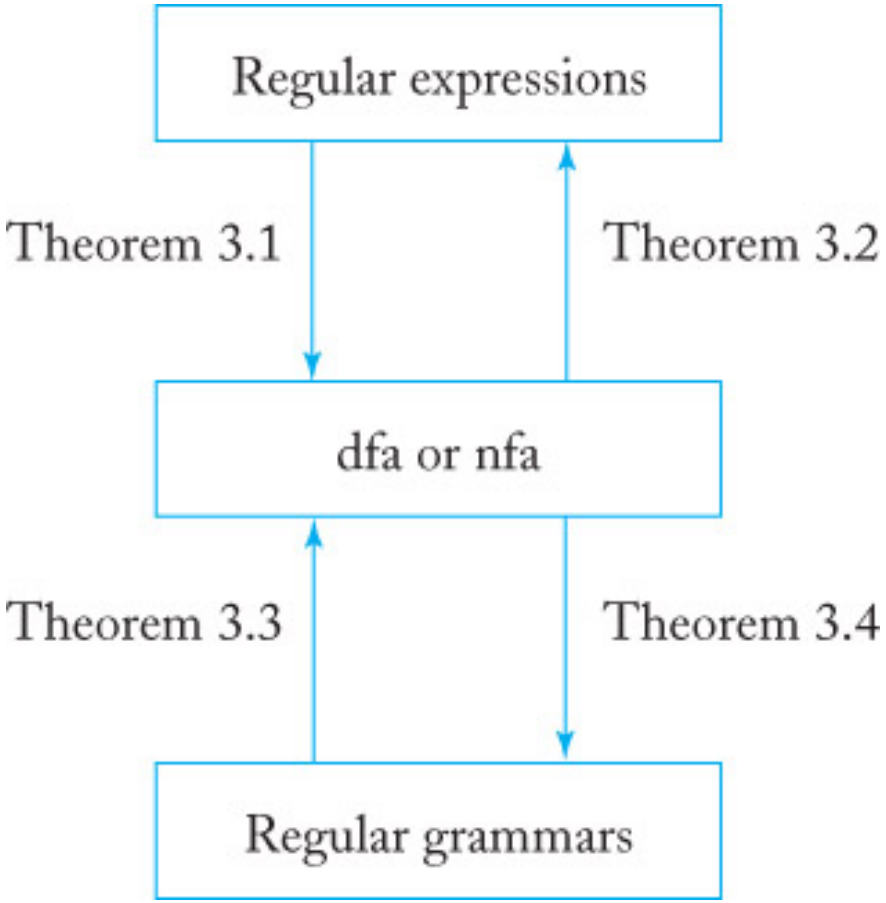
$B \rightarrow bbB \mid b$

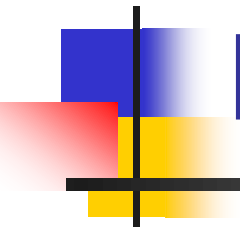
Summary-1



- We now have all 4 representations of RL:
  - FA, TG, RE, RG
- In some instance, one of them may be most suitable
- They are equally powerful in representing RL
- When you need all 4 representations for a given RL, you can start to work on one of the representation that is easiest to you first then ….

Summary-2


Regular expressions
Theorem 3.1
Theorem 3.2
dfa or nfa
Theorem 3.3
Theorem 3.4
Regular grammars

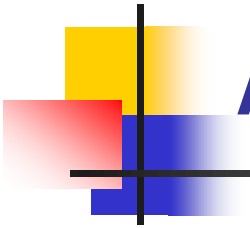


Chapter 4 Properties of Regular Languages

Closure Properties of RL

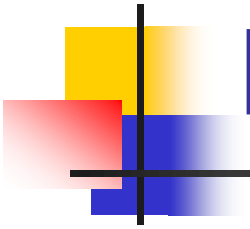
Elementary Questions about RL

Identifying Nonregular Languages



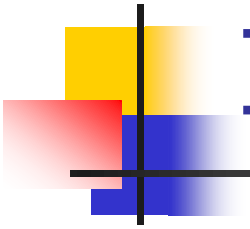
About RL …

- What is a regular language? Give a definition.
- Can you give the formal definition of RE?
- What is Kleen's Theorem?



RL Review

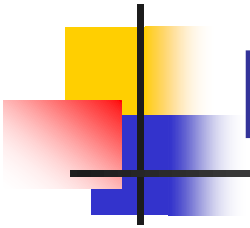
- RL – a language that can be defined by a RE or FA.
- RE
  - 1. $\varnothing$, λ, and $a \in \Sigma$ are RE's.
  - 2. If r_1 and r_2 are RE, so are $r_1 + r_2$, $r_1 . r_2$, r_1^* and (r_1).
- Kleen's Theorem: L(FA)=L(TG)=L(RE)



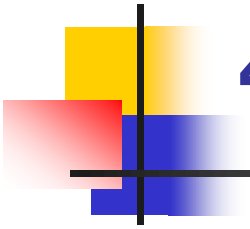
Issues on Regular Languages

- Set operations on RL – would that result another RL?
- Is a given language finite or not?
- Is every finite language regular?
- How can we tell whether a given language is regular or not?

Motivations to study properties of RL



- One way ***to show a language is not regular*** is to study properties that are shared by all RL
  - If we know some such property and we can show a candidate language does not have it, then we can tell that the language is not regular
- FA is also an powerful **algorithm** to recognize membership, equality, and more

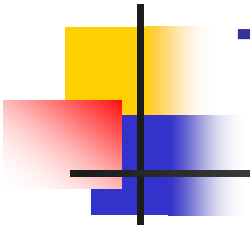


4.1 Closure Properties of RL

- Theorem 4.1
  - If L_1 and L_2 are RL, then so are $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1\ L_2$, L_1', and L_1^*. We say that the family of regular languages is closed under union, intersection, concatenation, complementation, and star-closure.

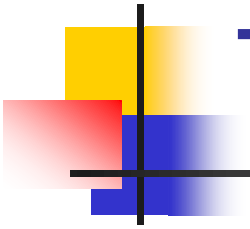
Theorem 4.1 Proof Outline - 1

- If L_1 and L_2 are RL, then there exist RE r_1 and r_2 such that $L_1=L(r_1)$ and $L_2=(r_2)$. By definition, we have
  - $L_1 \cup L_2$: $r_1 + r_2$
  - $L_1 L_2$: $r_1 \cdot r_2$
  - L_1^* : r_1^*



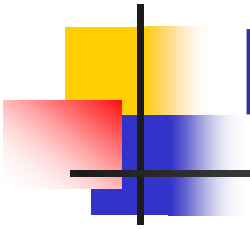
Theorem 4.1 Proof Outline - 2

- If L is a RL, then L’ is also a RL
- Proof – if L is a RL, then there is a FA M such that L = L(M). We can make a new FA M’ by
  - Change all final states of M to non-final states
  - Change all non-final states to final states
  - L(M’) = L’



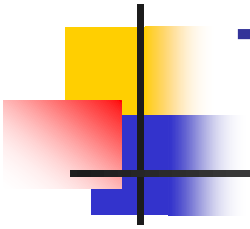
Theorem 4.1 Proof Outline - 3

- If L_1 and L_2 are RL, then $L_1 \cap L_2$ is also RL.
- Proof (key step)
  - $L_1 \cap L_2 = (L_1' + L_2')'$



Example 4.1

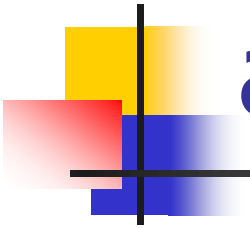
- Show that RL is closed under difference
  - If L_1 and L_2 are regular so are L_1- L_2
- Proof
  - L_1- L_2 = $L_1 \cap L_2'$
  - L_2 is RL implies L_2' is RL (Theorem 4.1)
  - Then, because closure of RL under intersection, we have $L_1 \cap L_2'$ is regular



Theorem 4.2

- The family of RL is closed under reversal.
- Proof (try to give a outline)
  - Use RE or FA?

4.2 Elementary Questions about Regular Languages



- Given a language L and a string *w*, can we determine whether or not w is an element of L?
- Need a membership algorithm
- Algorithm – a method for which one can write a computer program (informal)

Theorem 4.5
FA as membership algorithm

- Given a standard representation of any RL L on Σ and any $w \in \Sigma^*$, there exists an algorithm for determining whether or not w is in L.
- Proof. Represent L by a dfa, then test w to see if it is accepted by this automaton

Theorem 4.6

RL empty, finite, or infinity determination using FA

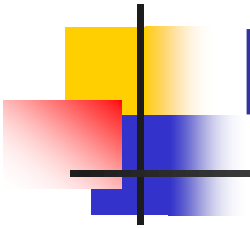
- There exists an algorithm for determining whether a regular language L, given in standard representation, is **empty, finite, or infinite.**
- Proof. Represent L as a dfa.
  - If there is a path from the initial vertex to any final vertex, then L is not **empty**.
  - Find all the vertices that are the base of a cycle. If any of these are on a path from an initial to final vertex, L is **infinite**. Otherwise, it is **finite**.

Theorem 4.7

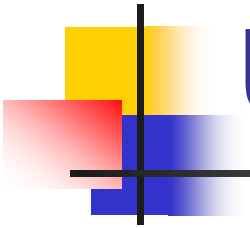
determine whether or not two RLs are equal

- Given two regular languages L_1 and L_2, there exists an algorithm to determine whether or not $L_1 = L_2$
- Proof
  - Using L_1 and L_2, we define the language
    - $L_3 = (L_1 \cap L_2') \cup (L_1' \cap L_2)$
    - By closure L_3 is regular and we can use the algorithm in Theorem 4.6 to determine if L_3 is empty
    - If L_3 is empty then $L_1 = L_2$

4.3 Identifying Nonregular Languages

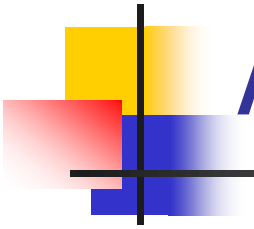


- How do you prove a language to be regular?
  - Constructive proof
- How do you prove a language to be nonregular?
  - Proof by negation
    - **Using the Pigeonhole Principle**
    - **A Pumping Lemma**



Using the Pigeonhole Principle

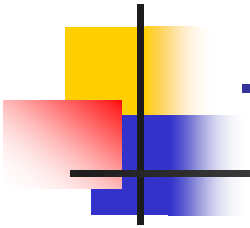
- **Case 1**. Finite number of pigeonholes, H, for finite number of pigeons, P.
  - When |H| < |P| and all of pigeons went into the pigeonholes, then at least one hole contains at least two pigeons
- **Case 2**. When |H| = N, and |P| = ∞, then at least one hole contains infinite number of pigeons
- Applying the principle to an FA that has limited memory and can accept an infinite RL.
  - Pumping Lemma = another form of the pigeonhole principle



A Pumping Lemma

- Let L be an infinite RL. Then there exists some positive integer m such that any $w \in L$ with $|w| \geq m$ can be decomposed as
  - $w = xyz$
  - with $|xy| \leq m$
  - and $|y| \geq 1$
  - such that $w_i = xy^iz$ is also in L for all i = 0, 1, 2, …

A Pumping Lemma
-- in other words



- Every sufficiently long string in L can be broken into three parts in such a way that an arbitrary number of repetitions of the **middle part** (y) yields another string in L. We say that the middle string is "**pumped**".
- w = xyz
  - $|w| \geq m$ (m is the integer in Pumping Lemma)
  - x or z can be empty but y must be nonempty
  - You can think m to be the number of the states of the FA that accepts L

Example 4.7
Show $L = \{a^n b^n : n \geq 0\}$ is not regular

- Proof.
  - **Assume** that L is regular, so that the pumping lemma must hold. **Let m** be the integer in the pumping lemma.
    - **Let w** = $a^m b^m$ ($|w| \geq m$)
    - By the pumping lemma, w may be written as xyz, where $|y| \geq 1$ and $|xy| \leq m$ and xy^iz is also in L for all i = 0, 1, 2, …
    - **Let i** = 2, $xy^2z = a^{m+k}b^m \notin L$ with $1 \leq k \leq m$
    - This string brings at least one more a into the a's segment, and is not a string in L. **This contradicts** the pumping lemma and thereby indicates that the assumption that L is regular must be false.

In applying the Pumping Lemma…

- The correct argument can be viewed as a **game** we play against an opponent
- Our goal is to win the game by establishing a contradiction of the pumping lemma, while the opponent tries to foil us. There are **4 moves**:
  - The opponent picks m
  - Given m, **we pick w** in L with $|w| \geq m$
  - The opponent chooses the decomposition xyz, subject to $|xy| \leq m$, $|y| \geq 1$
  - **We pick i** in such a way that the pumped string w_i, is not in L. If we can do so, we win the game

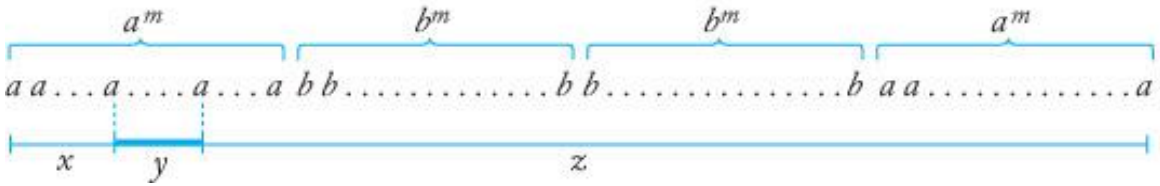
Key elements

apply Pumping Lemma to prove L is non-regular

1. Assume L is regular and m is integer in Pumping Lemma
2. Let w = f(m) in L and |w|≥m
3. W =xyz, subject to $|xy| \leq m$, $|y| \geq 1$
4. Let i = ? to show resulting Wi make Pumping property fail, and therefore a contradiction. Conclusion: L is not regular

Example 4.8
show L = {ww^R: $w \in \Sigma^*$} is not regular

- Proof.
  - Assume that L is regular, so that the pumping lemma must hold. Let m be the integer in the pumping lemma. Let
    - $w = a^m b^m b^m a^m$ ($|w| \geq m$) -- **step 2**
    - Because this choice of w, in **step 3,** y can only consists of a's; in **step 4**, we may use i = 0 and the contradiction follows – the resulting string has fewer a's on the left.



Key steps:

pick the w and i to make the argument easier for us

- Ex. 4.9 $L = \{w \in \Sigma^*, n_a(w) < n_b(w)\}$
  - $w = a^m b^{m+1}$ $i = 2$, $w_2 = ?$ for $y = a^k$
- Ex. 4.10 $L = \{(ab)^n a^k : n > k, k \geq 0\}$
  - $w = (ab)^{m+1} a^m$ $i = 0$, $w_0 = ?$ for all possible y's
- Ex. 4.11 $L = \{a^n : n \text{ is a perfect square}\}$
  - Pick $n = m^2$ $i = 0$, $w_0 = ?$ for $y = a^k$
  - You may try $i = 2$, and it works too

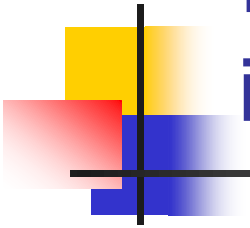
Example 4.13

For $L = \{a^n b^l : n \neq l\}$, there is a better way than the Pumping Lemma to show L is not regular

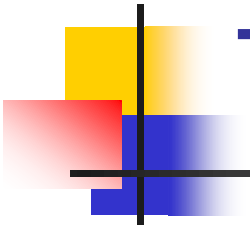
- Suppose L is regular. Then by Theorem 4.1, L′ is also a RL, and
  - $L_1 = L' \cap L(a^*b^*)$ would also regular.
  - But $L_1 = \{a^n b^n : n \geq 0\}$ is nonregular.
  - Consequently, L cannot be regular.

  - Note: this is also "prove by contradiction"

More examples/exercises
in proving a language to be non regular

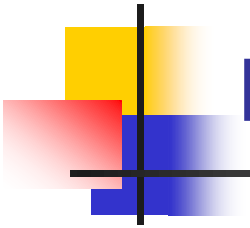


- PALINDROME
- PRIME



The Pumping Lemma is …

- Difficult to understand
- Easy to make mistakes when applying it
- Common mistakes
  - Using it to show a language is regular
  - Pick w is not in L or not easy to argue
  - Mix up m and i
  - Make some assumption about the decomposition xyz



To apply the Pumping Lemma well:
Knowledge of the rule + a good strategy

- Knowledge of the rules is essential, but that alone is not enough to play a good game
- Need a good strategy to win
- Read more examples + do more exercises will help – just like playing any games!

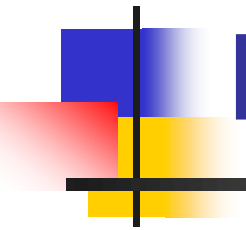
The Pumping Lemma: Poem

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

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

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

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

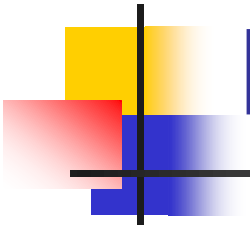


Chapter 5 Context-Free Languages

Context-Free Grammars (CFG)

Parsing and Ambiguity

CFG and Programming Languages



Needs for CFG

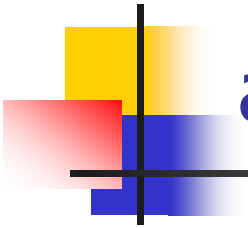
- RL is effective in describing certain simple pattern
- Regular expressions were inadequate to define all interesting languages, such as (()), nested structure in programming languages
- CFL has important applications in the design of programming languages as well as in the construction of efficient compilers

5.1 Context-Free Grammars

Definition 5.1

- A grammar G = (V, T, S, P) is said to be **context-free** if all productions in P have the form
  - $A \rightarrow x$
  - where $A \in V$ and $x \in (V \cup T)^*$.
- A language L is said to be context-free if and only if there is a CFG G such that L = L(G)

Grammar Set Notations:
a concise way of design description



G = (V, T, S, P)

T – a set of terminal symbols
V – a set of nonterminal symbols
S – a element of V, starting symbol
P – a set of production rules with certain format restrictions

Write out the production rule format in set notation for each of the following grammars :

- RG
- CFG

Context Free Grammar (CFG): the representation defining CFL

- A grammar is said to be context-free if all productions in P have the form
  - $A \rightarrow x$
  - English description:
    - where A is a single nonterminal and x can be any string of terminals or/and nonterminals
    - Single nonterminal of the left handside of the production rule makes the grammar context free
  - Set notation:
    - where $A \in V$ and $x \in (V \cup T)^*$

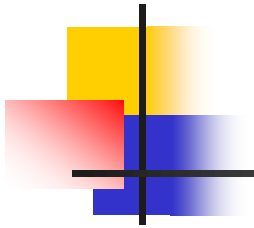
Regular Grammars (RG):
one of 4 representations of RL

- A subset of CFG
- A CFG is said to be regular grammar if all productions in P have the form
  - $A \rightarrow xB$, $A \rightarrow x$; or $A \rightarrow Bx$, $A \rightarrow x$
  - English:
    - Where A is a single nonterminal and x can be word (string of terminals) or semiword (a string of terminals ended or starting with a nonterminals
  - Set notation:
    - Where $A, B \in V$, and $x \in T^*$

Example 5.1
G = ({S}, {a, b}, S, P}

- is context-free with productions
  - $S \rightarrow aSa$
  - $S \rightarrow bSb$
  - $S \rightarrow \lambda$
- A derivation in this grammar is
  - $S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabbaa$.
  - We can derive that $L(G) = \{ww^R : w \in \{a, b\}^*\}$.
  - The language is context-free, but is not regular (as shown in Example 4.8)

Example 5.2



- The grammar G is context-free with productions
  - $S \rightarrow abB$
  - $A \rightarrow aaBb$
  - $B \rightarrow bbAa$
  - $A \rightarrow \lambda$
- Show that
  - $L(G) = \{ab(bbaa)^n bba(ba)^n : n \geq 0\}$.

Example 5.3 -- 1

To show L = $\{a^n b^m : n \neq m\}$ is context-free

- We need to produce a CFG for L
- We know how to produce a CFG with n = m
  - Write out
- Now we need to add 2 sets of productions
  - For the case n > m we add the first set
    - $S \rightarrow AS_1$
    - $S_1 \rightarrow a\ S_1 b | \lambda$
    - $A \rightarrow aA | a$

Example 5.3 -- 2

To show L = $\{a^n b^m : n \neq m\}$ is context-free

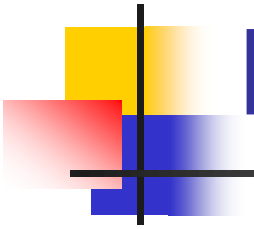
- For the case n < m, we add another similar set, and we get the answer
  - $S \rightarrow AS_1 | S_1B$
  - $S_1 \rightarrow a\ S_1 b | \lambda$
  - $A \rightarrow aA | a$
  - $B \rightarrow bB | b$
- The resulting grammar is context-free, hence L is CFL.

Example 5.4

programming language L_2 include strings such as (()) and ()()()

- CFL L can be generated by CFG with productions
  - $S \rightarrow aSb|SS|\lambda$
  - Substitute a with (and b with) in the productions, we can generate L_2 above
  - $L = \{w \in \{a, b\}^* : n_a(w) = n_b(w)$ and $n_a(v) \geq n_b(v)$, where v is any prefix of w$\}$

Leftmost and Rightmost Derivations



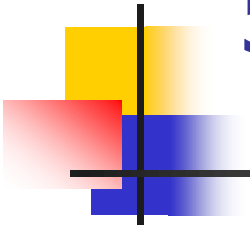
- Definition 5.2
  - A derivation is said to be **leftmost** if in each step the leftmost variable in the sentential form is replaced. If in each step the right most variable is replaced, we call the derivation **rightmost**.

Example 5.5

Leftmost and Rightmost Derivations

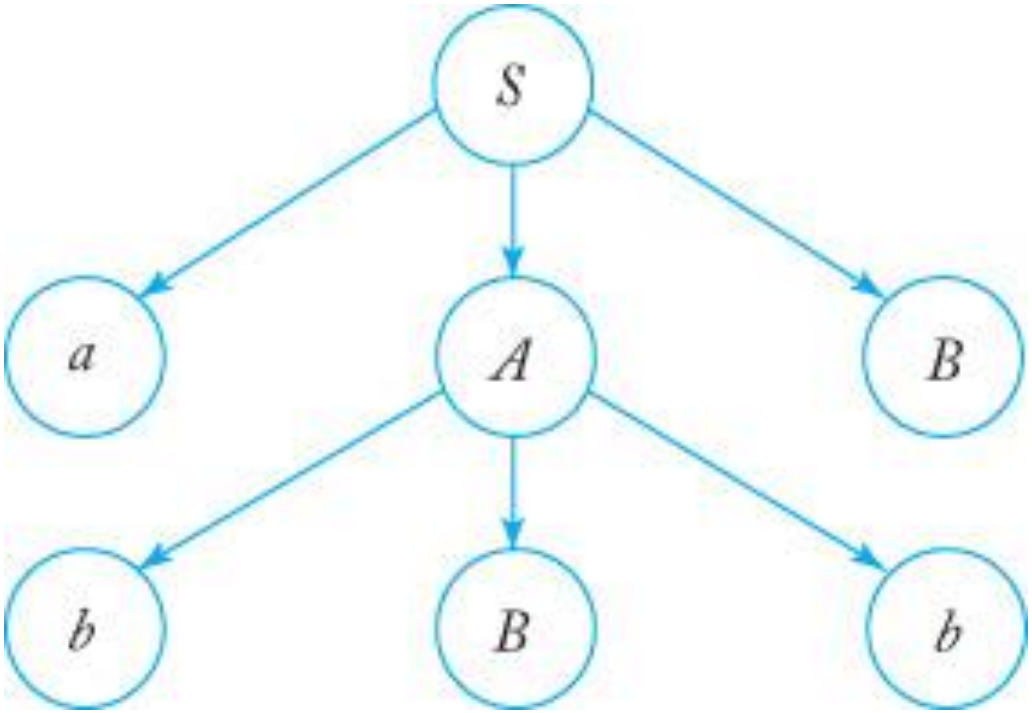
- $S \rightarrow aAB$
- $A \rightarrow bBb$
- $B \rightarrow A|\lambda$
  - Leftmost derivation of abbbb
    - $S \Rightarrow aAB \Rightarrow abBbB \Rightarrow abAbB \Rightarrow abbBbbB \Rightarrow abbbbB \Rightarrow abbbb$
  - Rightmost derivation of abbbb
    - $S \Rightarrow aAB \Rightarrow aA \Rightarrow abBb \Rightarrow abAb \Rightarrow abbBbb \Rightarrow abbbb$

5.2 Derivation Trees/Parse Trees

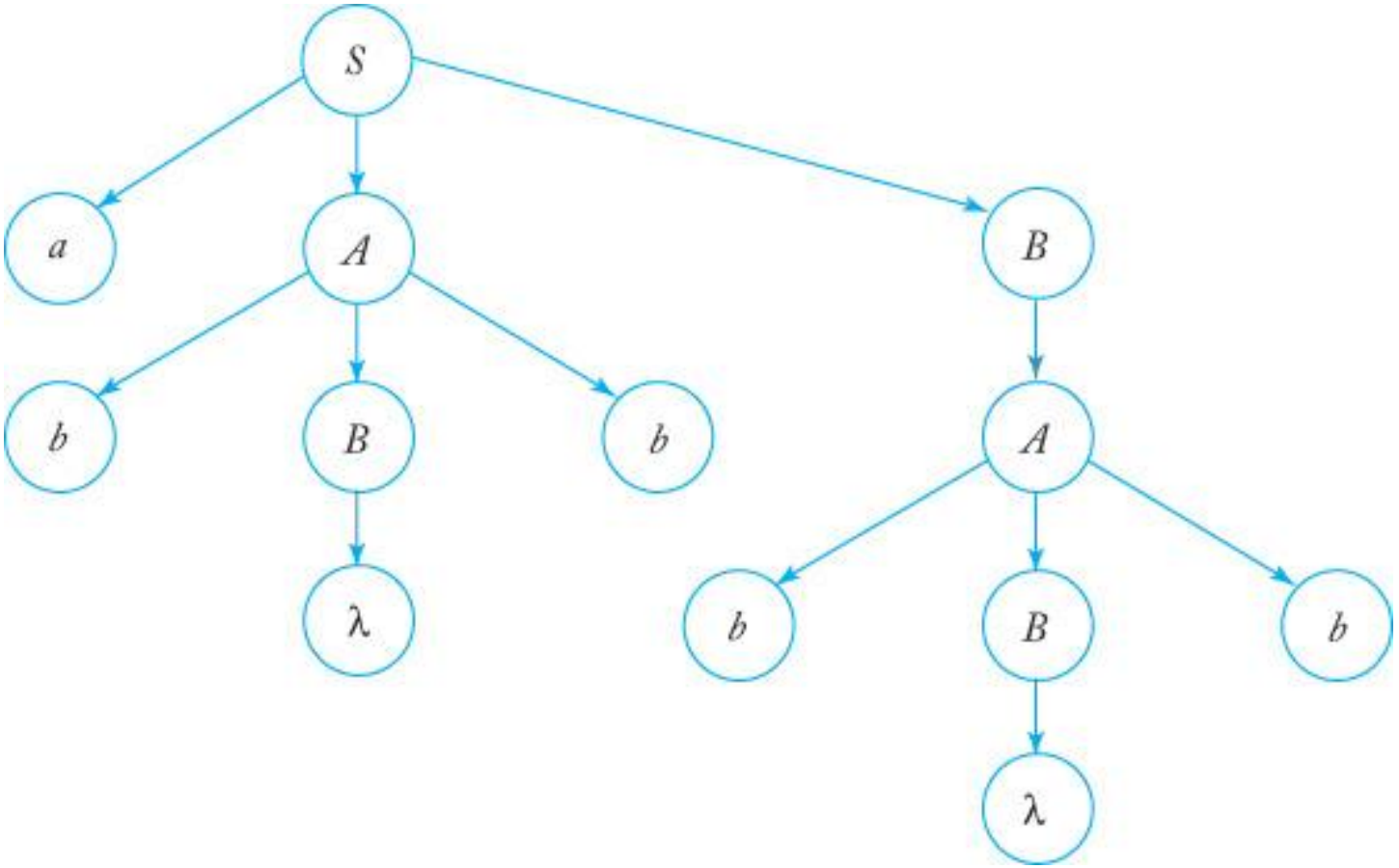


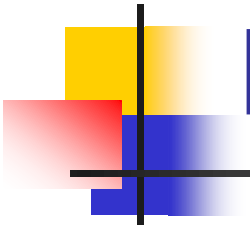
- A second way of showing derivations, independent of the order in which productions are used
- Definition 5.3 (page 130, a formal definition of **derivation tree** and **partial derivation tree**)

Example 5.6: Partial derivation tree $S \rightarrow aAB$, $A \rightarrow bBb$, $B \rightarrow A|\lambda$



Example 5.6: a complete Derivation tree
$S \rightarrow aAB,\ A \rightarrow bBb,\ B \rightarrow A|\lambda$



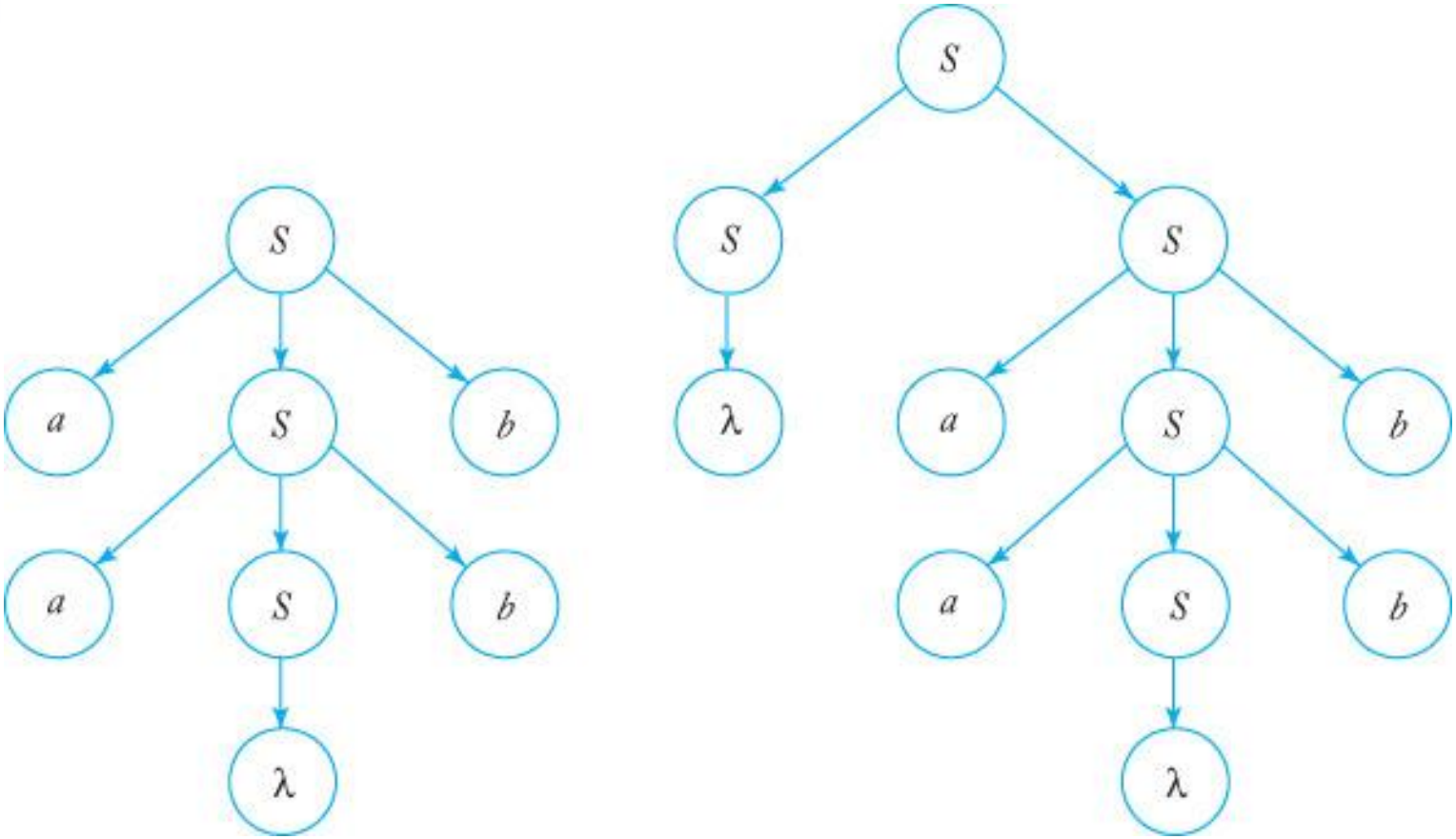


Parsing and Ambiguity

- **Parsing** – finding a sequence of productions by which a w $\in$ L(G) is derived
- Definition 5.5
  - A CFG G is said to be **ambiguous** if there exists some w $\in$ L(G) that has at least two distinct derivation trees. Alternatively ambiguity implies the existence of two or more leftmost or rightmost derivations

Example 5.10

Grammar $S \rightarrow aSb|SS|\lambda$, is ambiguous since aabb has the two derivation trees shown



Example 5.11 -- 1

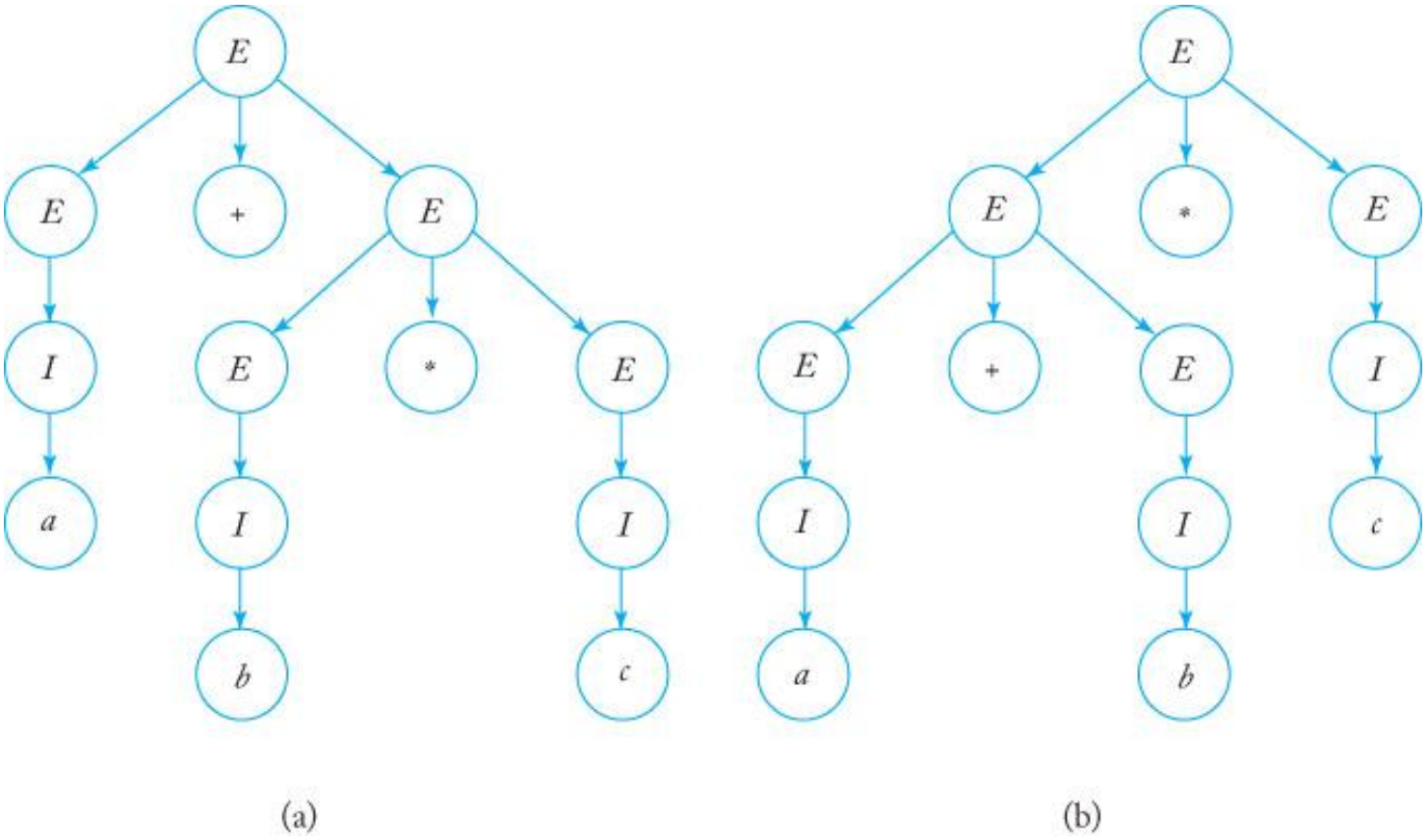
G= (V, T, E, P) with
V= {E, I}, T = {a, b, c, +, *, (,)}

- and productions
  - E $\rightarrow$ I | E+E | E*E | (E)
  - I $\rightarrow$ a | b | c
- The grammar is ambiguous since a+b*c has two different derivation trees
  - Try to draw the two different derivation trees

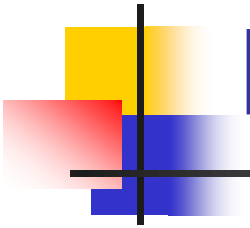
Example 5.11 – 2

$E \rightarrow I \mid E+E \mid E*E \mid (E)$, $I \rightarrow a \mid b \mid c$

a+b*c has two different derivation trees



Example 5.12 -- 1
Rewrite the grammar



- One way to resolve the ambiguity is to associate precedence rules with the operators + and *
- * has higher precedence than +
  - + is one level closer to root of the derivation tree, E
- We introduce new variables
  - V = {E, T, F, I}
  - Replace productions in such a way that + is closer to E than * in the derivation tree

Example 5.12 – 2
Rewrite the productions

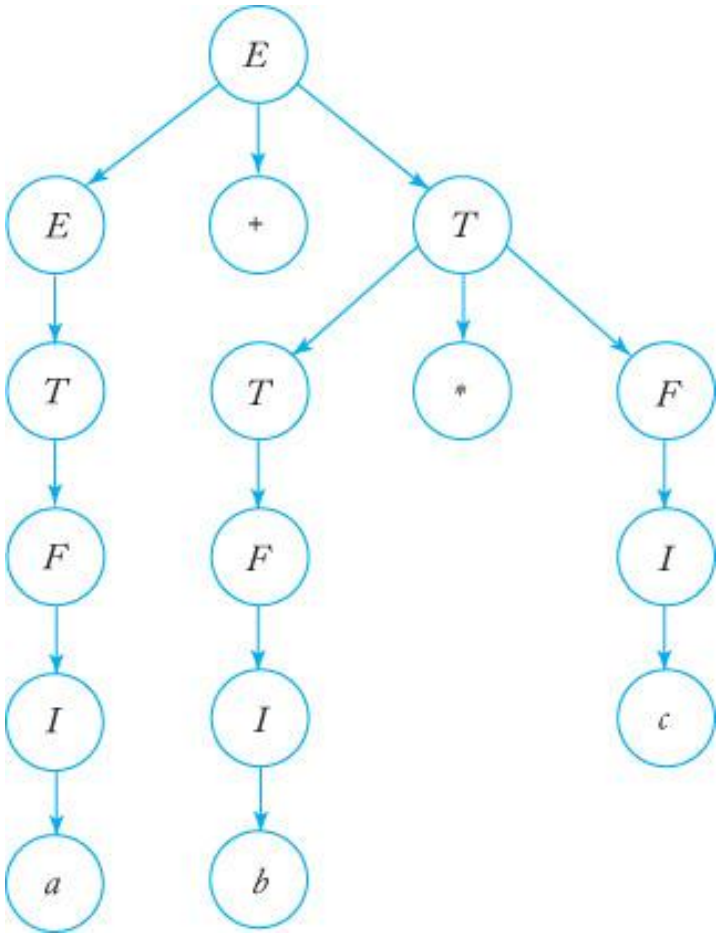
- ambiguous
- $E \rightarrow I \mid E+E \mid E*E \mid (E)$
- $I \rightarrow a \mid b \mid c$

Rewrite to remove ambiguity

- $E \rightarrow E + T \mid T$
- $T \rightarrow T * F \mid F$
- $F \rightarrow (E) \mid I$
- $I \rightarrow a|b|c$

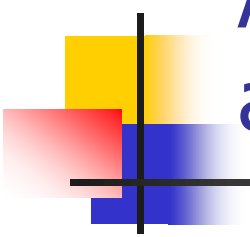
Example 5.12 – 3:

a unique derivation tree of a+b*c (using new grammar)



A Practical Problem

Assume the following rules of associativity and precedence for expressions



- Precedence:
  - Highest *, /, **not**
  - +, -, &, **mod**
  - - (unary)
  - =, ≠, <, ≤, ≥, >
  - **and**
  - Lowest **or**, **xor**
- Associativity: left to right
- Write a CFG for the expression. Assume the only operands are the names a, b, c, d, and e.

Recursion to specify associativity
Precedence cascade to specify precedence

<expr> ::= <expr> or <e1> | <expr> xor <e1> | <e1>

<e1> ::= <e1> and <e2> | <e2>

<e2> ::= <e2> = <e3> | <e2> /= <e3> | <e2> < <e3>
| <e2> <= <e3> | <e2> > <e3> | <e2> >= <e3> | <e3>

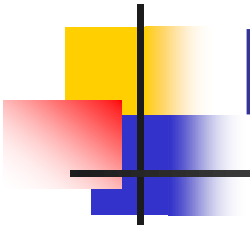
<e3> ::= <e4> | -<e4>

<e4> ::= <e4> + <e5> | <e4> - <e5> | <e4> & <e5> | <e4>
mod <e5> | <e5>

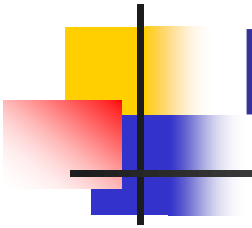
<e5> ::= <e5> * <e6> | <e5> / <e6> | not <e5> | <e6>

<e6> ::= a | b | c | d | e | const | (<expr>)

CFG and Programming Languages



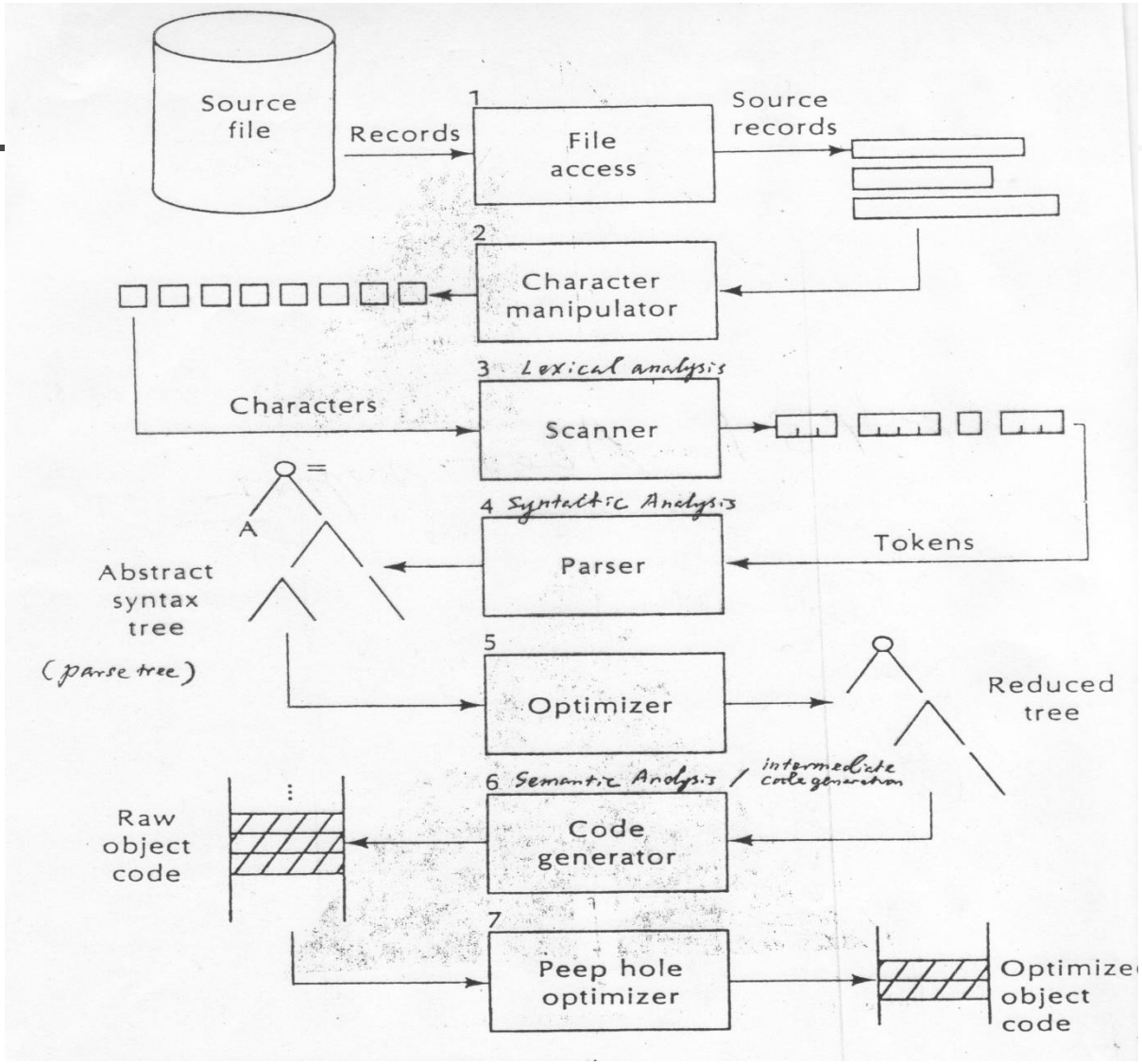
- One of the most important uses of the computing theory is in
  - the definition of programming language
  - the construction of interpreters/compilers
  - Both RL and CFL are used in model all aspects
  - We can define a programming language by a grammar



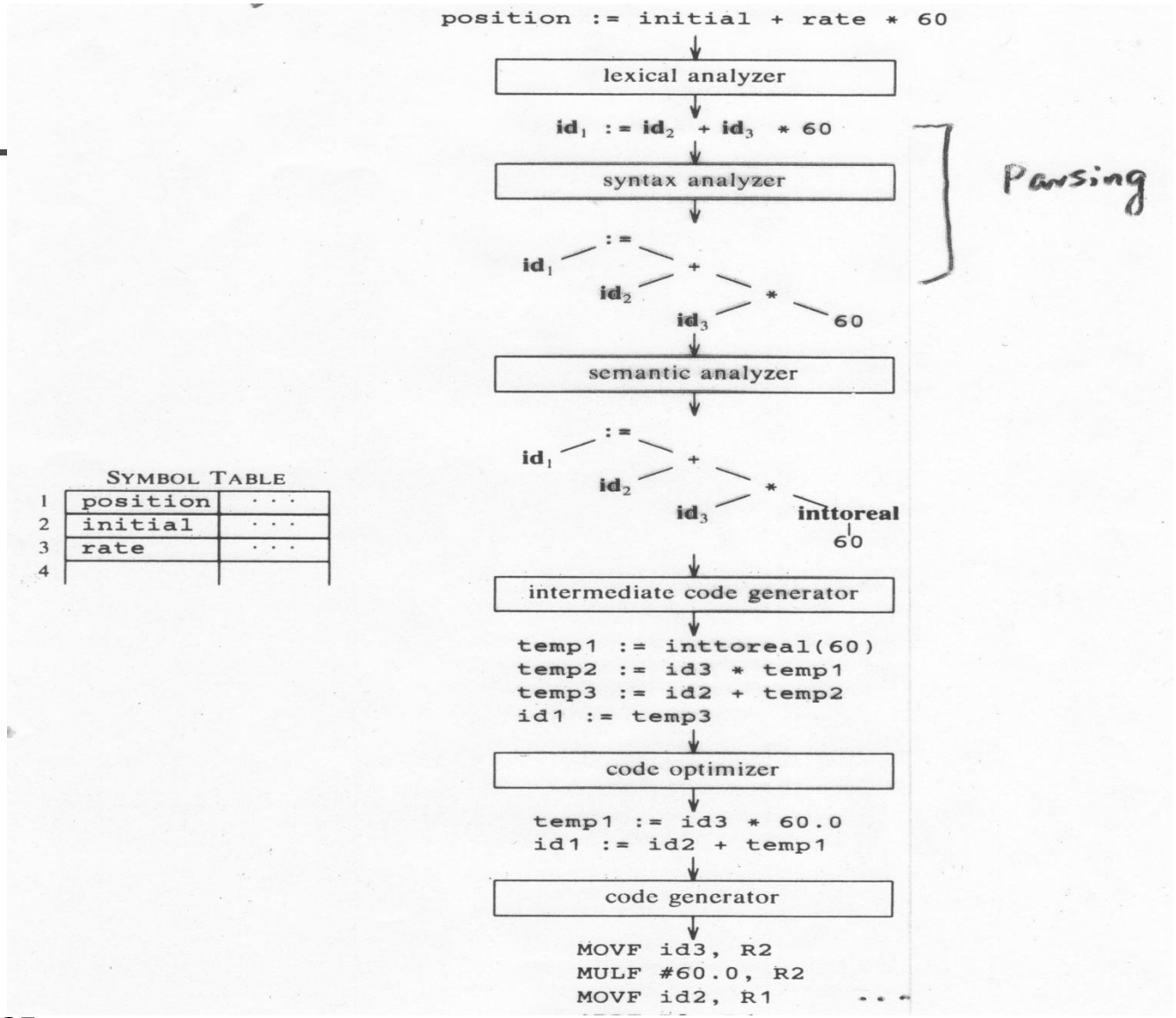
BNF and CFG

- A traditional notation in writing on programming languages is called the Backus-Naur form or BNF
- BNF is in essence the same as notation we use for CFG here
- <expression>::= <term>|<expression>+<term>
- <term>::= <factor>|<term>*<factor>
- <while_statement>::=while<expression><statement>

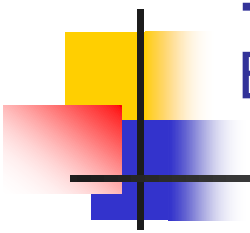
Major Operations in a Compiler



The Phases of a Compiler



In-class exercise: for given grammar, show BNF, EBNF, Syntax Diagram, first and follow sets



- Write G_3 in EBNF
- Draw syntax diagrams
- What are the 2 requirements for a predictive parser to be able to distinguish between choices in the grammar rule?
- Give the first sets/follow sets of G_3.
- Do we need to compute follow set for G_3?

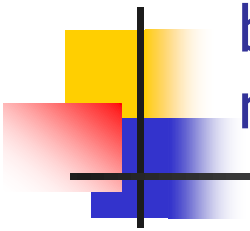
G_3 in BNF → EBNF -- 1

- To remove recursion and ε
- V → SR$
- S → + | - |ε
- R → .dN | dN.N
- N → dN | ε

BNF → EBNF -- 2

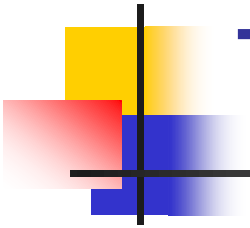
- To remove recursion and ε
- V → SR$ — V ::= SR$
- S → + | - |ε — S ::= [+ | -]
- R → .dN | dN.N — R ::= .dN | dN.N
- N → dN | ε — N ::= {d}

- Draw syntax diagrams for each variable
- Can we simplify EBNF more?



What are the 2 requirements for a predictive parser to be able to distinguish between choices in the grammar rule? (Hint: use first and follow sets)

1. Given A → B | C
2. Given A → D[E]F



Two Requirements

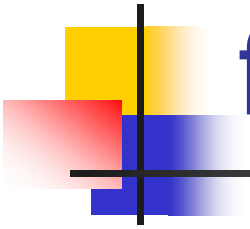
(1) Given A → B | C

First (B) ∩ first (C) = ∅

(2) Given A → D[E]F

First (E) ∩ follow (E) = ∅

EBNF to Syntax Diagram first sets and follow sets -1

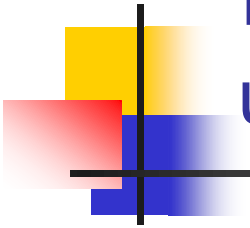


- V ::= SR$
- S ::= [+ | -]
- R ::= .dN | dN.N
- N ::= {d}

first sets and follow sets - 2

- V ::= SR$ — first (V)=first (S) ∪ first (R) = {+, -, ., d}
- S ::= [+ | -] — first (S) = {+, -}
- R ::= .dN | dN.N — first (R) ={. , d}
- N ::= {d} — first (N) ={d}

- Follow (V) = {}, Follow (R) ={$}
- Follow (S) = first (R) = {., d}
- Follow (N) = {.} ∪ follow (R) = {., $}



EBNF

using substitution to simplify

- V ::= SR$
- S ::= [+ | -]
- R ::= .dN | dN.N
- N ::= {d}

- V ::= [+|-] R $
- R ::= .d{d} |d{d}.{d}

A Top-down Parser - 1

$G_3 = (\{V, S, R, N\}, \{+, -, ., d, \perp\}, P, V)$, where P is:

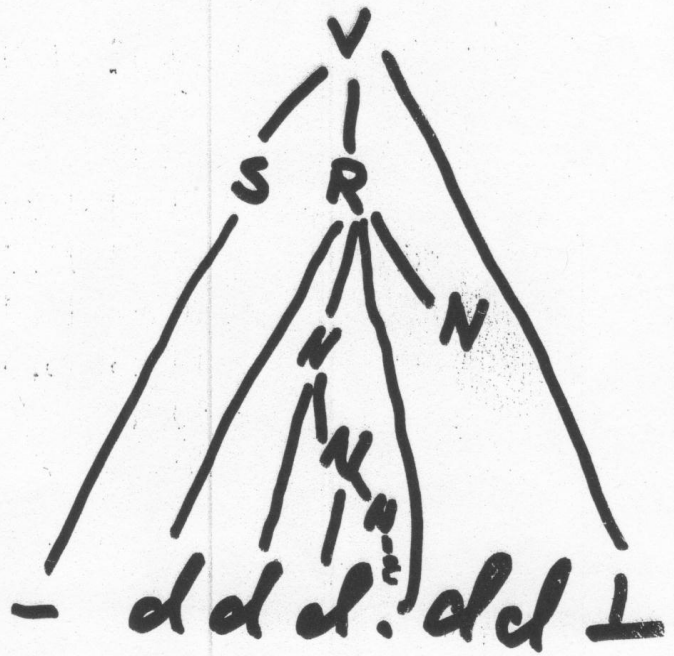
1. V → SR⊥ {⊥ is a stop symbol}
2. S → +
3. S → -
4. S → ϵ {ϵ is the empty string}
5. R → .dN {d is a decimal digit}
6. R → dN.N
7. N → dN
8. N → ϵ

Left-most Exposed Nonterminal	Next token: +	−	.	d	⊥
V	1	1	1	1	×
S	2	3	4	4	×
R	×	×	5	6	×
N	×	×	8	7	8

Figure 2.16. A top-down LL(1) parsing table for grammar G_3.

A Top-down Parser - 2

Frontier	Remaining Input	Production
V	−ddd.dd⊥	1
SR⊥	−ddd.dd⊥	3
−R⊥	−ddd.dd⊥	(match, drop −)
R⊥	ddd.dd⊥	6
dN.N⊥	ddd.dd⊥	(match)
N.N⊥	dd.dd⊥	7
dN.N⊥	dd.dd⊥	(match)
N.N⊥	d.dd⊥	7
dN.N⊥	d.dd⊥	(match)
N.N⊥	.dd⊥	8
.N⊥	.dd⊥	(match)
N⊥	dd⊥	7
dN⊥	dd⊥	(match)
N⊥	d⊥	7
dN⊥	d⊥	(match)
N⊥	⊥	8
⊥	⊥	(match and halt)



A Bottom-up Parser

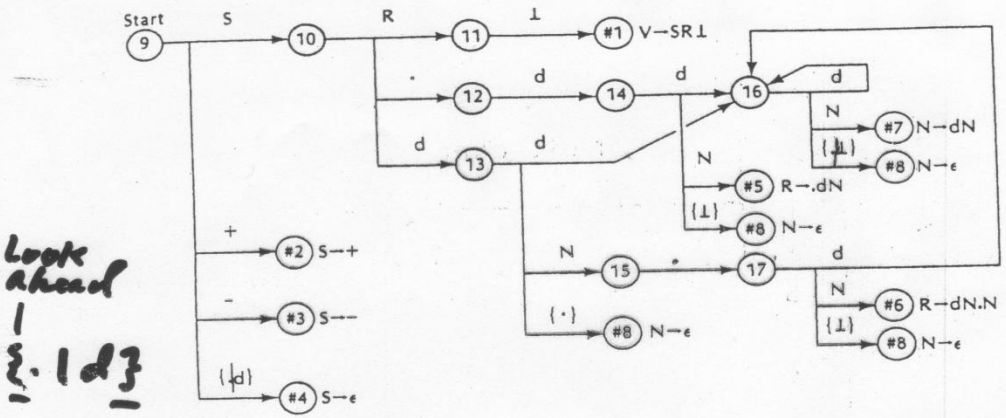


Figure 2.17. A bottom-up parsing machine for grammar G_3.

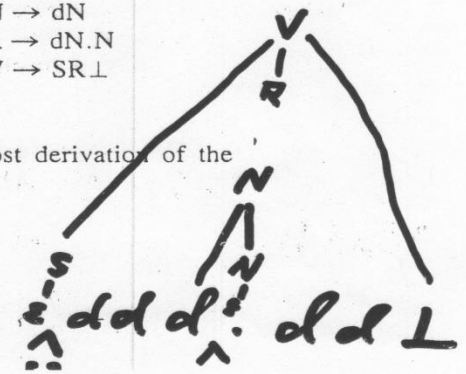
LR(1)

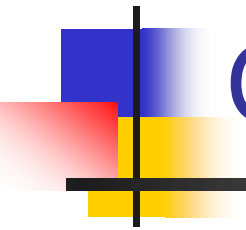
Rootline	State Path	Production
ddd.dd⊥	9	4, S → ϵ
Sddd.dd⊥	9, 10, 13, 16, 16	8, N → ϵ
SdddN.dd⊥	9, 10, 13, 16, 16	7, N → dN
SddN.dd⊥	9, 10, 13, 16	7, N → dN
SdN.dd⊥	9, 10, 13, 15, 17, 16, 16	8, N → ϵ
SdN.ddN⊥	9, 10, 13, 15, 17, 16, 16	7, N → dN
SdN.dN⊥	9, 10, 13, 15, 17, 16	7, N → dN
SdN.N⊥	9, 10, 13, 15, 17	6, R → dN.N
SR⊥	9, 10, 11	1, V → SR⊥
V	(halt)	

It should be clear that we have reproduced a right-most derivation of the sentence, in reverse order.

HW

-d.d⊥ , +d.dd⊥



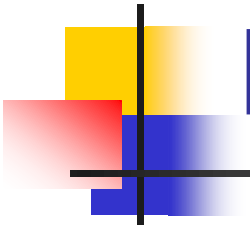


Chapter 7 Pushdown Automata

Nondeterministic Pushdown Automata

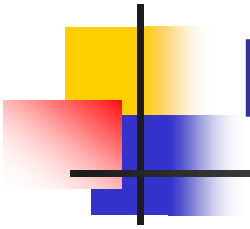
PDA and CFL

DPDA and DCFL



Pushdown Automata (PDA)

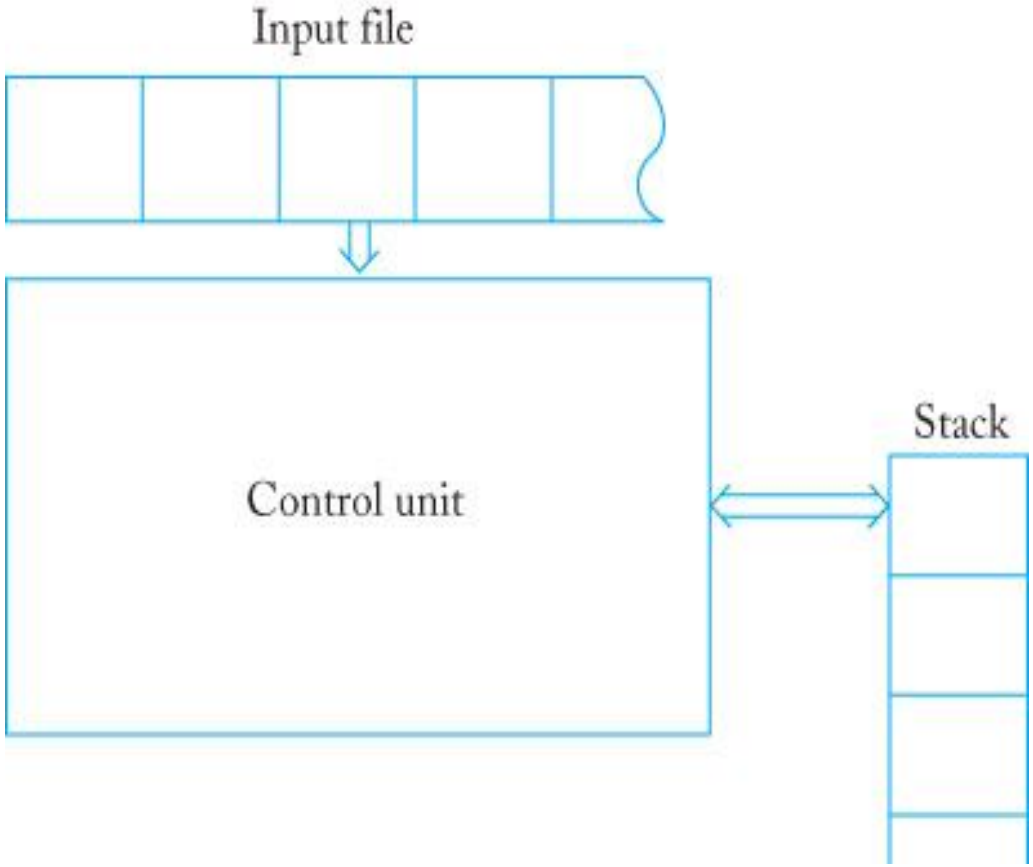
- Automata: FA, PDA, TM…
- Grammar: RG, CFG, CSG …
- Language: RL, CFL, CSL …
  - PDA is the model of nondeterministic top-down parser
  - For CFL
    - CFG – generator
    - PDA - recognizer



Difference between FA and PDA

- An informal definition of PDA
  - An input string
  - A read head to exam input once cell at a time
  - A finite state machine to control moves
  - LIFO push down stack
- DPDA is not equivalent to NPDA

Memory makes PDA a more powerful recognizer/accepter



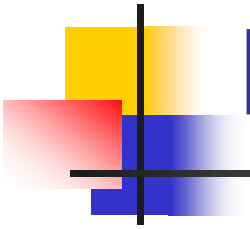
Definition 7.1

A nondeterministic pushdown accepter (npda) is defined by the septuple

- $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$
- Q, Σ, δ, q_0, and F are the same as they are in FA
- Γ is a finite set of symbols called the stack alphabet
- $z \in \Gamma$ is the stack start symbol
- δ: $Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow$ finite subset of $Q \times \Gamma^*$

Example 7.1
sample NPDA transition rule

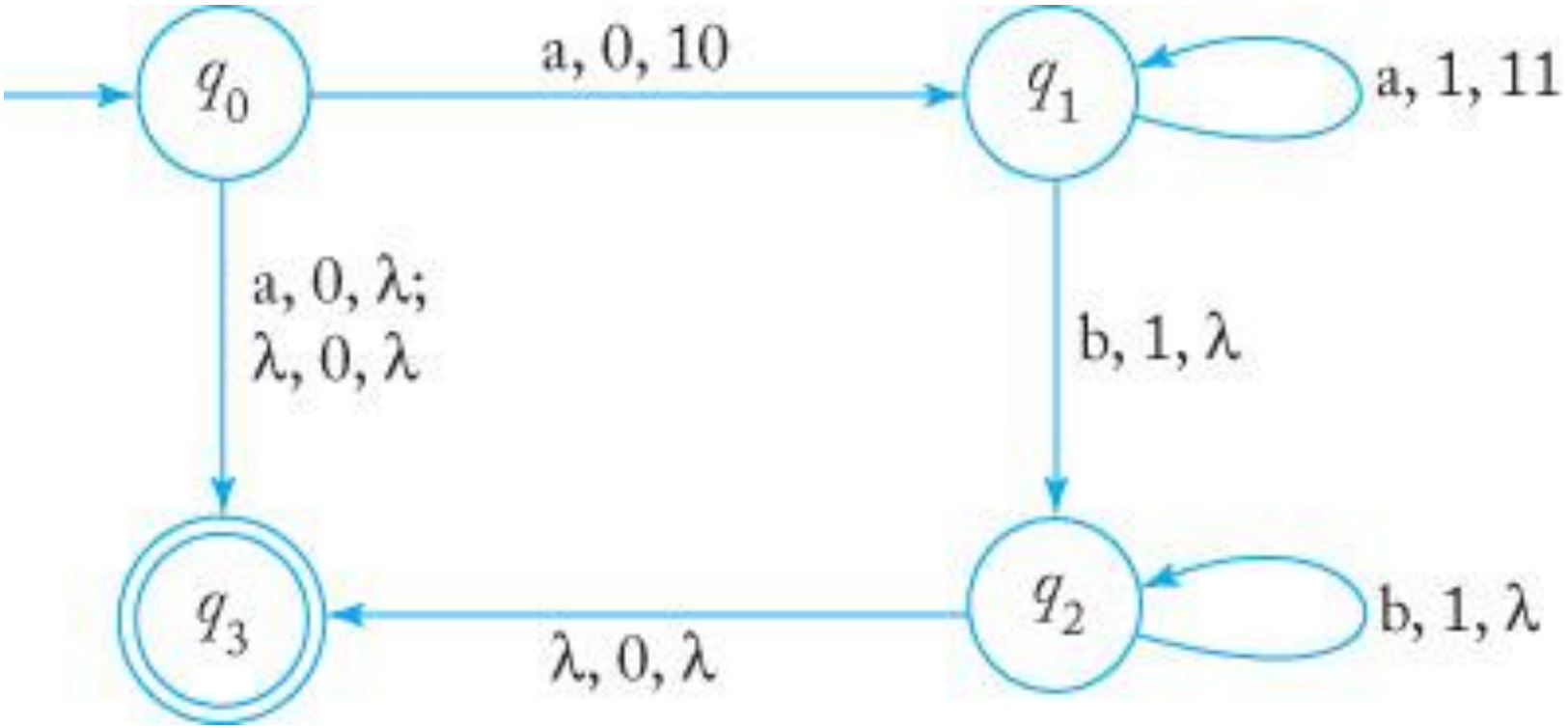
- $\delta\ (q_1, a, b) = \{(q_2, cd), (q_3, \lambda)\}$
- What does this transition rule mean?
  - Two things can happen when in q_1, read a, and top of the stack is b
    1. Move to q_2, and the string cd replaces b
    2. Move to q_3, and pop b



NPDA Examples

- FA $\rightarrow$ NPDA
  - Every FA can be transformed to a PDA
- $\{a^n b^n, n \geq 0\}$
  - Can you design an algorithm for this PDA?
- **Example 7.2**. $L = \{a^n b^n, n \geq 0\} \cup \{a\}$
  - PDA design and implementation
    - Design an algorithm first
    - Present your design in transition function, TG, or in picture notation (in Cohen book)

Example 7.3
TG for npda in example 7.2



The Language Accepted by a PDA

- $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$

  - $L(M) = \{w \in \Sigma^* : (q_0, w, z) \rightarrow^*_M (p, \lambda, u), p \in F, u \in \Gamma^*\}$

  - Language accepted by M is the set of all strings that can put M into a final state at the end of the string.

More examples of NPDA -1

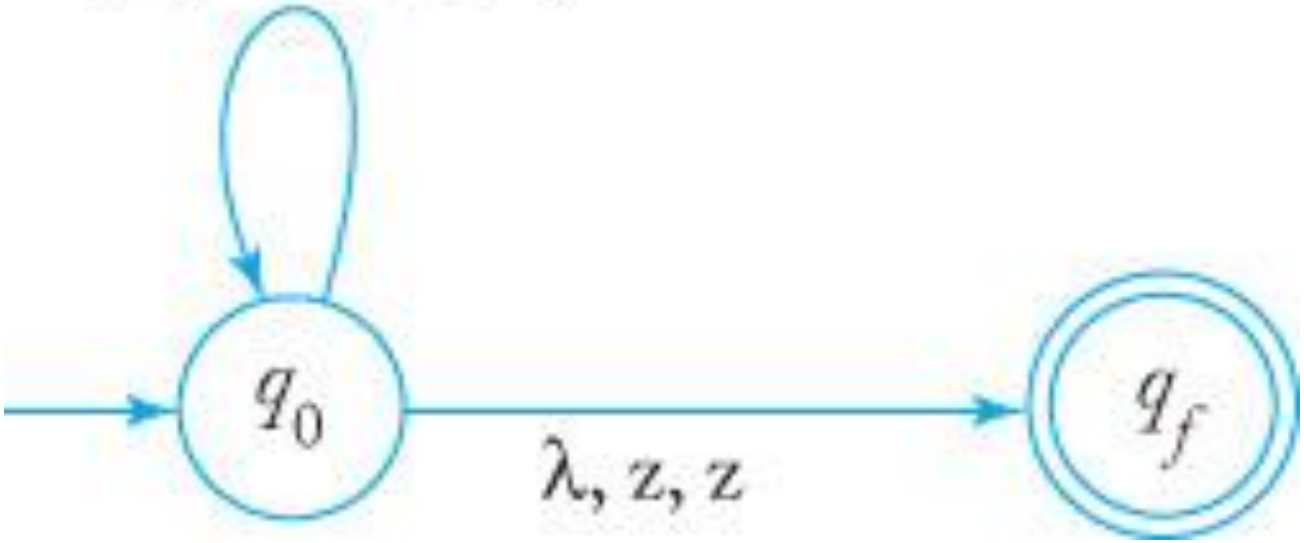
- DPDA for L = $\{wXw^R\}$ $w \in (a+b)^*$
  - Algorithm design in English
- NPDA for ODDPALINDROME
  - Algorithm design…
- NPDA for EVENPALINDROME
    - Example 7.5 Linz
    - Focus on its algorithm design first!
    - Try to draw the TG

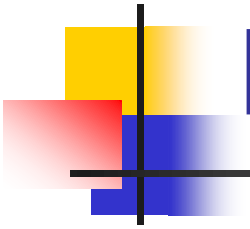
More examples of NPDA -2

- Example 7.4
  - Construct a npda for the language
  - $L = \{w \in \{a, b\}^* : n_a(w) = n_b(w)\}$
    - Algorithm design for this npda
      - Counter 0 is pushed into stack for each a read
      - Negative counter 1 for counting b's that are to be matched against a's later
    - TG in next slide

TG for $n_a(w) = n_b(w)$

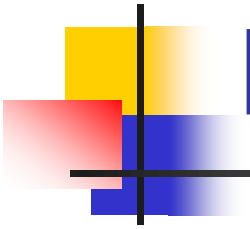
a, 0, 00; b, 1, 11
a, z, 0z; b, 0, λ;
b, z, 1z; a, 1, λ,





Exercises from Linz 7.1

- Construct npda's that accept the following languages on $\Sigma = \{a, b, c\}$
- Exercise #4
  - (c) $L = \{a^n b^m c^{n+m} : n \geq 0, m \geq 0\}$
  - Try it in class
    - Algorithm first
    - Implementation next



PDA and CFL

- Theorem 7.1
  - For every CFL, there is an npda M such that L = L(M)
  - Proof (outline)
    - CFL $\rightarrow$ CFG in Greibach normal form, G
    - G = (V, T, S, P)
    - M = ($\{q_0, q_1, q_f\}$, T, V$\cup\{z\}$, δ, q_0, z, $\{q_f\}$)
    - See detail steps in the constructive proof on page 186-187

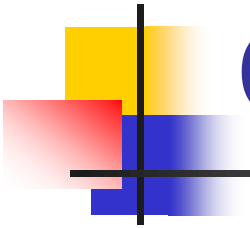
Example 7.6 - 1

Construct a pda that accepts the language generated by grammar $S \rightarrow aSbb \mid a$

- Transform to Greibach normal form
  - $S \rightarrow aSA \mid a$
  - $A \rightarrow bB$
  - $B \rightarrow b$
- The start symbol S is put on the stack by
  - $\delta(q_0, \lambda, z) = \{(q_1, Sz)\}$
- $S \rightarrow aSA \mid a$ are represented in the PDA by
  - $\delta(q_1, a, S) = \{(q_1, SA), (q_0, \lambda)\}$

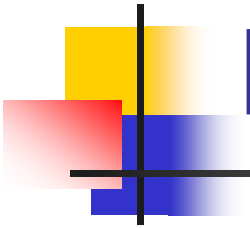
Example 7.6 - 2

- In an analogous manner,
  - A $\rightarrow$ bB gives
    - $\delta(q_1, b, A) = \{(q_1, B)\}$
  - B $\rightarrow$ b gives
    - $\delta(q_1, b, B) = \{(q_1, \lambda)\}$
  - Appearance of z signals the completion and pda is put into final state
    - $\delta(q_1, \lambda, z) = \{(q_2, \lambda)\}$



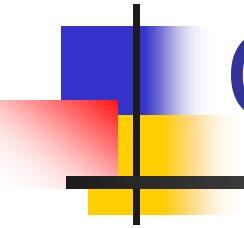
CFG for PDA

- Theorem 7.2
  - If L = L(M) for some npda M, the L is a CFL
  - Proof.
    - A constructive proof to show that it is always possible to construct a grammar from a npda.
  - This theorem tells us that there always a CFG for a language represented by a npda.
  - In practice, one may design a CFG directly from the give CFL



DPDA and DCFL

- A PDA M is said to be deterministic if it is an automaton as defined in Definition 7.1 for NPDA, subject to
  - For any given input symbol and stack top, at most one move can be made
  - When a λ-move is possible for some configuration, no input-consuming alternative is available
- DCFL L – if and only if there exists a dpda M such that L = L(M)



Chapter 9 Turing Machines

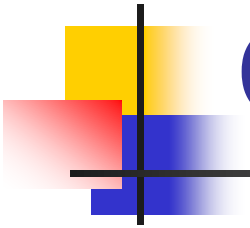
Definition of TM

TM as Accepters

TM as Transducers/Computer

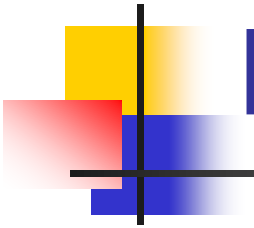
Turing thesis

Turing's Vision: the Birth of Computer Science



- Turing's theory forms the basis of computer science
- Combining abstract math theory and mechanical computation, Turing's theory provided elegant principle on how to use machine to mimic human brain to solve math problem

Computing Theory's Foundation

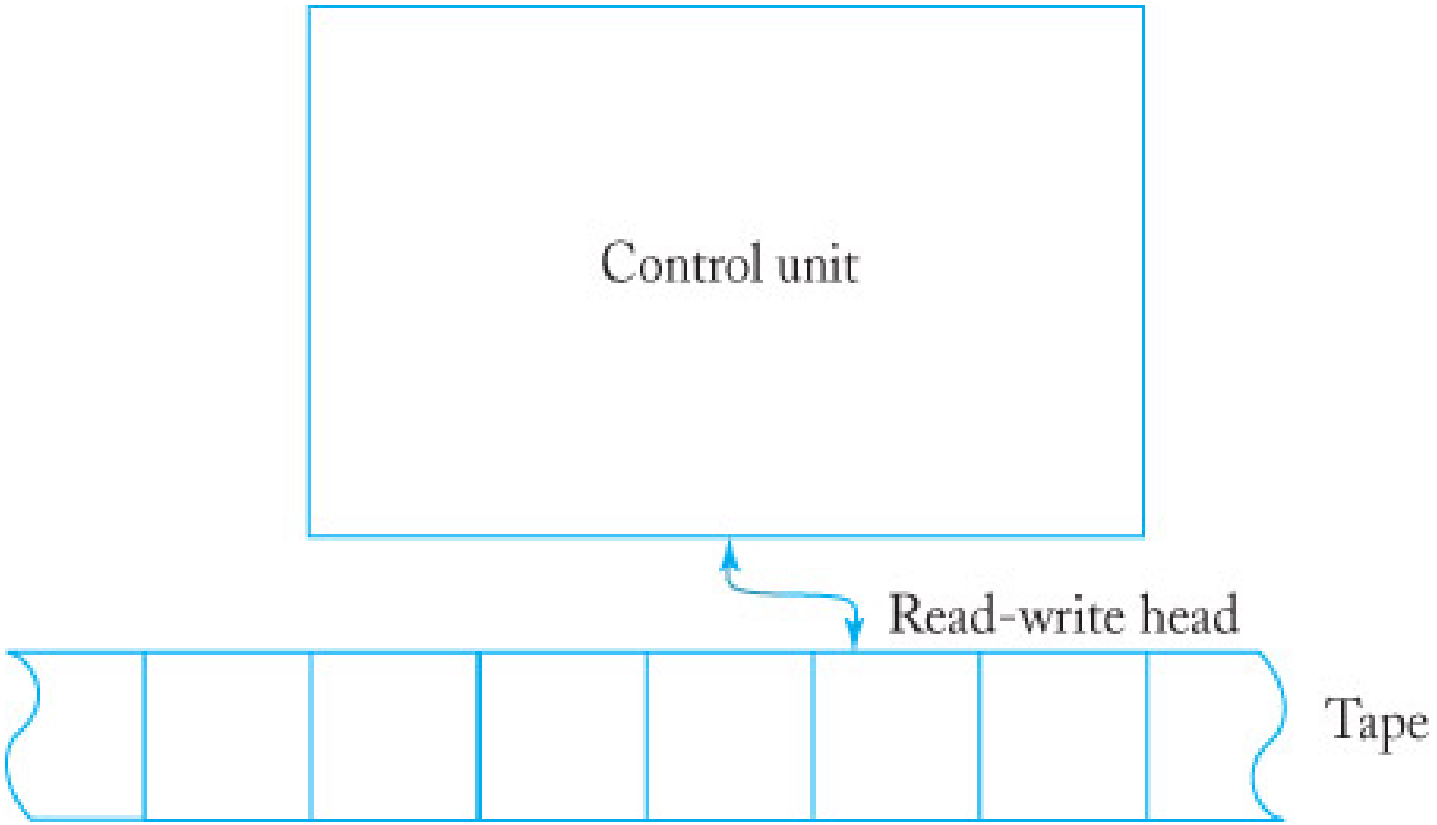


- In 1936, when he was just 24, Alan Turing wrote a remarkable paper in which he outlined the theory of computation, laying out the ideas that underlie all modern computers.
- 3 decision problems to explore the concept of undecidability:
  - To investigates theoretical computing machines, including **Turing machines**;
  - To explain universal machines; and
  - To prove that certain problems are undecidable

Fundamental ideas: automata

Automata	Storage	Accepter	Power level
FA	None	RL	Basic
PDA	Stack	CFL	More powerful
TM	Input/output Tape	Recursively Enumerable Language	Most powerful and flexible

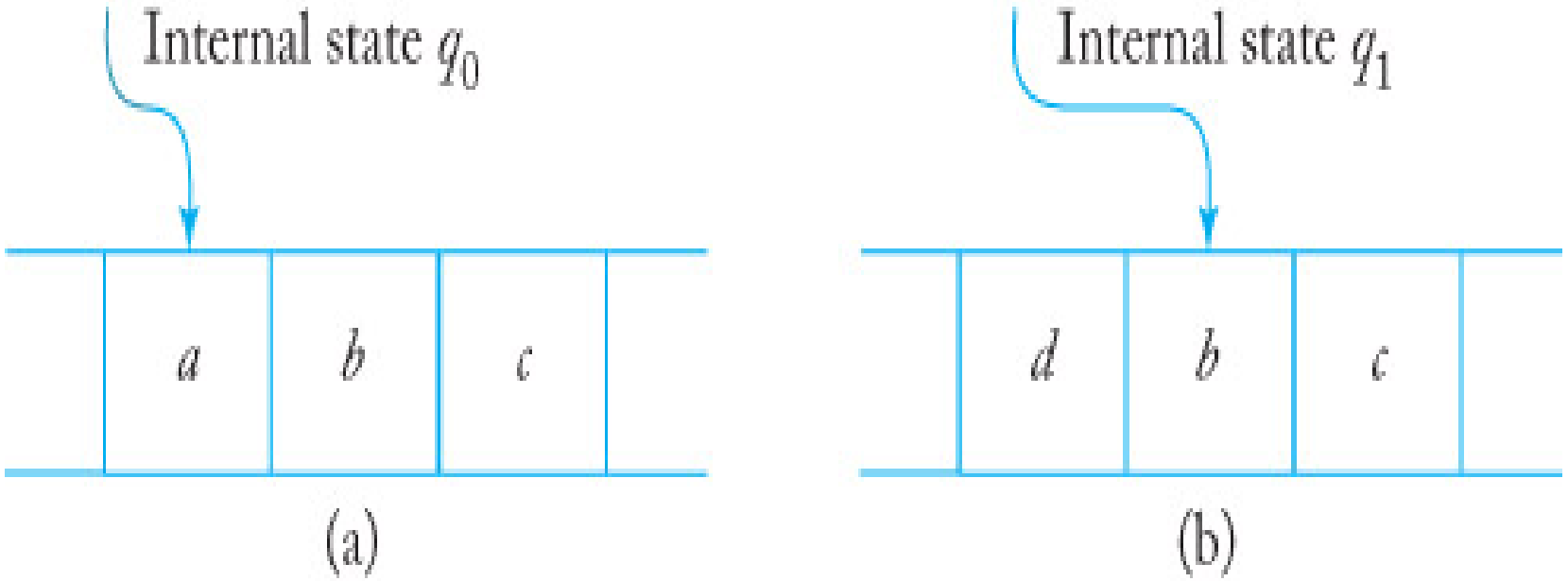
Visualization of a Turing machine



Definition 9.1: TM

- $M = (Q, \Sigma, \delta, \Gamma, q_0, \Box, F)$
- Q, Σ, q_0, and F are the same as they are in FA and PDA
- Γ is a finite set of symbols called the tape alphabet
- $\Box \in \Gamma$ is a special symbol called blank
- δ: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$

Fig 9.2 Before and after the move $\delta (q_0, a) = (q_1, d, R)$

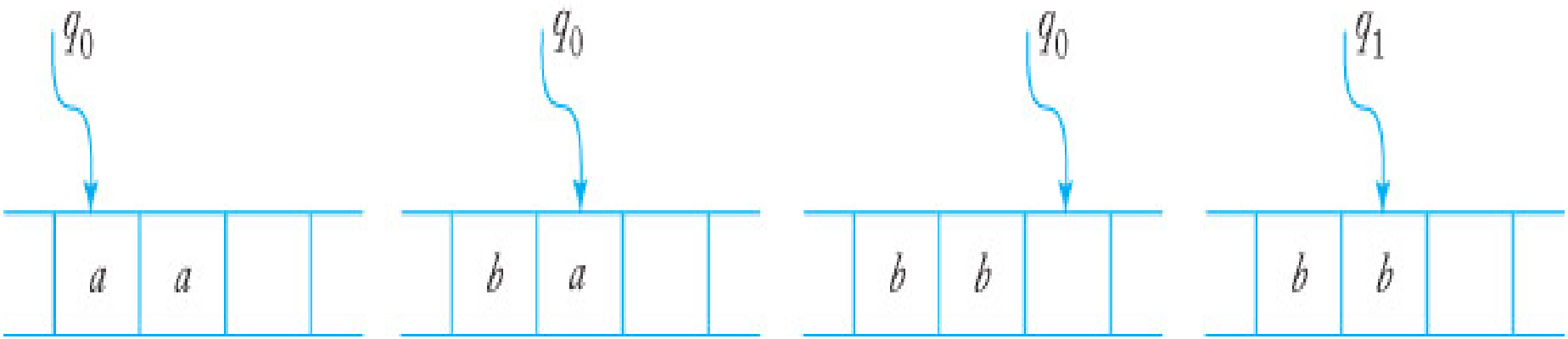


Example 9.2 - 1

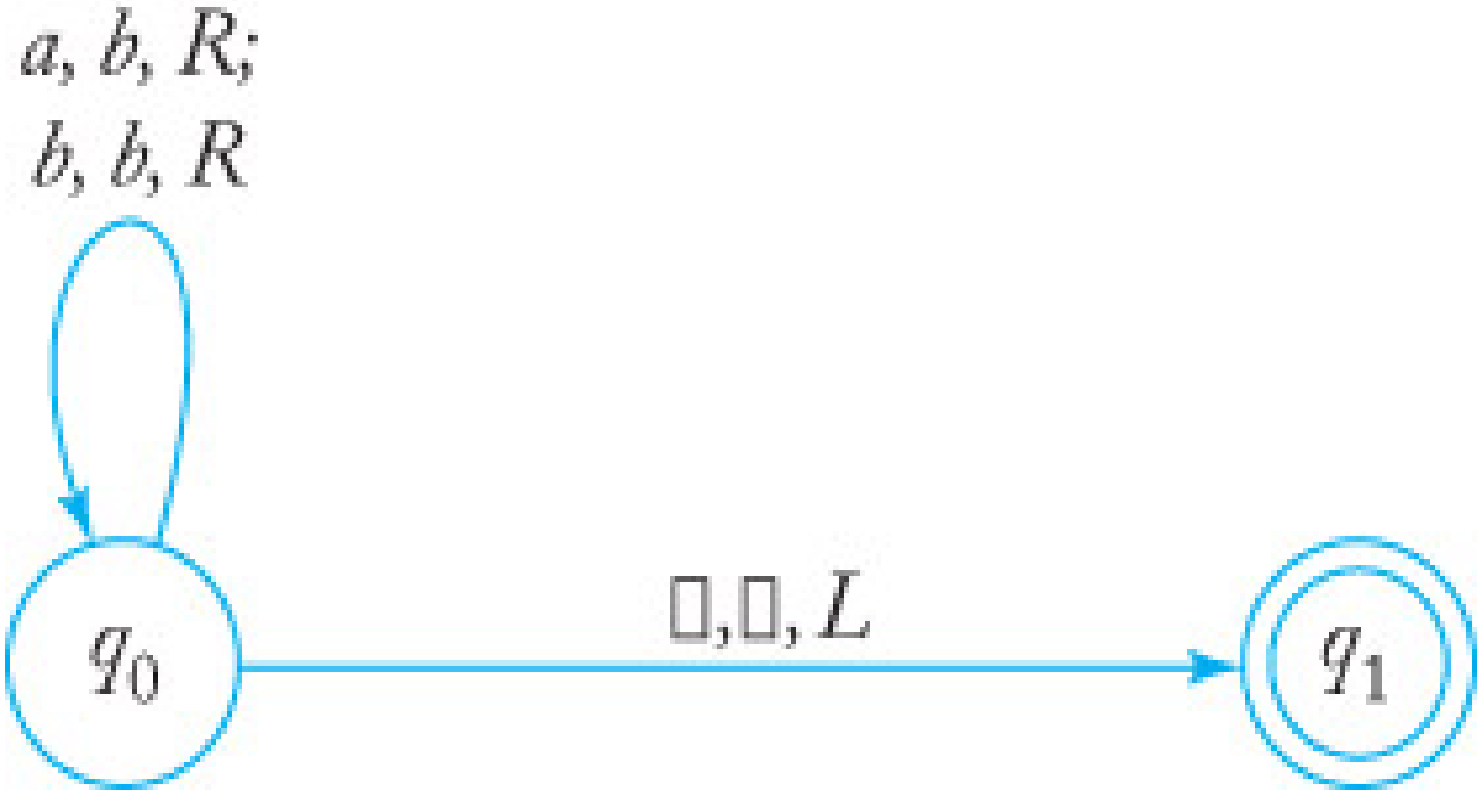
- $Q = \{q_0, q_1\}$,
- $\Sigma = \{a, b\}$,
- $\Gamma = \{a, b, \square\}$,
- $F = \{q_1\}$

Example 9.2 -2

$\delta(q_0, a) = (q_0, b, R)$, $\delta(q_0, b) = (q_0, b, R)$,
$\delta(q_0, \Box) = (q_1, \Box, L)$

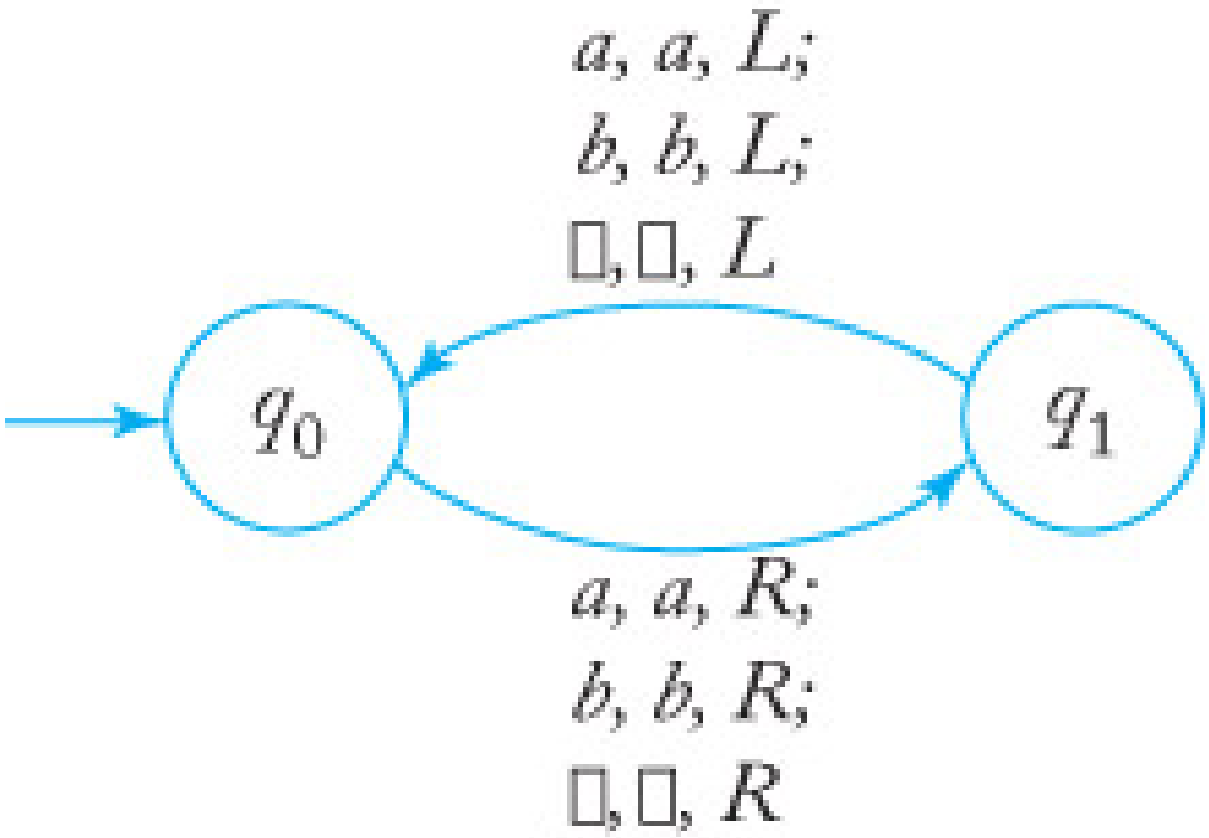


Example 9.2 -3 (Fig. 9.4)
Transition Graph Representation



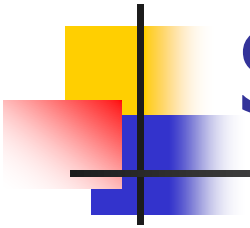
Example 9.3 (Fig. 9.5)

TM that does not halt = TM is in an infinite loop



Every TM T over the alphabet Σ divides the set of strings Σ^* into three classes

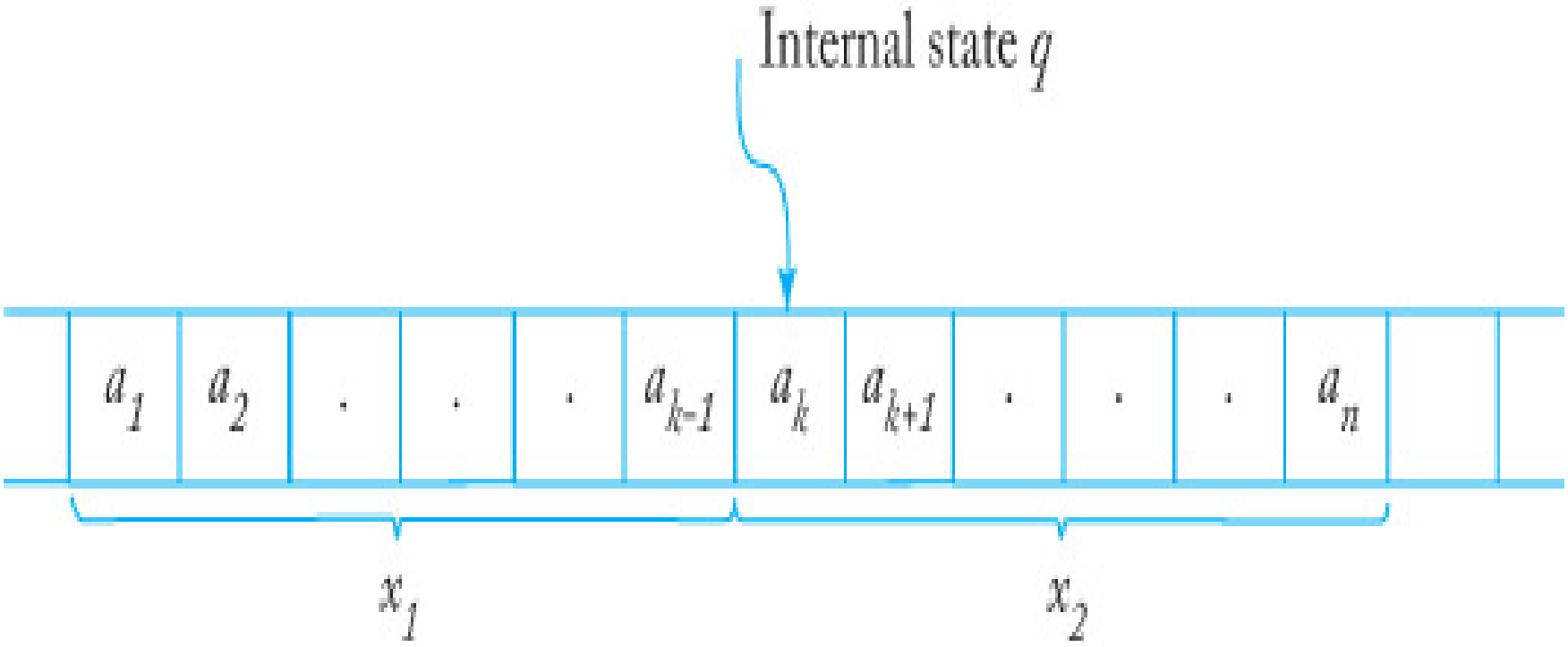
1. ACCEPT (T) is the set of all strings leading to a final state. This is also called the language accepted by T.
2. REJECT (T) is the set of all strings that halt in a non-final state.
3. LOOP (T) is the set of all other strings that loop forever while running on T.



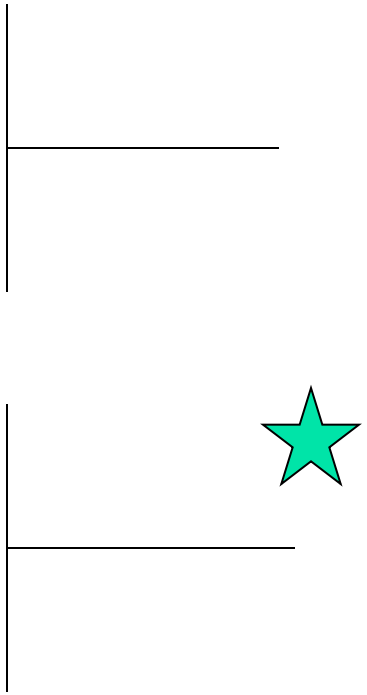
Standard Turing Machine

1. The TM has a tape that is unbounded in both directions allowing any number of left and right moves
2. The TM is deterministic in the sense that δ defines at most one move for each configuration
3. A tape for reading and writing (input and output)

Notation for a configuration: $x_1 q x_2$



A move from one configuration to another or an arbitrary number of moves

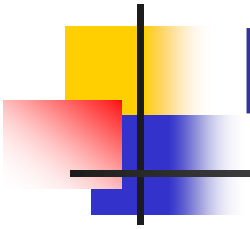


Definition 9.2 (page 228) Let M be a TM.

- Then any string $a_1 \ldots a_{k-1} q_1 a_k a_{k+1} \ldots a_n$ is an instantaneous description of M.
- A move
- $a_1 \ldots a_{k-1} q_1 a_k a_{k+1} \ldots a_n \vdash a_1 \ldots a_{k-1}\, b q_2 a_{k+1} \ldots a_n$
  - is possible iff $\delta(q_1, a_k) = (q_2, b, R)$
- M is said to halt if for any q_j and a for which $\delta(q_j, a)$ is undefined
- the sequence of configurations leading to a halt state will be called computation .

TM as Language Accepters

- Let M be a TM. Then the language accepted by M is
  - $L(M) = \{w \in \Sigma^+ : q_0\, w \vdash^* x_1 q_f x_2$ for some $q_f \in F$, $x1, x2 \in \Gamma^*\}$



FA $\Rightarrow$ TM

- Every RL has a TM that accepts it
  - Proof (using TG representations in both)
    - For each "a" on $\rightarrow$ in FA, change it to (a, a, R) on $\rightarrow$ in TM;
    - For initial state in FA, let it be the initial state in TM;
    - For final state in FA, let it be a state before going to a halt state with ($\square$, $\square$, R) on $\rightarrow$

Example 9.6
Design a TM for a RL

- $\Sigma = \{0, 1\}$
- Design a TM for RL L = L(00*)
  - Give a TG representation for FA
  - Give a TG representation for TM
  - Note, after obtained the TG for TM we may reduce the number of states
    - See text Example 9.6, L = L(00*), for an equivalent TM design with only 2 states.

Example 9.7 (page 230 - 231)

Design a TM for a CFL L = $\{a^n b^n : n \geq 1\}$

- Algorithm
  - Starting at the leftmost a, we check it off by replacing it with x.
  - We then let the read-write head travel right to find the leftmost b, with in turn is checked off by replacing it with y.
  - We go left again to the leftmost a, replace it with x, and so on.
  - Traveling back and forth, we match each a with a corresponding b.

Example 9.8

Design a TM for non-CFL $L = \{a^n b^n c^n : n \geq 1\}$

- Algorithm
  - Very similar with the algorithm in Example 9.7 although that one is CFL and this is not
  - We match each a, b, c by replacing them in order by x, y, z respectively.
  - At the end, we check that all original symbols have been rewritten.

TM as Transducers/Computers

- TM can do more than as a language accepter
- We can view a TM transducer M as an implementation of a function f defined by
  - w'= f(w)
  - Provided that $q_0 w \vdash^*_M q_f w'$, $q_f \in F$

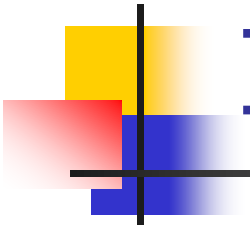
Definition 9.4 (page 232) Turing-computable

- Function f with domain D is said to be Turing-computable or just computable if there if there exists some TM M such that
  - $q_0 w \vdash^*_M q_f f(w), q_f \in F$
  - For all $w \in D$

Example 9.9

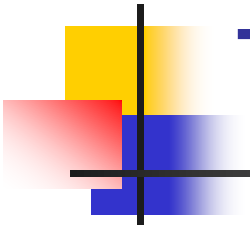
design a TM to add two positive integers: x and y

- Unary encoding
  - input x=3 and y=2 can be represented by
    - 111011, here 0 is a separator
  - Output x+y=5 can be represented by
    - 11111
  - See the sequence of instantaneous description for adding 111 to 11
    - $q_0111011 \vdash^* q_4111110$, q_4 is the final state



In-class Ex.

- Turing Machine as Computer:
  - (a) 3-interger adder;
  - (b) adder for any number of integers

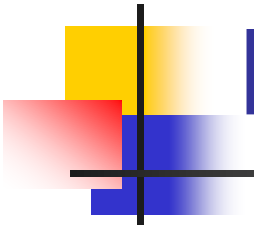


Turing's Thesis

1. Anything that can be done on any existing digital computer can also be done on a TM.
2. No one has yet been able to suggest a problem, solvable by what we intuitively consider an algorithm, for which a TM cannot be written.
3. Alternative models have been proposed for mechanical computation, but none of them is more powerful than the TM model.

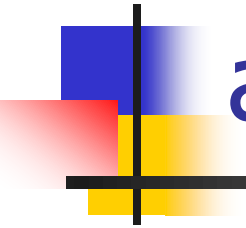
Definition 9.5
Algorithm

- An algorithm for a function f: D $\rightarrow$ R is a Turing machine M, which given as input any d $\in$ D on its tape, eventually halts with the correct answer f(d) $\in$ R on its tape.
  - $q_0 d \vdash^*_M q_f\, f(d)$, $q_f \in F$
  - For all d $\in$ D



References

- https://mitpress.mit.edu/books/turings-vision
- https://www.newscientist.com/article/mg23130803-200-how-alan-turing-found-machine-thinking-in-the-human-mind/
- https://en.wikipedia.org/wiki/Alan_Turing



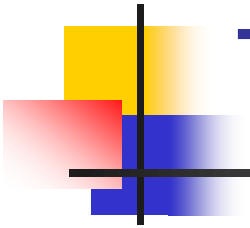
Chapter 11
A Hierarchy of Formal Languages and Automata

The Chomsky Hierarchy

Recursive and Recursively Enumerable Languages

Unrestricted Grammars

Context –Sensitive Grammars and Languages



The Chomsky Hierarchy

- Each of the four grammar classes has a simple yet powerful automaton
- A useful connection between grammars, automata, and languages
  - Type: 0, 1, 2, 3
  - Languages: r.e.L, CSL, CFL, RL
  - Grammar production restrictions: (X and Y)
    - $X \rightarrow Y$
  - Automata: TM, LBM, PDA, FA

To prove a language to be in certain language class

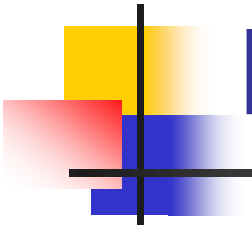
- You may show the language can be generated from associated grammar class
- For example,
  - G = 2 $\Rightarrow$ Automata =PDA
  - G = 0 $\Rightarrow$ Automata =TM

Chomsky Hierarchy of Grammars

The Chomsky Hierarchy of Grammars

Type	Name of Languages Generated	Production Restrictions $X \rightarrow Y$	Acceptor
0	Phrase-structure = recursively enumerable	X = any string with nonterminals Y = any string	TM
1	Context-sensitive	X = any string with nonterminals Y = any string as long as or longer than X	TM's with bounded (not infinite) TAPE, called linear-bounded automata LBA's‡
2	Context-free	X = one nonterminal Y = any string	PDA
3	Regular	X = one nonterminal $Y = t\,N$ or $Y = t$ t terminal N nonterminal	FA

‡The size of the tape is a linear function of the length of the input.

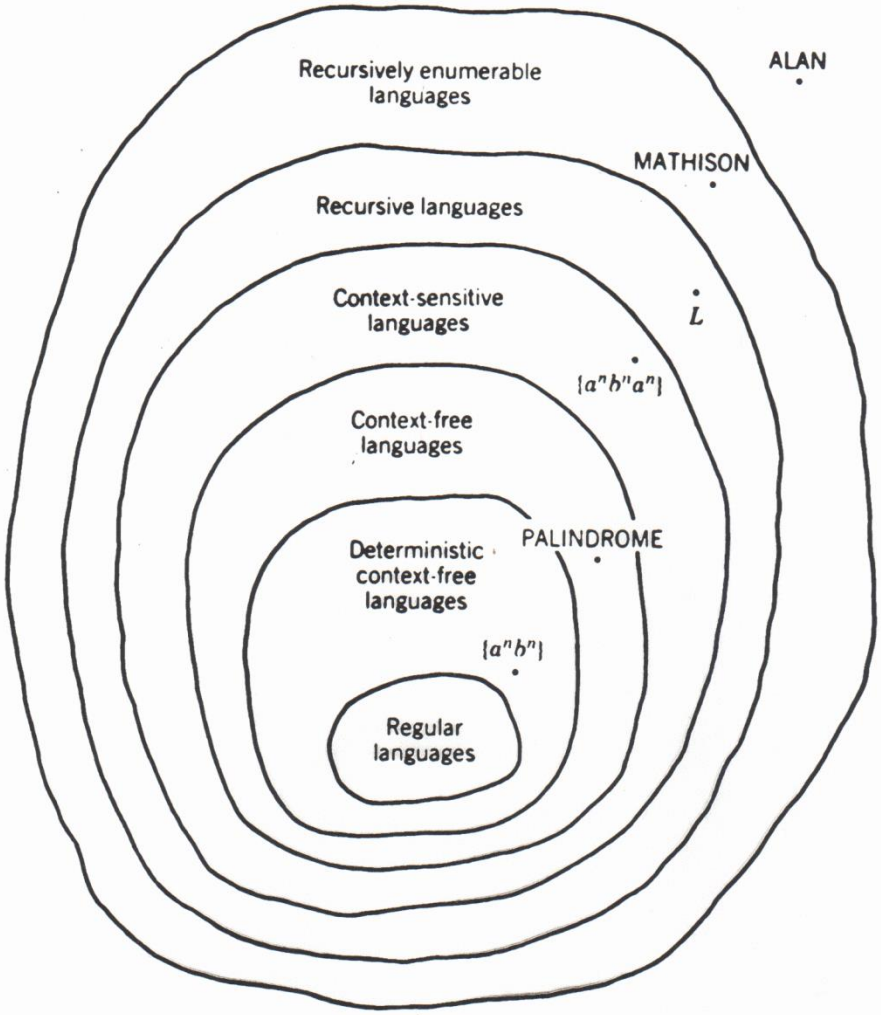


Note: exceptions

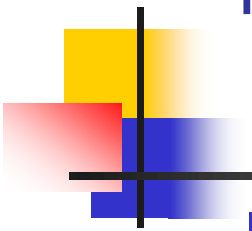
- Which type (0, 1, 2, 3) of grammar is the following grammar belong to?
  - S → aSb | λ

THE CHOMSKY HIERARCHY

These six classes of languages form a nested set as shown in the Venn diagram below.



No two of these categories are really the same



- $\{a^n b^n\}$ is deterministic context-free but not regular
- Palindrome is context free but not deterministic context-free
- $\{a^n b^n C^n\}$ is context-sensitive but not context-free

Chomsky Hierarchy of Grammars

(Make sure that you are able to fill it up!)

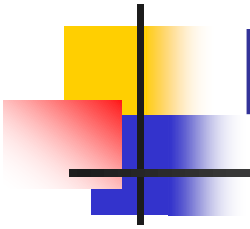
Type	Language	Production Restrictions	Acceptor	Representative

Every TM T over the alphabet Σ divides the set of strings Σ^* into three classes

1. ACCEPT (T) is the set of all strings leading to a final state. This is also called the language accepted by T.
2. REJECT (T) is the set of all strings that halt in a non-final state.
3. LOOP (T) is the set of all other strings that loop forever while running on T.

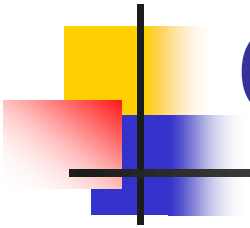
Definition 11.1

- A language L is said to be recursively enumerable if there exists a TM that accepts it



Definition 11.2

- A language L on Σ is said to be recursive if there exists a TM T that accepts L and that halts on every w in Σ^+
- In other words, a language is recursive if and only if there exists a membership algorithm
  - Note, in this case Loop(T) = ϕ



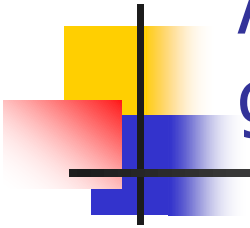
Question:

- Is recursive language a subset of recursively enumerable language or the other way around?

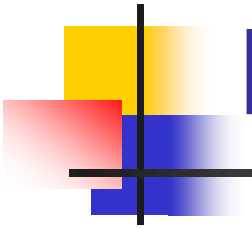
Definition 11.3
Unrestricted Grammar

- A grammar G = (V, T, S, P) is called unrestricted if all the productions are of the form
  - $u \rightarrow v$
  - Where u is in $(V \cup T)^+$ and v is in $(V \cup T)^*$
  - u should contain at least one non-terminal

Theorem 11.6
Any language generated by an unrestricted grammar is recursively enumerable

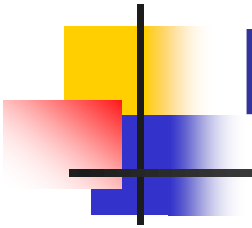


- Proof (highlight)
  - We can simulate the derivations of unrestricted grammar on a TM $\Rightarrow$ it is recursively enumerable



Definition 11.4

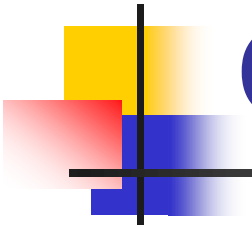
- A grammar G = (V, T, S, P) is said to be context-sensitive if all productions are of the form
  - $x \rightarrow y$
  - Where $x, y \in (V \cup T)^+$
  - and $|x| \leq |y|$



Definition 11.5

- A language L is said to be context-sensitive if there exists a context sensitive grammar G, such that L = L(G).

Context Sensitive Grammar Construction



- None of the examples is easy
- The next fifty years will see natural language processing as a dominant issue

Example 11.2

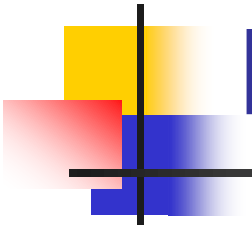
Show that L = $\{a^n b^n c^n : n \geq 1\}$ is CSL

- Construct a CSG for L
  - S $\rightarrow$ abc |aAbc
  - Ab $\rightarrow$ bA
  - Ac $\rightarrow$ Bbcc
  - bB $\rightarrow$ Bb
  - aB $\rightarrow$ aa |aaA
- Note: "A" is playing a role of messenger

Review exercises - 1

(need to be turned in together with HW)

- Page 296 #1 (b)
  - Find context-sensitive grammars for the following language:
  - $L = \{a^n b^n a^{2n} : n \geq 1\}$
- Turing Machine as Computer:
  - (a) 3-interger adder;
  - (b) adder for any number of integers;
  - (c) Construct a TM in TG that is an adder for any number of integers



Review Exercises -2

- Give a formal definition for the following terms: TM, CSG, Chomsky Hierarchy
- Construct a PDA in two steps (algorithm and TG) for the following languages:
  - $L=\{a^n b^m : n \leq m \leq 2n\}$
  - $L=\{a^n b^{3n} : n \geq 0\}$
  - $L=\{wcw^R : w \in \{a, b\}^*$

Review Exercises - 3

- Construct a TM in TG that will accept the following languages on {a, b},
  - L = {w: |w| is multiple of 2}
  - $L=\{a^n b^{3n} : n \geq 0\}$