Programming Languages
Lecture01 - Introduction

남 범 석

bnam@skku.edu

Instructor	Class code	Lecture hours	Office hours	Phone	Email
Nam, Beomseok	SWE3006	Tue 12:00 – 13:15 Thu 13:30 – 14:45	Tue, 8pm #85566	7967	bnam@skku.edu

- http://dicl.skku.edu/class/2019/pl

Programming Languages

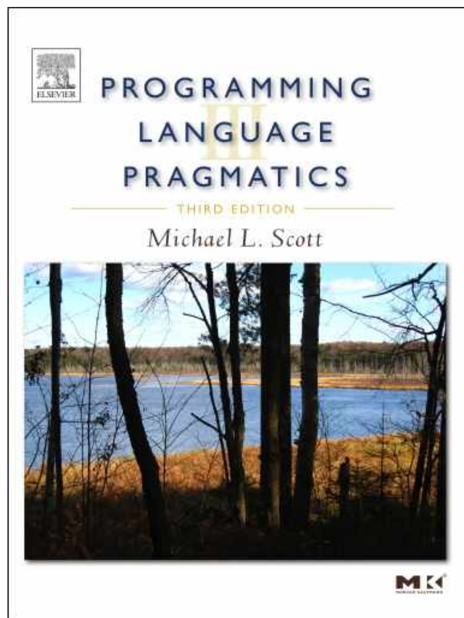
- Prerequisites
  - Programming in C
  - Programming in C++
  - Programming in Java
  - Programming in Python

  - Data Structures (SWE2015)
  - System Programming (SWE2001)
  - Automata (SWE2003)

Textbook

- Programming Language Pragmatics (3rd edition)
  - Author: Michael L. Scott
  - Publisher: Morgan Kaufmann

[3rd edition]



Grading

- Exams
  - Midterm 30%
  - Final 30%

- Assignments 30%
  - Programming assignments

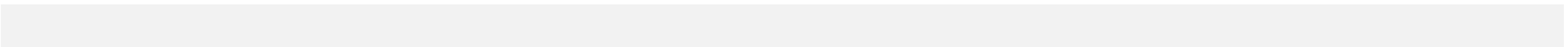
- Participation & quiz 10%
  - Attendance
  - Quiz

Contents

[Scott]

- Introduction – ch 1
- Compiler – ch 1, ch2.1
- Names, Scopes, Bindings – ch 3
- Controls – ch 6
- Types – ch 7
- Control Abstraction – Subroutines – ch 8
- Data Abstraction – Objects – ch 9
- Concurrency – ch 12
- Runtime Program Management – ch 15

Introduction to Formal Language for Those Who Haven’t Taken Automata



Formal Languages

- Examples: English, C, Pascal, arithmetic formulas
- Definitions
  - A **formal language** is a set of *strings* from an *alphabet*
  - An *alphabet* is a finite set of symbols, usually denoted by Σ
    - Ex.: letters, numbers, punctuation, etc.
  - A *string* is a finite sequence of concatenated symbols from an alphabet
    - Ex.: abcd, abab
  - The *length* of a string w is denoted by $|w|$. The empty string is denoted ε, and $|\varepsilon| = 0$.
  - The *concatenation* of two strings w_1 and w_2 is $w_1 w_2$.
    - Ex.: if w_1 = dog and w_2 = house, then $w_1 w_2$ = doghouse

Formal Languages (cont.)

- ϕ (the empty set) and $\{\varepsilon\}$ are different languages
  - $\{\varepsilon\}$ has a member, but ϕ doesn't
- Ex. 1: set of palindromes over alphabet $\{0,1\}$
  - infinite language
  - elements include ε, 0, 1, 00, 11, 010, 1101011, …
- Ex. 2: set of all strings over Σ, denoted Σ^*
  - if $\Sigma = \{a\}$, then $\Sigma^* = \{\varepsilon, a, aa, aaa, ...\}$
  - if $\Sigma = \{0, 1\}$ then $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$

Building languages from other languages

- Alphabet Σ, with L, L_1, L_2 sets of strings from Σ

1. *Concatenation* of L_1 and L_2, denoted L_1L_2, is the set

   $L_1L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$

   so get all possible combinations

   If $x = a_1a_2 \ldots a_n$ and $y = b_1b_2 \ldots b_m$,

   then the concatenation of x and y, denoted by xy is

   $a_1a_2 \ldots a_nb_1b_2 \ldots b_m$

Building languages (cont.)

2. *Union* of L_1 and L_2

$L_1 \mid L_2 = \{x \mid x \in L_1 \text{ or } x \in L_2\}$

no duplicates allowed

3. (*Kleene*) *closure* of L

define $L^0=\{\varepsilon\}$, $L^i=LL^{i-1}$ for $i \geq 1$

e.g. $L^2=LL$, $L^3=LLL$, …

then

Building languages (cont.)

$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup \ldots$$

is the **star closure** of *L*

$$L^+ = \bigcup_{i=1}^{\infty} L^i = L^1 \cup L^2 \cup L^3 \cup \ldots$$

is the **positive closure** of *L*

Building languages (cont.)

- So L^* denotes strings formed by concatenating any number of strings from L

- L^+ is the same, without concatenating 0 strings (to get ε), so ε is in L^+ if and only if ε is in L

- Examples:
  - $L_1 = \{10, 1\}$ and $L_2 = \{011, 11\}$
  - $L_1L_2 = \{10011, 1011, 111\}$
  - $L1 \mid L2 = \{10, 1, 011, 11\}$
  - $L_1^* = \{\varepsilon, 10, 1, 101, 110, 11, 1010, \ldots\}$
  - $L_1^+ = \{10, 1, 101, 110, 11, 1010, \ldots\}$

Languages

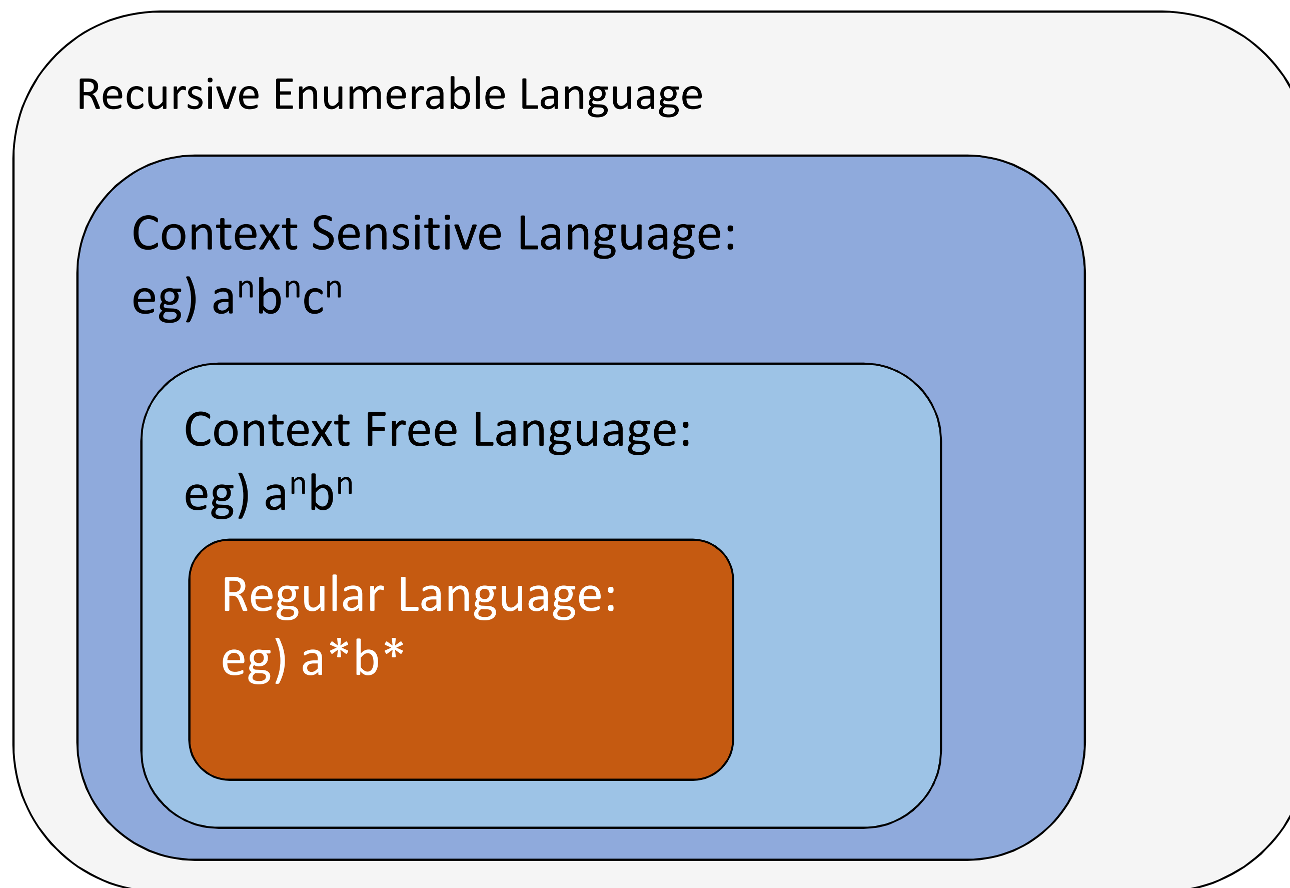
- What do we want to do with a language L over alphabet Σ?
  1. Determine, given a string $x \in \Sigma^*$, is $x \in L$? Preferably, an algorithm or constructive technique, called a *recognizer*
  2. Generate all strings in L , from a compact description - called a *generator*

Eg) *Automata, Turing machine, etc*

Turing machine is a theoretical device that gave a birth to modern computers (1936)

Formal Language Families - Chomsky Hiearchy

- *Noam Chomsky* (a founder of formal language) classified language types as follows:



Regular Languages

- For a class of languages called *regular languages*.
  - ***Regular expression (r.e.)*** is both recognizer and generator.
- r.e.'s are like pattern matching - given a string *x* from Σ^* and an r.e. *r*, a recognizer basically says "does *r* match *x*?"

Regular Expressions

- For an alphabet Σ and the r.e.'s over Σ, the regular languages (or regular sets) are:

1. ϕ is an r.e. for the empty set $\{\}$
2. ε is an r.e. for $\{\varepsilon\}$
3. For each $a \in \Sigma$, a is an r.e. for $\{a\}$

Regular Expressions (cont.)

- Let r and s be r.e.'s for sets R and S, respectively. Then
  4. $(r|s)$ is the r.e. for $R \cup S$
  5. (rs) is the r.e. for RS
  6. (r^*) is the r.e. for R^*

- Notation: allowed to drop parentheses around r.e.'s if it's clear what the r.e. means without them

Regular Expressions - Examples

- First, $L \Rightarrow$ r.e.
  1. L is the set of strings over S = {*a, b*} containing at least one *a* (*i.e.* a, aa, baaba, …)

$$(a \mid b)^* a (a \mid b)^* \quad or \quad b^* a (a \mid b)^*$$

  2. *L is set of all social security numbers (including the separator -)*

$$(0|1|2| \ldots |9)^6 - (0|1|2| \ldots |9)^7$$

Examples (cont.)

3. *L* is set of all signed and unsigned integer constants
   eg) +0, -1, 9, +86, -138, …

$$(+ \mid - \mid \varepsilon)(0|1| \ldots |9)^{+}$$

4. *L* is set of all strings in which every pair of adjacent 0's appears before any pair of adjacent 1's
   1010, 01, 010, 101000, 00001111, 1111,
   // 01100 is not a member of *L*

$$(10 \mid 0)^{*}(10 \mid 1)^{*}$$

Examples (cont.)

- Second, r.e. $\Rightarrow L$

1. $(0 \mid 1)^{*}11$

   is set of all strings ending in 11

2. $(0 \mid 1)^{*}111(0 \mid 1)^{*}$

   is set of all strings containing 3 consecutive 1's

3. $(000 \mid 00 \mid 1)^{*}(111 \mid 11 \mid 0)^{*}$

   is set of all strings over {0, 1} where all single 1's precede all single 0's.

   i.e. 10, 1011, 10110, 100100010, …

   // 01, 101, 1010 are not members

Language Recognizers

- We've seen how to write a pattern (r.e.) that describes a (regular) language
- Pattern matching is a common problem in many text processing systems.
  - Q: Does an input string match the given pattern (r.e.)?
- There's a ***machine (Automata)*** to do it
  - that's what Ruby (Perl, etc) uses for regular expression.

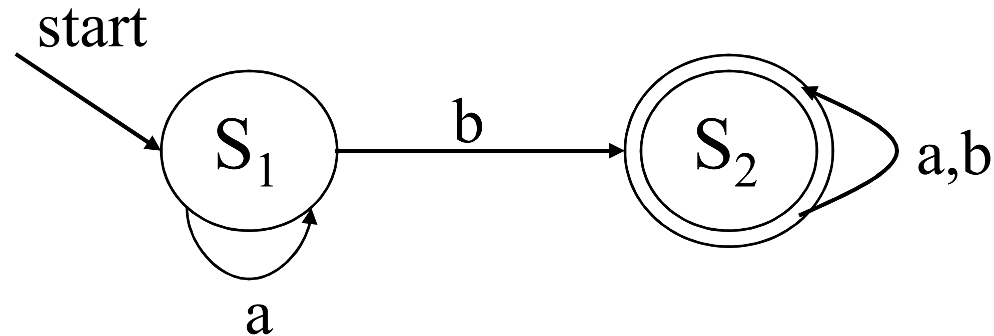
Deterministic Finite Automata (DFA)

- Let L be the set of strings over $\{a, b\}$ with at least one b. Then an r.e. for L is

$$a^*b(a \mid b)^*$$

- The DFA for L is a *labeled directed graph*, called a transition diagram

DFA (cont.)



- Terminology:
  - *states* s_1 and s_2
  - s_1 is *start/initial* state
  - s_2 is an *accepting/final* state (if end up in one of these, then the string is in *L*)
  - Graph edges are *transitions:*
    if current input symbol is *x*, then follow the edge labeled *x* to that state

DFAs (cont.)

- Formally, a DFA is a 5-tuple **$(Q, \Sigma, \delta, q_0, F)$**

  - Q is a ***set of states*** (in example, $Q = \{s_1, s_2\}$)
  - Σ is a ***finite input alphabet*** ($\Sigma = \{a, b\}$)
  - $q_0 \in Q$ is the ***initial state*** ($q_0 = s_1$)
  - $F \subseteq Q$ is the set of ***final states***
    - there can be more than one ($F = \{s_2\}$)
  - δ is the ***transition function***, mapping $Q \times \Sigma \rightarrow Q$, meaning $\delta(q, a) \in Q$ for each $q \in Q$ and $a \in \Sigma$

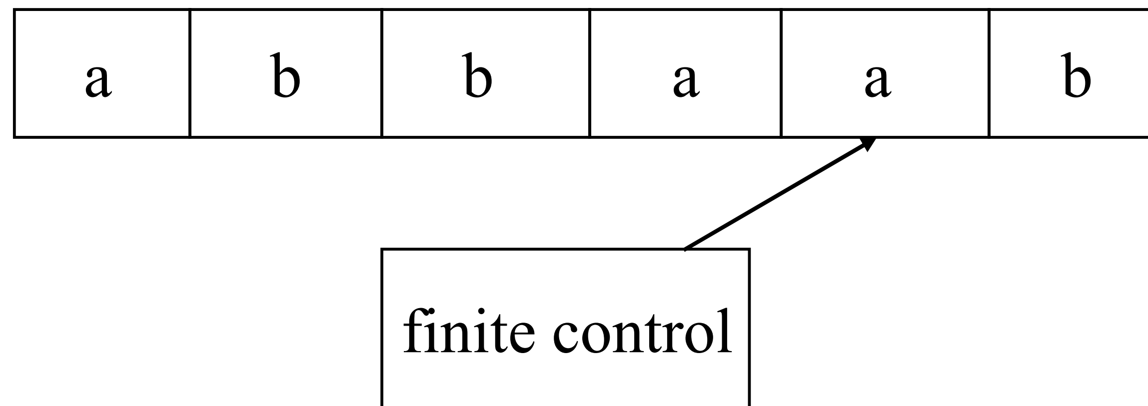
DFAs (cont.)

- For the example:

δ	**a**	**b**
$\mathbf{s_1}$	s_1	s_2
$\mathbf{s_2}$	s_2	s_2

DFAs (cont.)

- DFA can also be thought of as a *finite control*, in some state Q, reading symbols in Σ from an input tape or string
  - The control moves forward changing its current state.



  - A string x is *accepted* by a DFA $\boldsymbol{M}$ if $\delta(q_0,x) = p$, for some $p \in F$

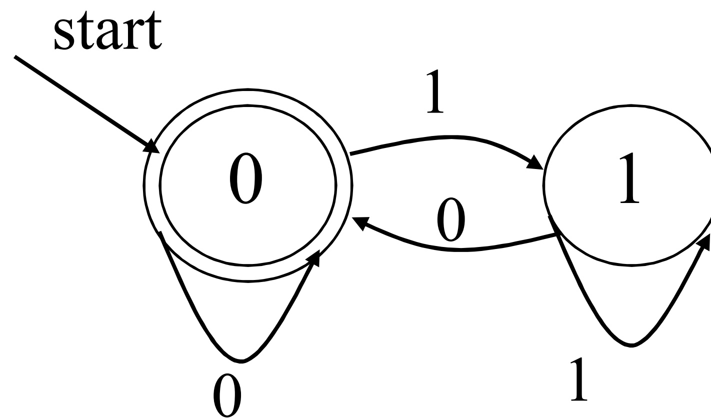
DFAs (cont.)

- The language L accepted by $\boldsymbol{M}$, $L(\boldsymbol{M})$, is the set $\{x \mid \delta(q_0,x) \in F\}$

- Regular expression vs DFA:
  - R.e and DFA have the same expression power

DFAs (cont.)

- Example: Build a DFA that accepts the language L, where L is the set of strings x over $\{0, 1\}$ such that if x is interpreted as a binary number, then x mod 2 = 0
  - key observation: input string is read from left to right, so starts from most significant digit of the binary number
  - Need 2 states, where being in each of the states means that the string seen so far, call it y, has the property y mod 2 = 0, or 1 - and label the states with those values
  - So, if already in one of the states, reading the next digit puts the DFA into the next state - multiply the state label by 2, and add the digit
    - that's the new value of the binary number!



What happens on input string 1000?
What happens on input string 1001?

Exercise: Draw a DFA that accepts L={x: x mod 3 = 0 }

Excercises

- Find all strings in $(a \mid b)^* b(a \mid ab)^*$ of length less than four.

- Give a regular expression for the following regular languages.
  - $L = \{a^n b^m : n \geq 1, m \geq 1, nm \geq 3\}$
  - $L = \{a^n b^m : (n+m) \text{ is even}\}$
  - $L = \{a^{2n} b^{2m+1} : n \geq 0, m \geq 0\}$

- Draw DFA's for the following regular languages.
  - $L = \{ ab^5 w b^4 : w \in \{a,b\}^* \}$
  - $L = \{ w : |w| \bmod 3 = 0 \}$
  - $L = \{ w : |w| \bmod 5 \neq 0 \}$

DFAs and R.E.'s

- Next question:
  - How to construct a DFA from an r.e. (an algorithm)
    - Can also construct an r.e. from a DFA, but it's harder
  - First, define a non-deterministic finite automata (*NFA*):
    - same as a DFA, except the transition function (δ) is not 1-1 (so there can be 2 or more transitions out of a state on the same input symbol)

NFAs

- An NFA *accepts* a string x if there is **at least one accepting path** (from the start state q_0 to some final state $f \in F$ labeled x)
  - There can be multiple paths for a given string x.

- NFAs and DFAs accept the same languages (the regular languages)
  - We can easily prove this, but that's for a theory class

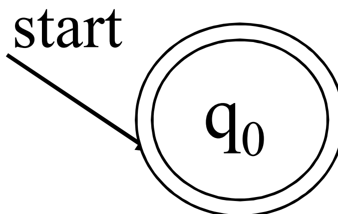
NFAs (cont.)

- Also, allow transitions on empty input, ε
  - if an edge in the transition diagram is labeled with ε, then the edge can be followed out of a state without consuming any input
- Again, NFAs with ε-transitions still only accept *regular* languages
- Since both DFAs and NFAs (either with or w/o ε-transitions) accept regular languages, we can construct a DFA w/o ε-transitions from an NFA.

NFAs from R.E.'s

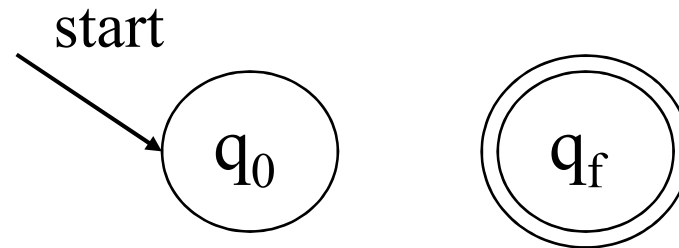
- Let's construct NFA from a r.e.
- We construct an NFA with ε-transitions from a r.e., by induction
- Base cases: for simple r.e.'s, call the r.e. r

  1. $r = \varepsilon$

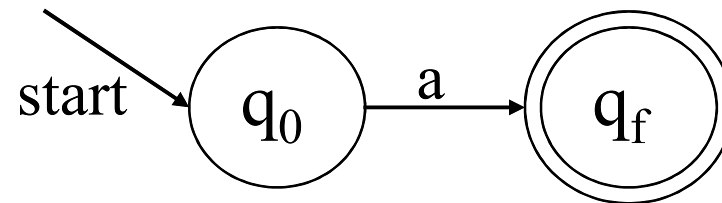


NFAs from R.E.'s (cont.)

2. $r = \phi$

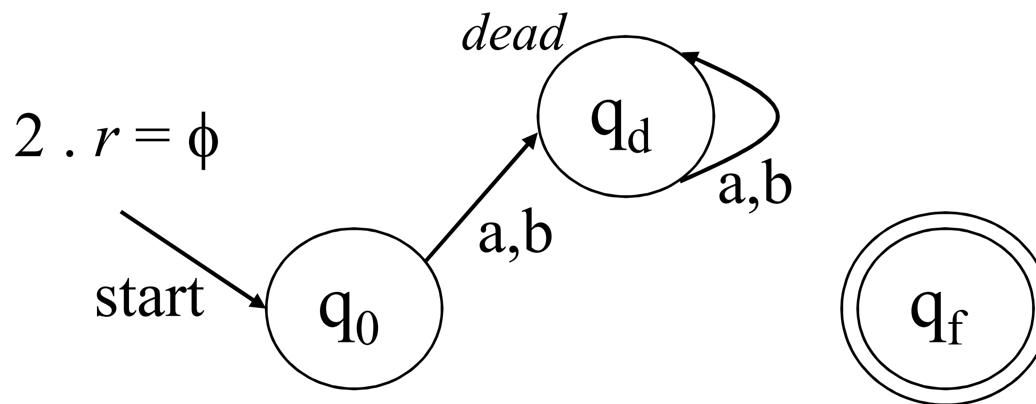


3. r = a, a $\in \Sigma$



NFAs from R.E.'s (cont.)

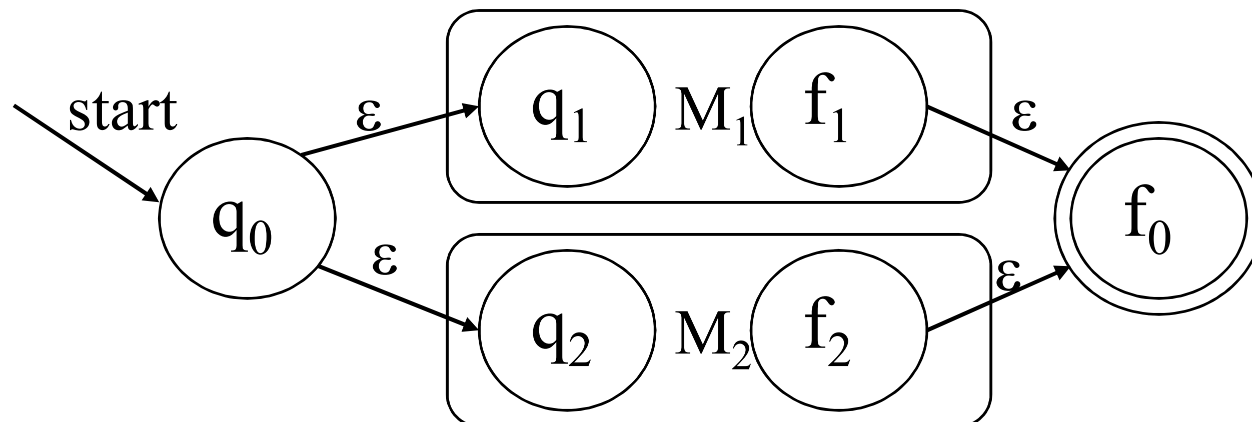
- Note: all the NFAs shown are incomplete, since they don't show transitions from all states on all input symbols in Σ - the missing ones are assumed to be to a *dead* state (one that has transitions on all input symbols back to itself, and it's not a final state)



NFAs from R.E.'s (cont.)

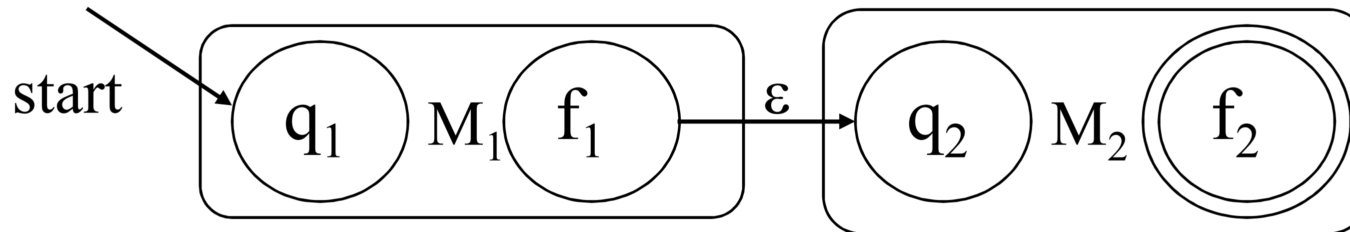
- For r.e.'s built from other r.e's by union, concatenation and closure (the inductive step), by construction with pictures:

  1. union - $(r_1 \mid r_2)$

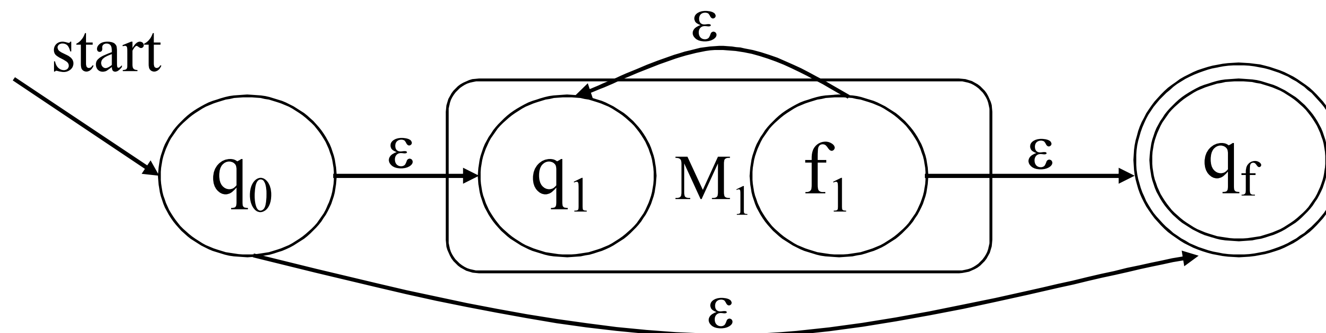


NFAs from R.E.'s (cont.)

2. Concatenation - $(r_1 r_2)$



3. Closure - $(r)^*$



Grammars

- A way to *generate* all strings in a language (and none that aren't)
- Example: Let L be the set of all strings over $\{a, b\}$ with at least one b.
  - the regular expression for L is $a^*b(a \mid b)^*$
- A ***BNF grammar*** for L is:

$$S \rightarrow aS \mid bB$$
$$B \rightarrow aB \mid bB \mid \varepsilon$$

Grammars (cont.)

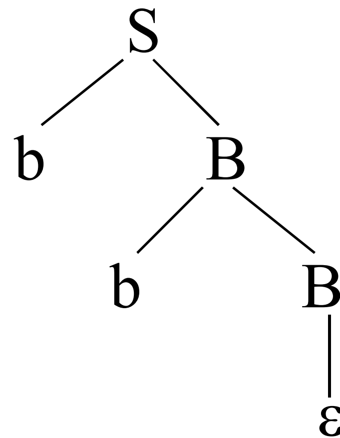
- A *derivation* of a string in *L*:

  $$S \Rightarrow bB \Rightarrow bbB \Rightarrow bb$$

- In a *derivation tree* for a string generated by a grammar, the leaves are symbols in the alphabet (called *terminals*), and the interior nodes are *variables* (or *non-terminals*)

Derivation Trees

- A derivation tree for the string *bb*:



Grammar defined

- Formal definition of a grammar is a 4-tuple $G = (\Sigma, N, P, S)$:
  - Σ is a finite alphabet (with each symbol called a *terminal*)
  - N is a finite set of variables (*non-terminals*)
  - P is a finite set of *productions*
  - S is a variable called the *start symbol*
  - and $\Sigma \cap N = \phi$

Grammars (cont.)

- Productions have the form:

$$A \rightarrow \alpha$$

  - sometimes written A ::= α
  - where A is a non-terminal, $A \in N$, and α is a sequence of terminals and non-terminals, $\alpha \in (\Sigma \cup N)^*$

Regular Grammars

- For *regular* languages, productions have a certain form
  1. $A \rightarrow cB$
  2. $A \rightarrow c$
  3. $A \rightarrow \varepsilon$

  where *A, B* are non-terminals and *c* is a terminal
- If a grammar only has productions of these forms, then it generates a regular language

Regular Grammars (cont.)

- The grammar form shown is called a *right linear* grammar
  - *linear* because generates regular languages
  - *right* because the non-terminal is to the right of the terminal in the first form of production
- The language *generated* by a grammar is the set of all strings that can be derived from *S* and consist only of terminals

BNF grammar can generate more complicated languages

Context Free Grammars (CFGs)

- Most programming languages fall into this class
  - Context Free Languages

- *All* productions must be of the form

$$A \rightarrow \alpha$$

  with A a non-terminal and $\alpha \in (\Sigma \cup N)^*$
  (i.e. a sequence of terminals and non-terminals)

Note: If terminals appear on the left side of production rule, the grammar is Context Sensitive Grammar.
eg) $\alpha S \beta \rightarrow \alpha \gamma \beta$

CFGs (cont.)

- Example: $L = a^n b^n$, $n \geq 0$
- L is *not* regular, but is context free, with grammar

$$S \rightarrow aSb \mid \varepsilon$$

  - | is shorthand for separate productions

- To derive the string *aabb*

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

- Note:
  If a programming language is a context free language (usually), called a CFL, then there is a CFG that produces *all* syntactically correct program source codes.

Generation vs. Recognition

- A grammar *generates* all strings in a language
  - Regular grammar generates regular languages
  - CFG generates context free languages

- A machine *recognizes* all strings in a language
  - DFA/NFA recognizes regular languages
  - What about context-free-languages?

  - Note: r.e. is a generator and recognizer as well.

Recognizers

- The recognizer for a CFL is a ***pushdown automata*** (basically a DFA, plus a stack for counting and matching)
  - Recognizer is also called a ***parser***
    *D*erivation trees are also called ***parse trees***

  - Compilers need to recognize whether a program is a legal string in the programming language (i.e. is $x \in L$?)
    - Compiler is made up of several components. One of them is a parser.
    - Parser checks for the syntax of sentences being correct.
    - The rest of components will be discussed later.

Overview

Language	Generator	Recognizer
Regular languages	Regular expressions Right linear grammars	DFA (NFA too)
CFL	CFG	Pushdown automata
Programming language	CFG (usually in Backus-Naur form – BNF)	Parser

Overview

Language	Generator	Recognizer
Regular languages	Regular expressions Right linear grammars	DFA (NFA too)
CFL	CFG	Pushdown automata
Programming language	CFG (usually in Backus-Naur form – BNF)	Parser

Grammars - Examples

- $L = \{a^n b^m \text{ for } n,m \geq 0\}$

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aA \mid \varepsilon \\ B &\rightarrow bB \mid \varepsilon \end{aligned}$$

- $L = \{a^n b^m a^{n+m} \text{ for } n,m \geq 0\}$
  - observation: rewrite as $a^n(b^m a^m)a^n$
  - Grammar is:

$$\begin{aligned} S &\rightarrow aSa \mid B \\ B &\rightarrow bBa \mid \varepsilon \end{aligned}$$

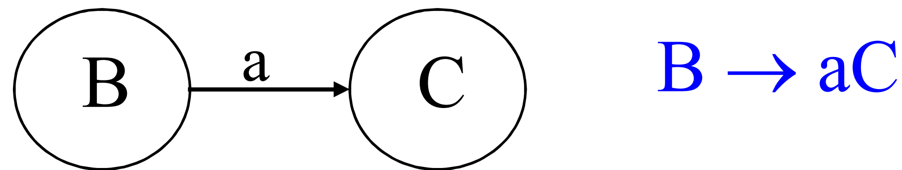
Examples (cont.)

- L = set of all strings over $\Sigma = \{(,)\}$ with balanced (matching) parentheses.
  - eg) ( ) ((( ) ( )))

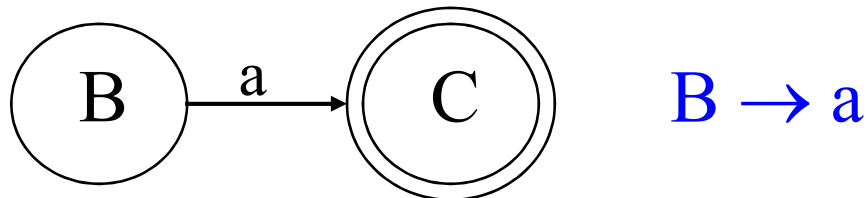
$$S \rightarrow (S) S \mid \varepsilon$$

Producing Grammars

- Algorithm to generate a (right linear) grammar from a DFA $M = (Q, \Sigma, \delta, q_0, F)$
- For all transitions in δ



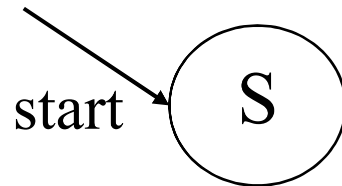
$B \rightarrow aC$



$B \rightarrow a$

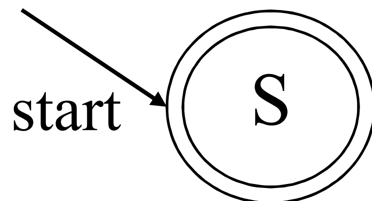
Grammars and DFAs

- For the start state



then S is the start symbol

- And, if $q_0 \in F$



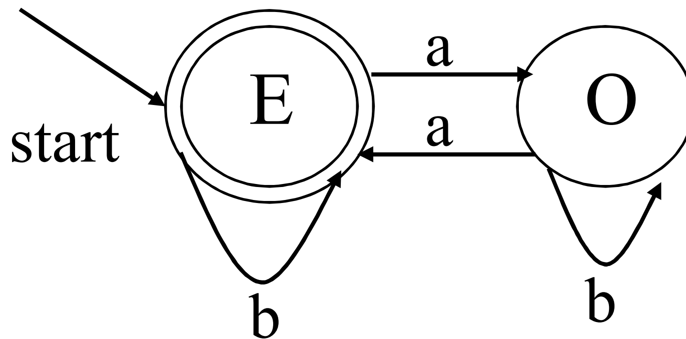
Add $S' \rightarrow S \mid \varepsilon$
and S' is new start symbol

Grammars and DFAs (cont.)

- So for $G = (\Sigma', N, P, S)$ and
  $M = (Q, \Sigma, \delta, q_0, F)$
  - $\Sigma' = \Sigma$ (terminals correspond to input symbols)
  - $N = Q$ (non-terminals correspond to states)
  - productions (P) correspond to transitions (δ)
  - $S = q_0$ (start symbol corresponds to start state)

Example

- L = set of strings over {a,b} with even number of a's (i.e. #a(x) mod 2 = 0)



$E \rightarrow bE \mid b$

$E \rightarrow aO$

$O \rightarrow bO$

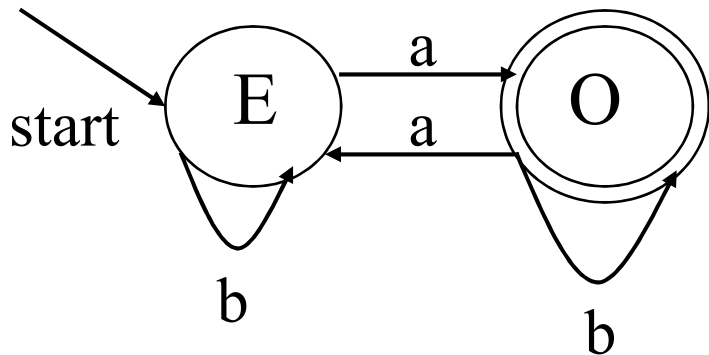
$O \rightarrow aE \mid a$

and new start symbol

$E' \rightarrow E \mid \varepsilon$

Example 2

- L = set of strings over {a,b} with odd number of a's (i.e. #a(x) mod 2 = 1)



$E \rightarrow bE$

$E \rightarrow aO \mid a$

$O \rightarrow aE$

$O \rightarrow bO \mid b$

and E is start symbol

Hints for Writing Grammars

I. Arbitrary length strings

- recursive rule/production, and non-recursive alternative

▪ Ex.: $k \geq 0$, $a^k = aa^{k-1}$, rule is

$$S \rightarrow aS$$

- $a^0 = \varepsilon$, rule is: $S \rightarrow \varepsilon$

- if $k \geq 1$, still have $a^k = aa^{k-1}$
  but alternative is $a^1 = a$, with rules

$$S \rightarrow aS \mid a$$

II. Matching superscripts

- generate string from the middle

▪ Ex. 1: $n \geq 0$, $a^n b^n = a(a^{n-1} b^{n-1})b$, so rule is

$$S \rightarrow aSb$$

- and $a^0 b^0 = \varepsilon$, so rule is

$$S \rightarrow \varepsilon$$

▪ Ex. 2: $n \geq 0$, $m = 2n$

$a^n b^m = a^n b^{2n} = a^n (bb)^n$, so rules are

$$S \rightarrow aSbb \mid \varepsilon$$

Matching Superscripts (cont.)

- Ex. 3: $n \geq 0$, $k \geq 2n$, language is $a^n b^k$
  - Combine I and II to get rules

$$S \rightarrow aSbb \mid B$$

$$B \rightarrow bB \mid \varepsilon$$

Concatenation Rule

III. Concatenation:

- Do each piece separately, and concatenate them

- Ex.: $a^n b^m$, $n,m \geq 0$.
- Rules are

$$S \rightarrow AB$$
$$A \rightarrow aA \mid \varepsilon$$
$$B \rightarrow bB \mid \varepsilon$$

Union Rule

IV. Union

- Use separate rules/productions

- Ex.: $a^n \cup b^m$, $m,n \geq 0$. Rules are

$$S \rightarrow A \mid B$$
$$A \rightarrow aA \mid \varepsilon$$
$$B \rightarrow bB \mid \varepsilon$$