Formal Languages

Strings

- **Alphabet**: a finite set of symbols
  - Normally characters of some character set
  - E.g., ASCII, Unicode
  - Σ is used to represent an alphabet
- **String**: a finite sequence of symbols from some alphabet
  - If s is a string, then $|s|$ is its length
  - The empty string is symbolized by ϵ

String Operations

Concatenation

- x = hi, y = bye $\longrightarrow$ xy = hibye
- sϵ = s = ϵs

$$s^i = \begin{cases} \epsilon\,, & \text{if } i = 0 \\ s^{i-1}s\,, & \text{if } i > 0 \end{cases}$$

Parts of a String

- Prefix
- Suffix
- Substring
- Proper prefix, suffix, or substring
- Subsequence

Language

- A language is a set of strings over some alphabet

  $$L \subseteq \Sigma^*$$

- Examples:
  - $\varnothing$ is a language
  - $\{\epsilon\}$ is a language
  - The set of all legal Java programs
  - The set of all correct English sentences

Operations on Languages

Of most concern for lexical analysis

- Union
- Concatenation
- Closure

Union

The union of languages L and M

$$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$$

Concatenation

The concatenation of languages L and M

$$LM = \{st \mid s \in L \text{ and } t \in M\}$$

Kleene Closure

The Kleene closure of language L

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

Zero or more concatenations

Positive Closure

The positive closure of language L

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

One or more concatenations

Example

- Let $L = \{A, B, C, \ldots, Z, a, b, c, \ldots, z\}$
- Let $D = \{0, 1, 2, \ldots, 9\}$

$L \cup D$	LD
L^4	L^*
$L(L \cup D)^*$	D^+

Regular Expressions

- A convenient way to represent languages that can be processed by lexical analyzers
- Notation is slightly different than the set notation presented for languages
- A regular expression is built from simpler regular expressions using a set of defining rules
- A regular expression represents strings that are members of some *regular set*

Rules for Defining Regular Expressions

- The regular expression r denotes the language $L(r)$
- ϵ is a regular expression that denotes $\{\epsilon\}$, the set containing the empty string
- If a is a symbol in the alphabet, then a is a regular expression that denotes $\{a\}$, the containing the string a
- How to distinguish among these notations

Combining Regular Expressions

- Let r and s be regular expressions that denote the languages $L(r)$ and $L(s)$ respectively

$(r)\|(s)$	is a regular expression denoting	$L(r) \cup L(s)$
$(r)(s)$	is a regular expression denoting	$L(r)L(s)$
$(r)^*$	is a regular expression denoting	$(L(r)^*)$
(r)	is a regular expression denoting	$L(r)$

- The language denoted by a regular expression is called a regular set

More Formally

$a \in \Sigma$

E and F are regular expressions

$$
\begin{array}{lcl}
L(\varnothing) & = & \varnothing \\
L(\boldsymbol{\epsilon}) & = & \{\boldsymbol{\epsilon}\} \\
L(a) & = & \{a\} \\
L(EF) & = & \{ab \mid a \in L(E) \text{and} b \in L(F)\} \\
L(E \mid F) & = & L(E) \cup L(F) \\
L((E)) & = & L(E) \\
L(E^*) & = & L(E)^*
\end{array}
$$

Precedence Rules

- Precedence rules help simplify regular expressions
  - Kleene closure has highest precedence
  - Concatenation has next highest
  - $|$ has lowest precedence
- All operators associate left-to-right

Example

- Let $\Sigma = \{a, b\}$
- Find the strings in the language represented by the following regular expressions:

$$a \mid b \qquad (a \mid b)(a \mid b)$$

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

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

Algebra of Regular Expressions

Property	Definition
$\mid$ is commutative	$r \mid s = s \mid r$
$\mid$ is associative	$(r \mid s) \mid t = r \mid (s \mid t)$
Concatenation is associative	$(rs)t = r(st)$
Concatenation distributes over $\mid$	$r(s \mid t) = rs \mid rt$ $(s \mid t)r = sr \mid tr$
ϵ is the identity element for concatenation	$\epsilon r = r = r\epsilon$
Relation between $*$ and ϵ	$(r \mid \epsilon)^* = r^*$
$*$ is idempotent	$r^{**} = r^*$

Mathematically Describing Relational Operators

$$\Sigma = \{ <, >, =, ! \}$$

$$relop = < \mid > \mid <= \mid >= \mid == \mid != $$

Identifiers and Numbers

Σ = { a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r,
s, t, u, v, w, x, y, z, A, B, C, D, E, F, G, H, I, J,
K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, 0, 1,
2, 3, 4, 5, 6, 7, 8, 9, _ }

letter = a| b| c| d| e| f| g| h| i| j| k| l| m| n| o| p| q| r|
s| t| u| v| w| x| y| z| A| B| C| D| E| F| G| H| I| J|
K| L| M| N| O| P| Q| R| S| T| U| V| W| X| Y| Z|_

digit = 0| 1| 2| 3| 4| 5| 6| 7| 8| 9

identifier = $\mathit{letter}\ (\ \mathit{letter} \mid \mathit{digit})^*$

number = $\mathit{digit}\ \mathit{digit}^*$

Finite Automata

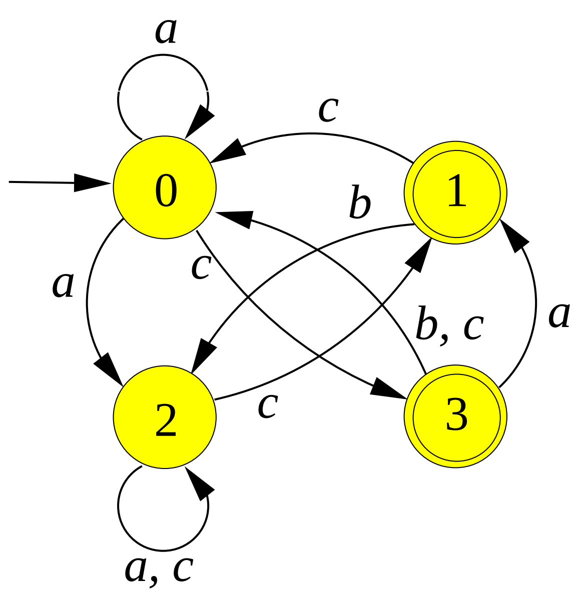
A non-deterministic finite automaton (NFA) is a 5-tuple:

$$\langle S, \Sigma, \phi, s_0, F \rangle$$

- S a set of states
- Σ a set of input symbols
- ϕ a transition function $(S, \Sigma) \longrightarrow S$
- s_0 a distinguished state called the *start state*
- F a set of accepting or final states

NFA Representation

An NFA can be conveniently represented by both a directed graph and a table

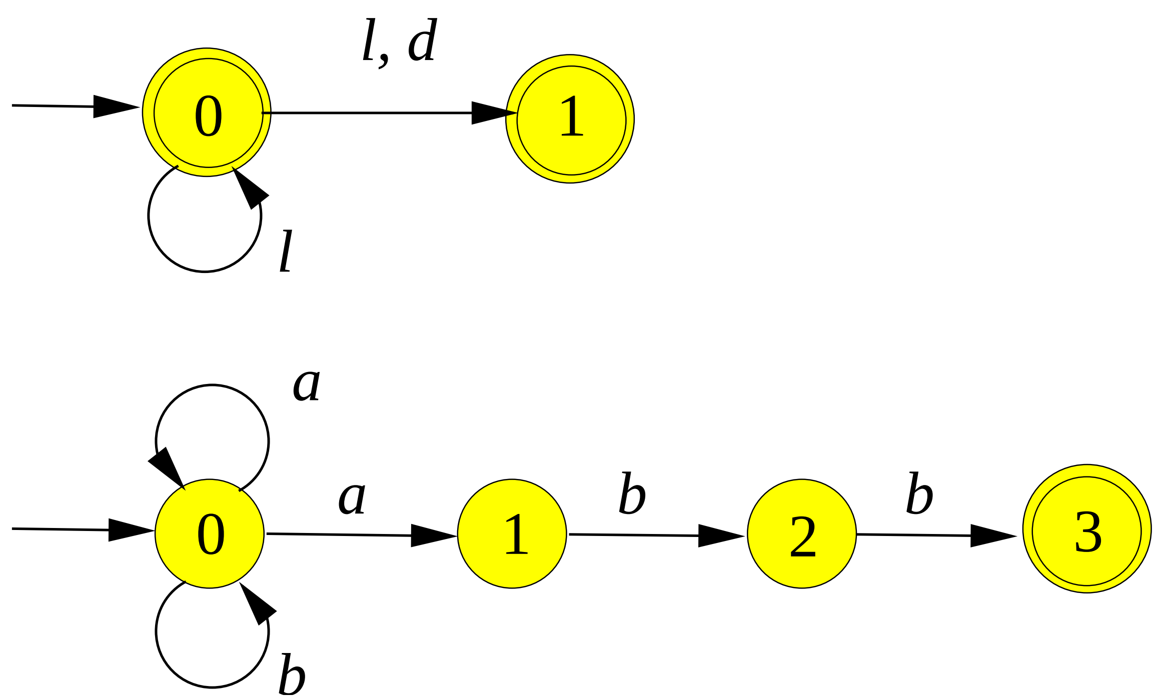


Current	Next State			
State	*a*	*b*	*c*	Output
0	{ 0, 2 }	–	3	0
1	–	2	0	1
2	2	–	{1, 2}	0
3	1	0	0	1

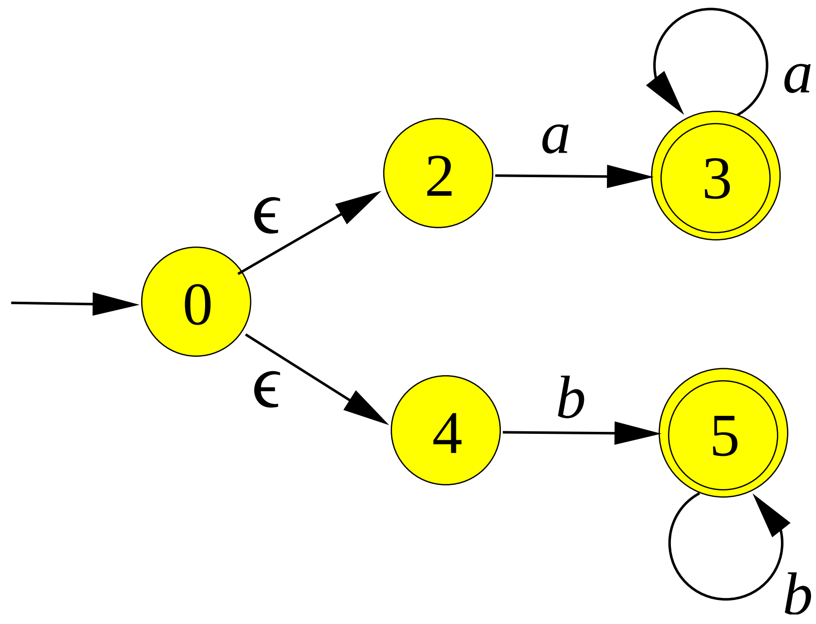
Final states

- are double circled (graph)
- output a 1 (table)

NFA Transition Graphs



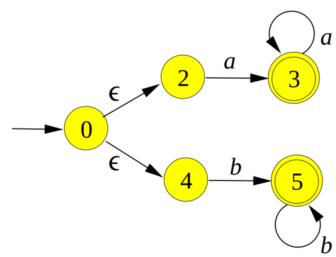
Another NFA



NFAs and Regular Sets

- An NFA can be built to recognize strings represented by a regular expression

    (i.e., strings that are members of some regular set)



NFAs as Recognizers

- Given an NFA M, $L(M)$ is the language recoginized by that machine

- If the NFA scans the complete string and ends in a final state, then the string is a member of $L(M)$

  We say M accepts the the string

- If the NFA scans the complete string and ends in a non-final state, then the string is not a member of $L(M)$

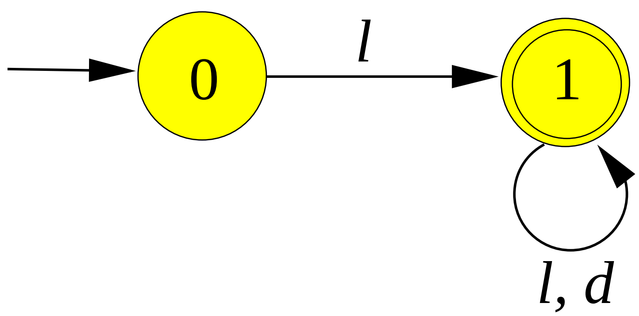
  We say M rejects the the string

- Because of non-determinism a string is accepted if there is a path to a final state; a string is rejected if there is no path to a final state

  Think about the NFA following all non-deterministic paths in parallel

Deteministic Finite Automata (DFA)

- A special case of an NFA
- Also called a *finite state machine*
- No state has an ϵ-transition
- $\forall s \in S$ and $\forall a \in \Sigma$, there is at most one edge labeled a leaving s



Current	Next State		
State	l	d	Output
0	1	–	0
1	1	1	1

DFA Simulation

DFA() $\{$

$s \leftarrow s_0$;

$c \leftarrow$ nextchar();

while $c \neq$ eof $\{$

$s \leftarrow$ move(s, c); —*move is the* $\phi : (S, \Sigma) \rightarrow S$ *function*

$c \leftarrow$ nextchar();

$\}$

if $s \in F$ $\{$

return **true**;

$\}$

return **false**;

$\}$

ϵ-closure

- If $s \in S$, then ϵ-closure(s) is the set of states reachable from state s using only ϵ-transitions

- If $V \subseteq S$, then ϵ-closure(V) is the set of states reachable from some state $s \in V$ using only ϵ-transitions

ϵ-closure Computation

```
StateSet ε-closure(StateSet T) {
    result ← T;    stack ← ∅;        —stack is a stack of states
    for all s ∈ T do {
        stack.push(s);
    }
    while stack ≠ ∅ {
        t ← stack.pop();
        for each state u with an edge from t to u labeled ε do {
            if u ∉ result {
                result ← result ∪ u;
                stack.push(u);
            }
    }
    return result;
}
```

NFA Simulation

NFA() $\{$

$V \leftarrow \epsilon$-closure($\{s_0\}$);

$c \leftarrow$ nextchar();

while $c \neq$ eof $\{$

—move here returns the set of states to which there is a

—transition on input symbol c from some state $s \in V$

$V \leftarrow \epsilon$-closure(move(V, c));

$c \leftarrow$ nextchar();

$\}$

if $V \cap F \neq \varnothing$ $\{$

return **true**;

$\}$

return **false**;

$\}$

Regular Expression $\longrightarrow$ NFA

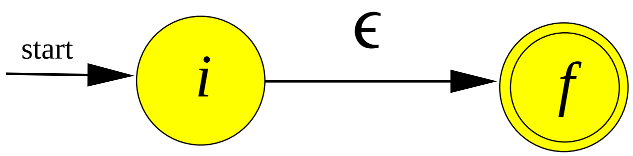
- There are several strategies to build an NFA from a regular expression
- Your book provides Thompson's method (p. 122)

1. Parse the regular expression into its basic subexpressions
   - ϵ is a basic expression
   - an alphabet symbol is a basic expression
2. Create primitive NFAs for these subexpressions
3. Guided by the regular expression operators and parentheses, inductively combine the sub-NFAs into the composite NFA representing the complete regular expression

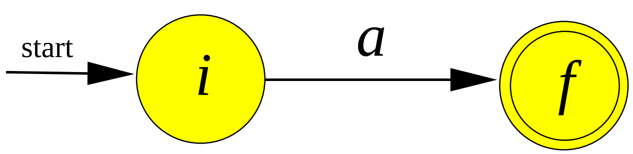
- This is a *syntax-directed* approach

Basic Expression $\longrightarrow$ Primitive NFA

For ϵ, the NFA is



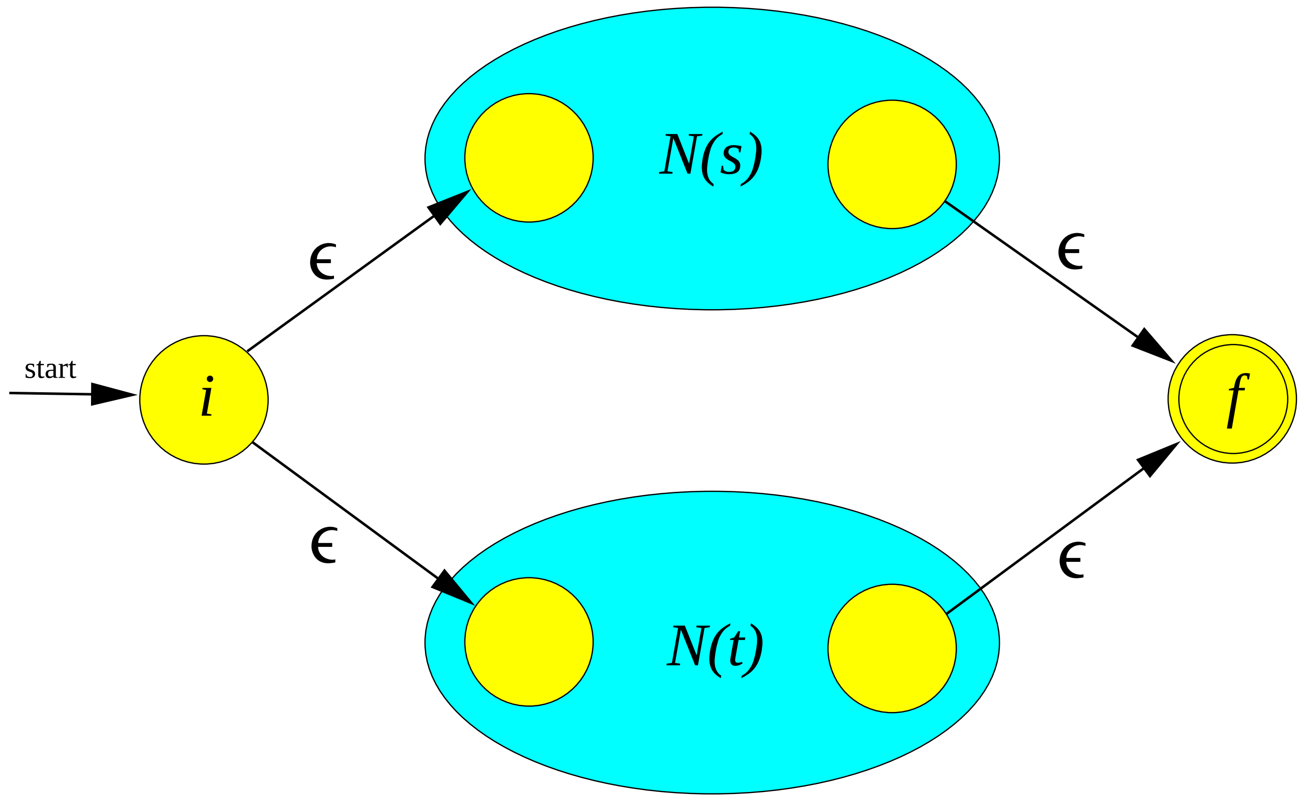
For $a \in \Sigma$, the NFA is



Observe that both of these NFAs have exactly one start state and one final state

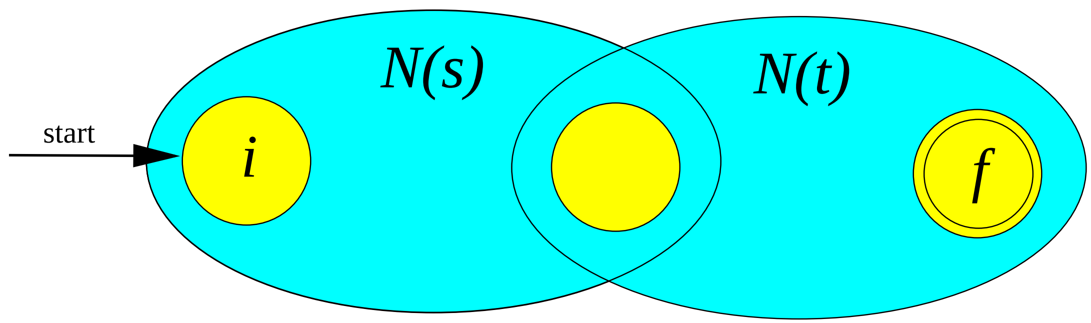
$s \mid t$

If $N(s)$ is the NFA for regular expression s, and $N(t)$ is the NFA for regular expression t, then $N(s \mid t)$ is



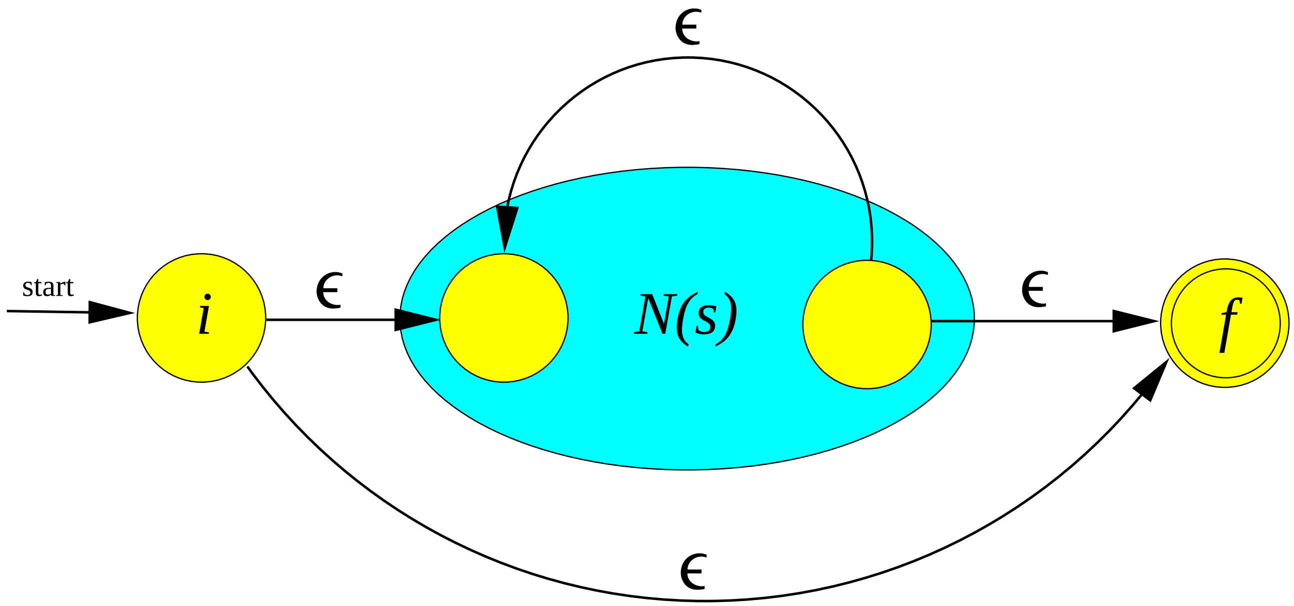
st

If $N(s)$ is the NFA for regular expression s, and $N(t)$ is the NFA for regular expression t, then $N(st)$ is



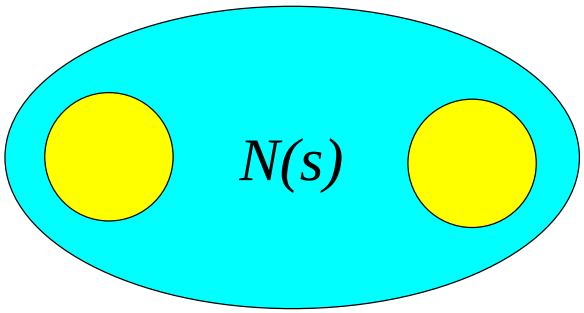
s^*

If $N(s)$ is the NFA for regular expression s, then $N(s^*)$ is



(s)

If $N(s)$ is the NFA for regular expression s, then $N((s)) = N(s)$ is



NFA $\longrightarrow$ DFA

- NFAs are difficult to simulate in a computer program

  Non-determinism on a deterministic machine

- Fortunately, any NFA can be converted into an equivalent DFA
  - A process known as *subset construction* is used to create the DFA
  - Each state in the DFA is derived from the subset of the states in the NFA
  - If the NFA has n states, its corresponding DFA may have up to 2^n states

    Fortunately, this theoretical maximum is rare in practice

Subset Construction

NFAtoDFA() $\{$

$E \leftarrow \epsilon$-closure($\{s_0\}$); E.mark $\leftarrow$ **false**; $D \leftarrow \{E\}$;

while $\exists\, T \in D$ such that T.mark = **false** do $\{$

T.mark $\leftarrow$ **true**;

for each $a \in \Sigma$ do $\{$

$U \leftarrow \epsilon$-closure(move(T, a));

if $U \notin D$ $\{$

U.mark $\leftarrow$ **false**;

$D \leftarrow D \cup U$;

$\}$

DTran[T][a] $\leftarrow U$;

$\}$

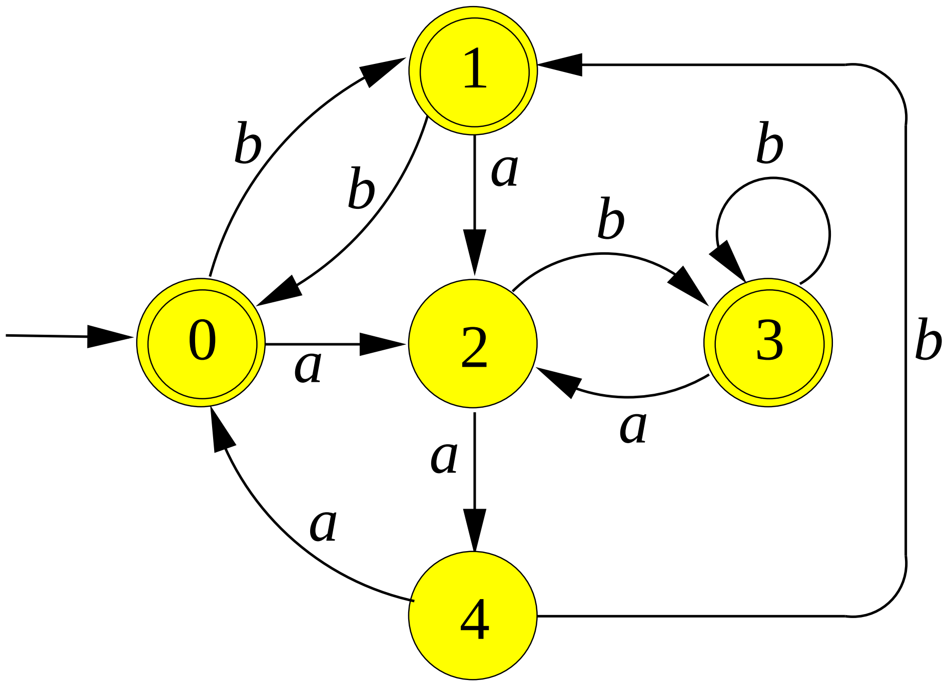
$\}$

$\}$

DFA Minimization

Goal: Given a DFA M, find a DFA M' such that M' exhibits the same external behavior as M, but M' has fewer states than M

Reason: M' will be simpler and more efficient



Current State	Next State a	Next State b	Output
0	2	1	1
1	2	0	1
2	4	3	0
3	2	3	1
4	0	1	0

DFA Minimization Procedure

1. Remove states unreachable from the start state

2. Ensure that all states have a transition on every input symbol (i.e., every element of Σ)

   - Introduce a new "dead state" d if necessary
   - $\forall a \in \Sigma$, $\phi(d,a) = d$ (i.e., `move`(d, a) = d, for all a)
   - $\forall s \in S$, if $\exists a$ such that $\phi(s,a)$ is undefined, define $\phi(s,a) = d$

3. Collapse equivalent states into a single, representative state

Equivalent States

- We say string w *distinguishes* state s from state t if

  1. starting DFA M in state s and feeding it string w we arrive at an accepting state, and
  2. starting DFA M in state t and feeding it string w we arrive at an non-final state

  or vice-versa

- $w = \epsilon$ distinguishes any final state from any non-final state

- We must find all sets of states that can be distinguished by some input string

- Two states that cannot be distinguished by any input string are called *equivalent states*

DFA Minimization Algorithm (1)

DFA minimize(DFA M) $\{$

Part 1: Find equivalent states

$\Sigma \leftarrow M.\Sigma$; *M's alphabet*

$S \leftarrow M.S$; *M's states*

$F \leftarrow M.F$; *M's final states*

$\phi \leftarrow M.\phi$; *M's transition function*

$\Pi \leftarrow \{F, S - F\}$; *Partition states into two blocks: final and non-final states*

$\Pi_{\text{old}} \leftarrow \varnothing$;

Iteratively partion the blocks until no further partitioning occurs

while $\Pi \neq \Pi_{\text{old}}$ $\{$

$\Pi_{\text{old}} \leftarrow \Pi$;

for each block $B \in \Pi$ do $\{$

Partition B into sub-blocks $B_1, B_2, \ldots, B_k$ such that two states s and t are in the same sub-block iff $\forall a \in \Sigma$ states s and t have transitions on a to states in the same block of Π;

$\Pi \leftarrow (\Pi - B) \cup \{B_1, B_2, \ldots, B_k\}$

$\}$

$\}$

DFA Minimization Algorithm (2)

Part 2: Build near-minimal DFA

$M'.\Sigma \leftarrow \Sigma$; $M'.S \leftarrow \varnothing$; $M'.F \leftarrow \varnothing$; $M'.\phi \leftarrow \varnothing$;

for each block $B \in \Pi$ do $\{$ *Basically a block in Π becomes a state in M'*

Choose one state s in B to be the *representative* of that block;

$M'.S \leftarrow M'.S \cup s$;

$\}$

for each state $s \in M'.S$ do $\{$ *Construct in the transition function for M'*

for each $a \in \Sigma$ do $\{$

if $\phi(s,a) = t$ $\{$

$M'.\phi(s,a) \leftarrow t' \in M'.S$ such that t'

is the representative state of the block in Π that contains t;

$\}$

$\}$

The start state of M' is the respresentative state of the block in Π that contains

the start state of M;

for each state $s \in M'.S$ do $\{$ *Assign final states*

if $s \in F$ $\{$ $M'.F \leftarrow M'.F \cup s$; $\}$

$\}$

DFA Minimization Algorithm (3)

Part 3: Remove superfluous states

```
    if M'.S contains a dead state d {                               Remove any dead states
        M'.S ← M'.S − d;
        for all s ∈ M'.S do {
            if ∃a ∈ Σ such that M'.φ(s,a) = d {
                M'.φ(s,a) ← undefined;
            }
        }
    for all s ∈ M'.S do {                                           Prune unreachable states
        if s is unreachable from the start state in M' {
            M'.S ← M'.S − s;
        }
    }
    return M';                                                      The minimized DFA
}
```

Minimization Example

Current	Next State		
State	a	b	Output
0	2	1	1
1	2	0	1
2	4	3	0
3	2	3	1
4	0	1	0

- a transitions are in red
- b transitions are in blue

$$\Pi_1 = \{\{2,4\},\{0,1,3\}\}$$

$$\Pi_2 = \{\{2\},\{4\},\{0,1,3\}\}$$

$$\Pi_3 = \{\{2\},\{4\},\{0,1,3\}\}$$

$$\Pi_2 = \Pi_3$$

Minimal DFA

$$\Pi_1 = \{\{2,4\},\{0,1,3\}\}$$

$$\Pi_2 = \{\{2\},\{4\},\{0,1,3\}\}$$

$$\Pi_3 = \{\{2\},\{4\},\{0,1,3\}\}$$

$$\Pi_2 = \Pi_3$$

Current	Next State		
State	a	b	Output
$0'$	$2'$	$0'$	1
$2'$	$4'$	$0'$	0
$4'$	$4'$	$0'$	0

- a transitions are in red
- b transitions are in blue
- $\{0,1,3\} \Rightarrow$ state $0'$ in M'
- $\{2\} \Rightarrow$ state $2'$ in M'
- $\{4\} \Rightarrow$ state $4'$ in M'

FAs and Regular Expressions

If $L \subseteq \Sigma^*$ is a language, the following four statements are equivalent:

1. L is a regular language
2. L can be represented by a regular expression
3. L is accepted by some NFA
4. L is accepted by some DFA

Limitations of Regular Languages

- Build a DFA to recognize

$$L = L(0^*1^*)$$

- Build a DFA to recognize

$$L = \{0^n 1^n \mid n \in \mathbb{N}\}$$

- Not all languages are regular
- See the *Pumping Lemma*

Context-free Grammars

- The syntax of programming language constructs can be described by *context-free grammars* (CFGs)

- Relatively simple and widely used

- More powerful grammars exist
  - Context-sensitive grammars (CSG)
  - Type-0 grammars

  Both are too complex and inefficient for general use

- Backus-Naur Form (BNF) and extended BNF (EBNF) are a convenient way to represent CFGs

Advantages of CFGs

- Precise, easy-to-understand syntactic specification of a programming language
- Efficient parsers can be automatically generated for some classes of CFGs
- This automatic generation process can reveal ambiguities that might otherwise go undetected during the language design
- A well-designed grammar makes translation to object code easier
- Language evolution is expedited by an existing grammatical language description

Context-free Grammar

Context-free Grammar (CFG) is a 4-tuple

$$\langle V_N, V_T, s, P \rangle$$

- V_N is a set of non-terminal symbols
- V_T is a set of terminal symbols
- s is a distinguished element of V_N called the start symbol
- P is a set of productions or rules that specify how legal strings are built

$$P \subseteq V_N \times (V_N \cup V_T)^*$$

CFG Elements

- **Terminals**: basic symbols from which strings are formed (typically corresponds to tokens from lexer)

- **Non-terminals**: syntactic variables that denote sets of strings and, in particular, denoting language constructs

- **Start symbol**: a non-terminal; the set of strings denoted by the start symbol is the language defined by the grammar

- **Productions**: set of rules that define how terminals and non-terminals can be combined to form strings in the language

$$A \rightarrow bXYz$$

Example

Symbol table interpreter

$$G = \langle V_N, V_T, s, P \rangle$$

$$\begin{array}{lcl} V_N & = & \{S\} \\ V_T & = & \{\mathbf{new}, \mathbf{id}, \mathbf{num}, \mathbf{insert}, \mathbf{lookup}, \mathbf{quit}\} \\ s & = & S \\ P & : & S \rightarrow \mathbf{new}\ \ \mathbf{id}\ \ \mathbf{num} \\ & & \quad\ \mid\ \ \mathbf{insert}\ \ \mathbf{id}\ \ \mathbf{id}\ \ \mathbf{num} \\ & & \quad\ \mid\ \ \mathbf{lookup}\ \ \mathbf{id}\ \ \mathbf{id} \\ & & \quad\ \mid\ \ \mathbf{quit} \end{array}$$

Example

An arithmetic expression language

$$G = \langle V_N, V_T, s, P \rangle$$

$$\begin{array}{lcl} V_N & = & \{E\} \\ V_T & = & \{\mathbf{id}, +, *, (,), -\} \\ s & = & E \\ P & : & E \rightarrow E + E \\ & & \quad\;\; \mid \; E * E \\ & & \quad\;\; \mid \; (E) \\ & & \quad\;\; \mid \; -E \\ & & \quad\;\; \mid \; \mathbf{id} \end{array}$$

Example

A programming language construct

$$
\begin{aligned}
\mathit{stmt} \;\; \rightarrow \;\; & ; \\
\mid \;\; & \textbf{if} \; (\; \mathit{expr} \;) \; \mathit{stmt} \; \textbf{else} \; \mathit{stmt} \\
\mid \;\; & \textbf{while} \; (\; \mathit{expr} \;) \; \mathit{stmt} \\
\mid \;\; & \mathit{blk} \\
\mid \;\; & \textbf{id} \; = \; \mathit{expr} \; ;
\end{aligned}
$$

$$
\mathit{blk} \;\; \rightarrow \;\; \{ \; \mathit{stmt}^{*} \; \}
$$

Regular Languages and CFLs

- All regular languages are context-free
- Consider the regular expression

$$a^*b^*$$

Let $G = \langle \{A, B\}, \{a, b\}, A, \{A \rightarrow aA \mid B, B \rightarrow bB \mid \epsilon\} \rangle$

Producing a Grammar from a Regular Language

1. Construct an NFA from the regular expression

2. Each state in the NFA corresponds to a non-terminal symbol

3. For a transition from state A to state B given input symbol x, add a production of the form

$$A \rightarrow xB$$

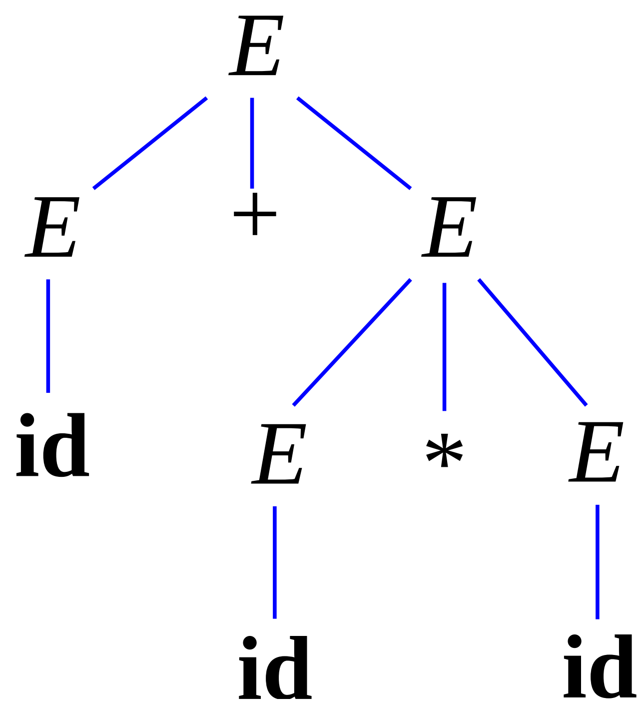
4. If A is a final state, add the production

$$A \rightarrow \epsilon$$

Parse Trees

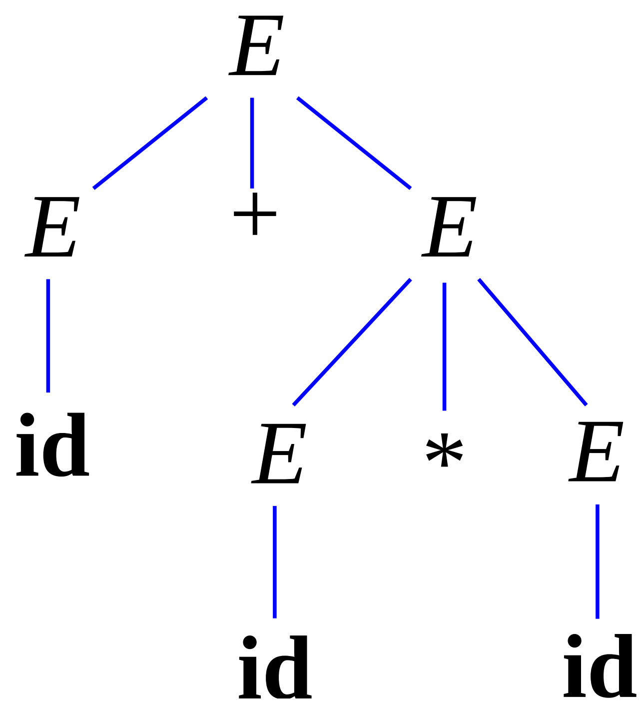
- A graphical representation of a sequence of derivations

- Each interior node is a non-terminal and its children are the right side of one of the non-terminal's productions

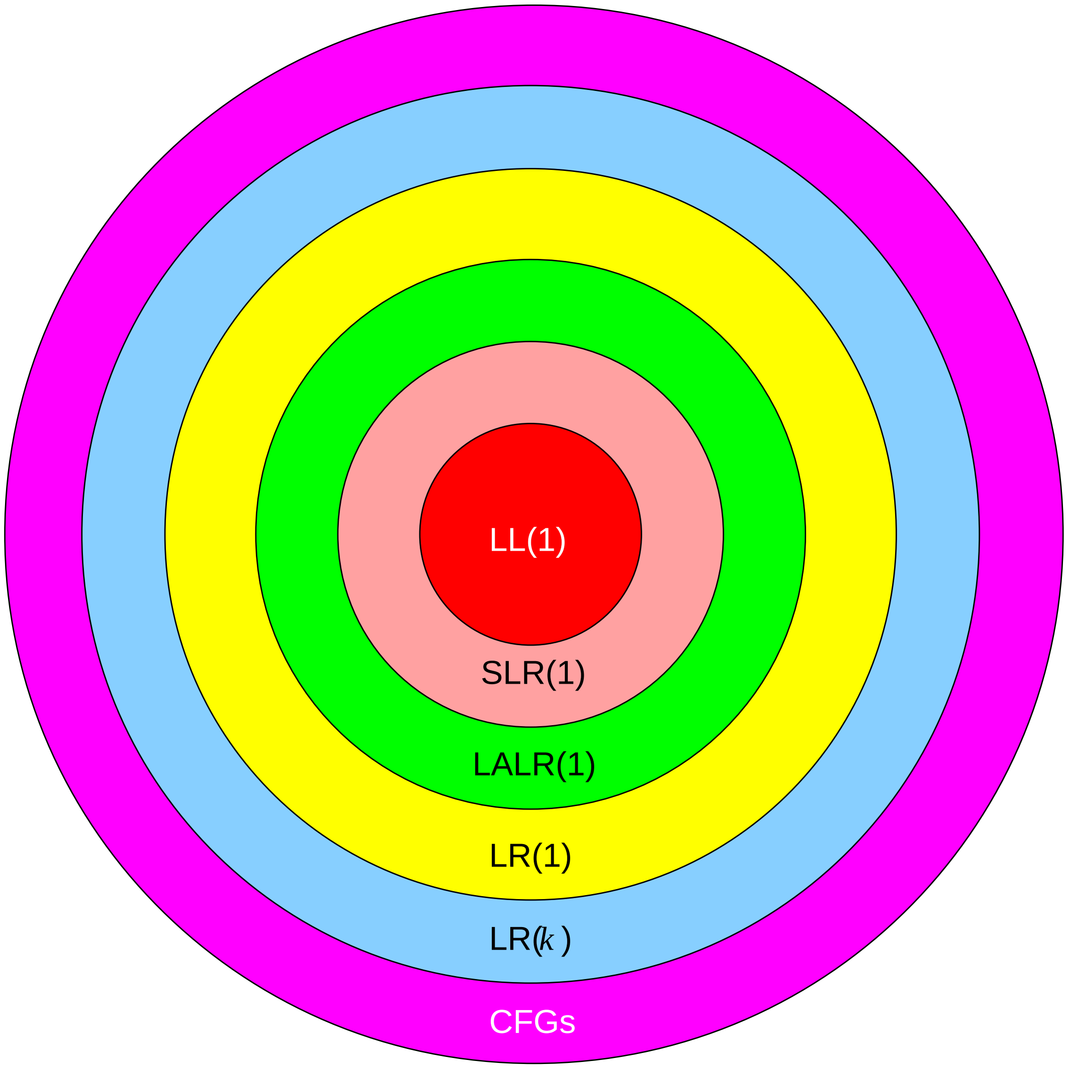


Parse Trees

- If you read the leaves of the tree from left to right they form a sentential form
  - Also called the "yield" or "frontier" of the parse tree
- All the leaves need not be terminals; the parse tree may be incomplete
- Valid sentential forms *can* contain non-terminals



Comparing Context-free Grammars



Chomsky's Grammar Hierarchy

Consider productions of the form $\alpha \to \beta$

Type	Name	Criteria	Recognizer
Type 3	Regular	$A \to a \mid aB$	Finite automaton
Type 2	Context-free	$A \to \alpha$	Push-down automaton
Type 1	Context-sensitive	$\lvert\alpha\rvert \leq \lvert\beta\rvert$	Linear bounded automaton
Type 0	Unrestricted	$\alpha \neq \epsilon$	Turing machine

Grammar Hierarchy