Principles of Compiler Construction

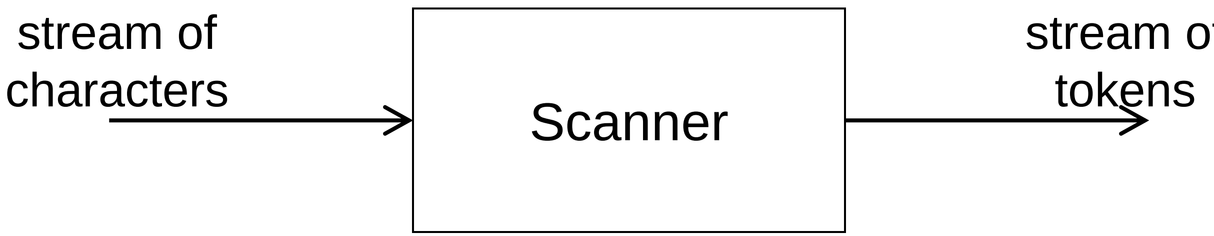
Prof. Wen-jun LI
School of Computer Science and Engineering
lnslwj@mail.sysu.edu.cn

Lecture 3. Lexical Analysis

1. Introduction
2. Scanner Construction
3. Lexical Specification
4. Finite Automata
5. Transformation and Equivalence
6. Limits of Regular Languages
7. Lexical Analysis in Practice

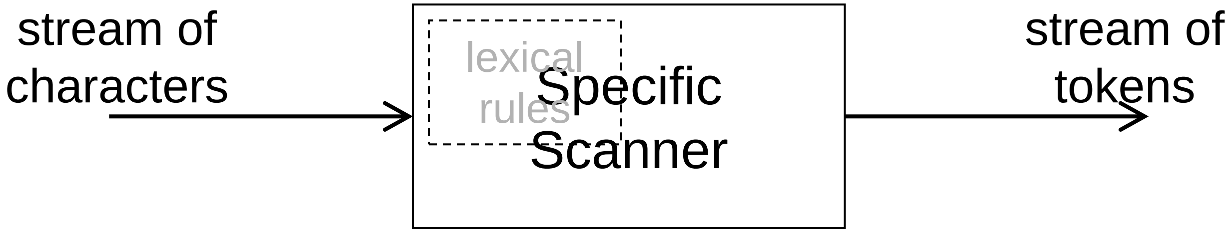
1. Introduction

- Software Architecture: Pipes and Filters



Structure of a Scanner

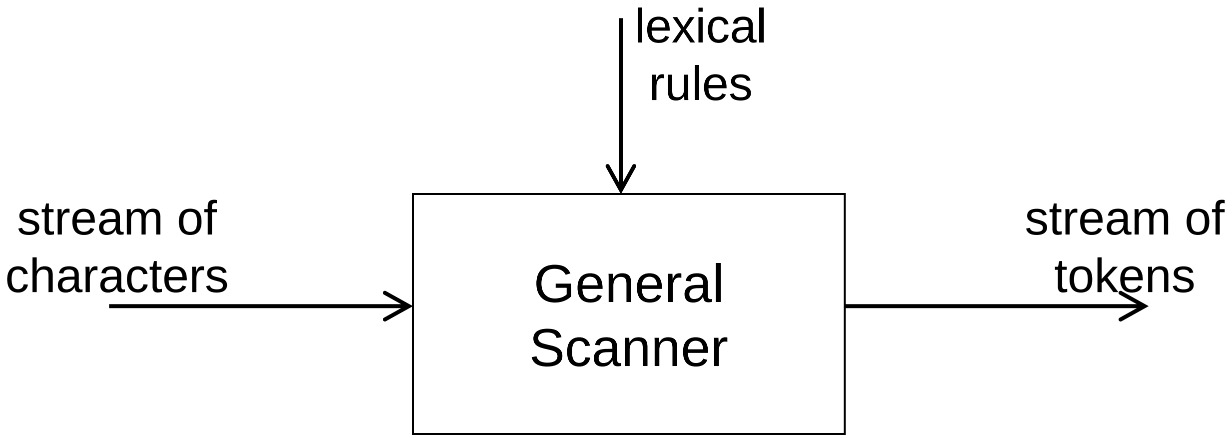
- Implicit lexical rules



Specific to some predefined language.

Structure of a Scanner (cont')

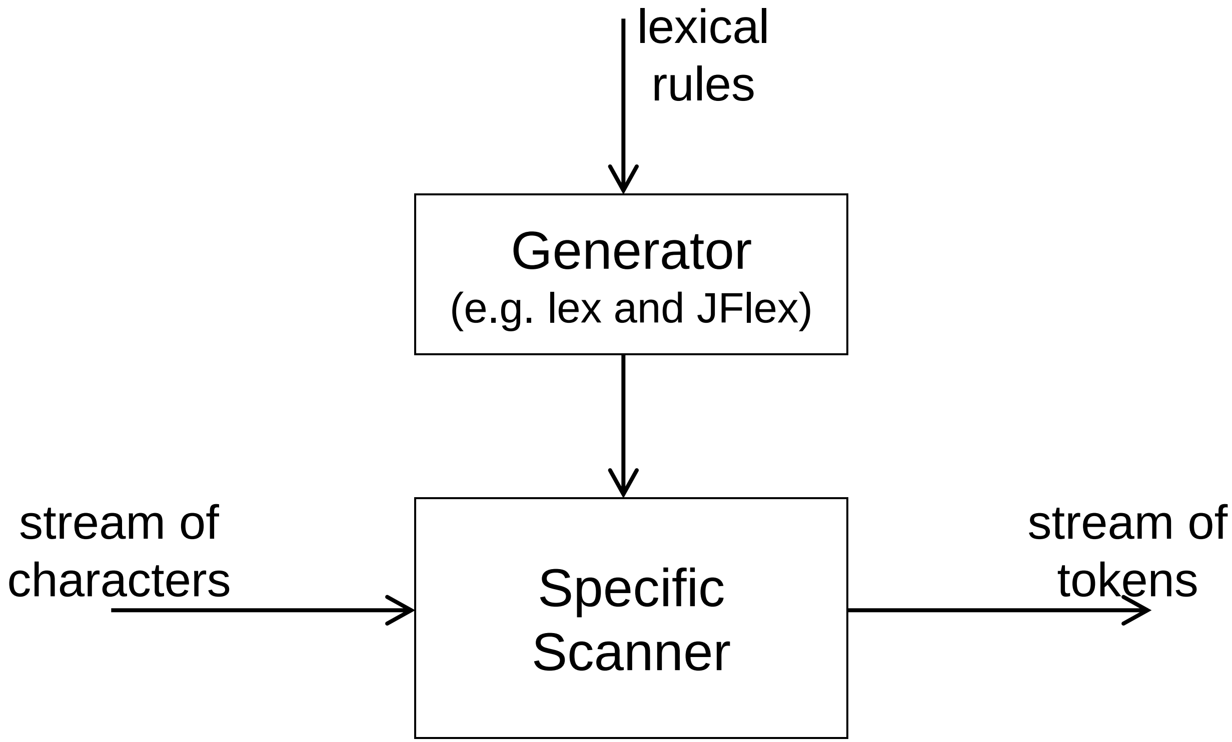
- Explicit lexical rules (interpretation model)



No hard-coding of language-specific code.

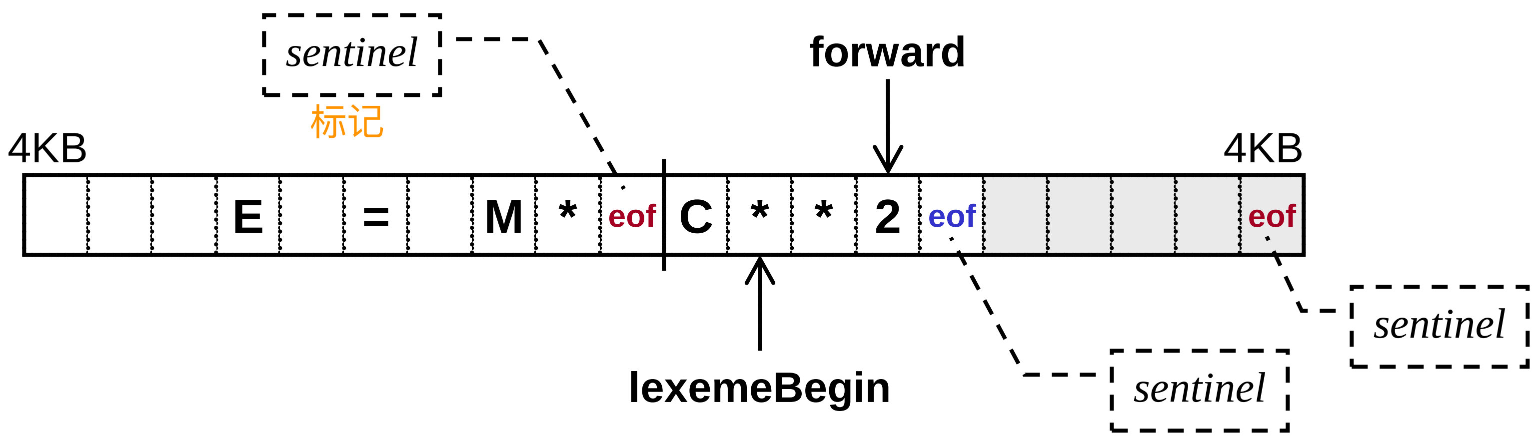
Structure of a Scanner (cont')

- Explicit lexical rules (compilation model)



Input Buffering

- Most runtime of the front end of a compiler is spent on scanning.
  - But the most time consuming stage is optimization.
- Buffer Pairs
  - 2-buffer scheme: each buffer is of size N.
  - Avoid overwriting: $|lexeme| + |lookaheads| \leq N$



Output Tokens

- 5 kinds for most modern programming languages
  - Identifiers: getBalance, weight, ...
  - Reserved words: IF, ELSE, WHILE, ...
  - Constants: 10, 3.14, -1.26E-5, 'a', "abc", ...
  - Operators: +, -, *, /, <<, ...
  - Punctuation: (,), ;, :, ...
- Output
  - A Pair of <kind, associatedAttributeValues>
    - Some values are pointers to the symbol table.
  - Symbol (string) table
    - Array, LinkedList, HashSet, TreeSet, ...

Interaction with Parser

- Aim at reducing passes
- 2 levels of abstraction
  - Logically
    - A pipe through which tokens are transferred.
  - Physically
    - A disk file of the token sequence
    - Concurrent threads or co-routines
    - Method invocation

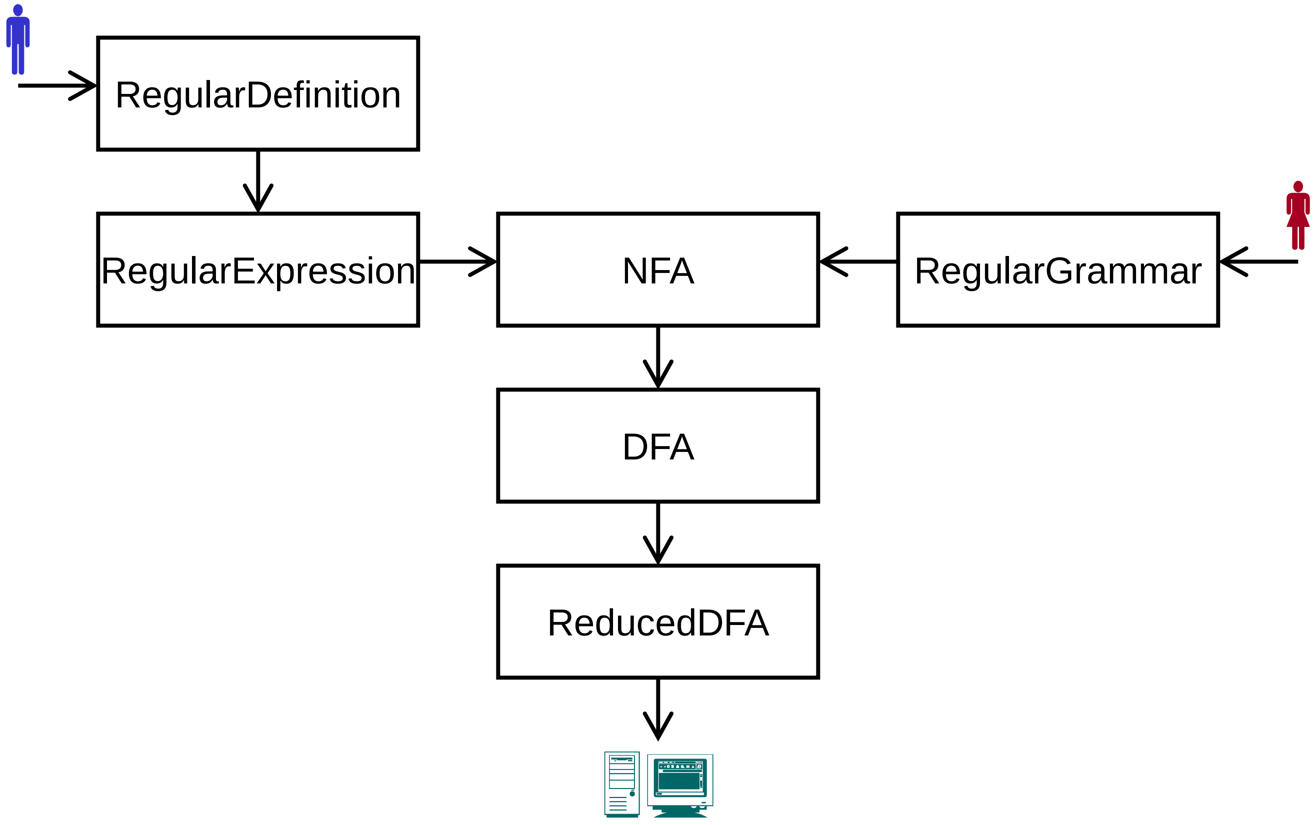
2. Scanner Construction

- Definition of lexical rules: 3 equivalent notations
  - Expressions
    - Regular expression
    - Regular definition
  - Grammars
    - Regular grammar
    - Left/right linear grammar
  - Finite automata
    - Deterministic Finite Automata (DFA)
    - Nondeterministic Finite Automata (NFA)

Begin with Formal Specification

- Specification of tokens
  - Regular definition
    - Hierarchical and brief, suitable for **humans**.
    - Then transferred to regular expressions.
    - Then transferred to finite automata.
  - Regular grammar
    - Compliant with the spec. of syntax rules.
    - Then transferred to finite automata.
- Transferring finite automata
  - NFA → DFA → Reduced DFA
  - Suitable for implementation on a **machine**.

Steps of Construction



Programming a Scanner

- Implicit transition diagram
  - A manual approach
  - State transitions are hard-coded in the program.
- Explicit transition diagram
  - A table-driven approach
  - No hard-coding of specific lexical rules
  - Lead to automatic scanner generation.

3. Lexical Specification

- Regular Expression
- Regular Definition
- Regular Grammar
- Transformation and Equivalence

Regular Expression

- Regular expression: constructively defined
  - Basis:
    - $\boldsymbol{\varepsilon}$ is a regular expr; $L(\boldsymbol{\varepsilon}) = \{\varepsilon\}$.
    - **a** is a regular expr if $a \in \Sigma$; $L(\mathbf{a}) = \{a\}$.
  - Induction: if **r** and **s** are regular expressions,
    - **r | s** is a regular expr; $L(\mathbf{r} \mid \mathbf{s}) = L(\mathbf{r}) \cup L(\mathbf{s})$.
    - **r s** is a regular expr; $L(\mathbf{r}\,\mathbf{s}) = L(\mathbf{r})\,L(\mathbf{s})$.
    - $\mathbf{r}^*$ is a regular expr; $L(\mathbf{r}^*) = (L(\mathbf{r}))^*$.
    - **(r)** is a regular expr; $L(\mathbf{(r)}) = L(\mathbf{r})$.
  - Extensions:
    - $\mathbf{r}^+ = \mathbf{r}\,\mathbf{r}^* = \mathbf{r}^*\,\mathbf{r}$ $\qquad$ $\mathbf{r}^* = \mathbf{r}^+ \mid \varepsilon$
    - $\mathbf{r}^? = \mathbf{r} \mid \varepsilon$
    - **[ace]** = **a | c | e** $\qquad$ **[A–Z]** = **A | B | … | Z**

Regular Expression Language

- Regular expression: a language
  - Syntax
    - **a (a | b) b**
    - **a (a | b)***
  - Semantics
    - {aab, abb}
    - {a, aa, ab, aaa, aab, aba, abb, ...}
- Language is an alternative approach to problem solving.

Discussions

- Signature of an operator
  - true: $\rightarrow$ bool
  - +: int $\times$ int $\rightarrow$ int
  - +: real $\times$ real $\rightarrow$ real (overloading)
- What is the signature of the semantic function L in the previous slides ?
  - Syntactic category
  - Semantic category
  - Mapping

Alphabet: Lexical vs. Syntax

- Both scanner and parser are based over an alphabet Σ.
- But the meaning (elements) of Σ is quite different.
  - Scanner: elements in Σ are characters in source programs.
    - ASCII, EBCDIC, Unicode, ...
  - Parser: elements in Σ are tokens generated and passed by the scanner.

Algebraic Laws for Regular Expressions

- $r \mid s = s \mid r$
- $r \mid (s \mid t) = (r \mid s) \mid t$
- $r \mid r = r$
- $r(st) = (rs)t$
- $r(s \mid t) = rs \mid rt$
- $(s \mid t)r = sr \mid tr$
- $r\varepsilon = \varepsilon r = r$
- $r^*r^* = r^*$
- $r^* = \varepsilon \mid r \mid rr \mid \ldots$
- $(r^*)^* = r^*$
- $rr^* = r^*r$
- $(r^* \mid s^*)^* = (r^*s^*)^*$
- $(r^*s^*)^* = (r \mid s)^*$
- $(rs)^*r = r(sr)^*$
- $(r \mid s)^* = (r^*s)^*r^*$

Discussion: what does it mean that two regular expressions are equivalent ?

Binding to Practice

- Identifiers in Pascal

(a | b | ... | z | A | B | ... | Z) ((a | b | ... | z | A | B | ... | Z) | (0 | 1 | ... | 9))*

- Hard to read and write

Regular Definition

- In the form of
  - $d_1 \rightarrow r_1$
  - $d_2 \rightarrow r_2$
  - ...
  - $d_n \rightarrow r_n$
    - where $d_i \notin \Sigma \wedge (d_i = d_j \Rightarrow i = j)$, and
    - r_i is a regular expr over $\Sigma \cup \{d_1, d_2, ..., d_{i-1}\}$

Binding to Practice

- Identifiers in Pascal (extended)
  - letter_ $\rightarrow$ A | B | … | Z | a | b | … | z | _
  - digit $\rightarrow$ 0 | 1 | … | 9
  - id $\rightarrow$ letter_ (letter_ | digit)*
- Real numbers in Pascal
  - digit $\rightarrow$ 0 | 1 | … | 9
  - digits $\rightarrow$ digit digit*
  - optionalFraction $\rightarrow$. digits | ε
  - optionalExponent $\rightarrow$ (E (+ | - | ε) digits) | ε
  - real $\rightarrow$ digits optionalFraction optionalExponent
- Hierarchical representation of regular expression

Binding to Practice (cont')

- Writing a regular definition with extended operators
  - digit $\rightarrow$ [0–9]
  - digits $\rightarrow$ digit^{+}
  - real $\rightarrow$ digits (. digits)$^{?}$ (E [+-]$^{?}$ digits)$^{?}$

Note the difference between '–' and '-'

Transforming between Regular Definitions and Regular Expressions

- Regular definition → regular expression
  - Top-down, stepwise substitution
  - **Discussion**: does the substitution process always terminate ?
- Regular expression → regular definition
  - Simply add a new symbol and "$\rightarrow$".
- Equivalence
  - Defined by the language they denote
  - Bi-direction

Transforming Regular Definitions to Regular Grammars

- Difficulties arise from the closure operator
  - Transform to a left/right recursive production.
- In regular definition
  - id $\rightarrow$ letter (letter | digit)*
- Introduce a right part
  - id $\rightarrow$ letter rid
  - rid $\rightarrow$ rid (letter | digit) | ε (not permitted in Grammar)
    that is, rid $\rightarrow$ rid letter | rid digit | ε
  - or right-linear, rid $\rightarrow$ letter rid | digit rid | ε

Class 3 Grammars

- Right Linear (Regular) Grammar
  - $A \rightarrow aB$ or $A \rightarrow a$, where $A, B \in N \wedge a \in \Sigma \cup \{\varepsilon\}$.
- Left Linear Grammar
  - $A \rightarrow Ba$ or $A \rightarrow a$, where $A, B \in N \wedge a \in \Sigma \cup \{\varepsilon\}$.
- Extended Forms
  - Right Linear (Regular) Grammar
    $A \rightarrow \alpha B$ or $A \rightarrow \alpha$, where $A, B \in N \wedge \alpha \in \Sigma^*$.
  - Left Linear Grammar
    $A \rightarrow B\alpha$ or $A \rightarrow \alpha$, where $A, B \in N \wedge \alpha \in \Sigma^*$.

4. Finite Automata

- 3 types of finite automata
  - DFA (Deterministic Finite Automata)
  - NFA (Nondeterministic Finite Automata)
  - ε-NFA (Nondeterministic Finite Automata with empty transitions)
- Languages defined by a finite automaton
- Representation of finite automata
  - For humans: Transition Diagrams
  - For computers/machines: Transition Tables

Formal Definition

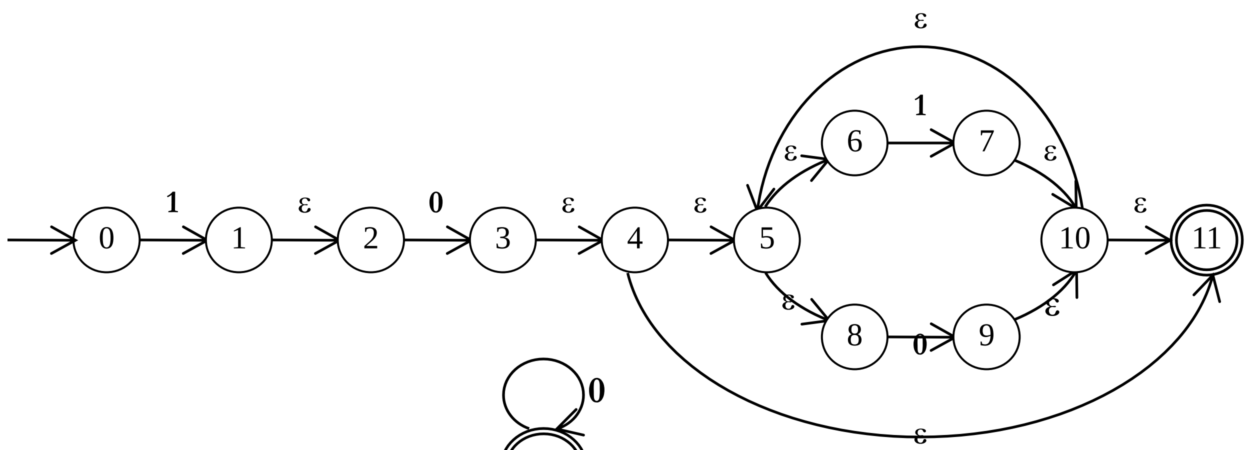
- A finite automaton is a 5-tuple: $M = (\Sigma, S, \Delta, s_0, F)$, where
  - Σ is the input alphabet.
  - $S \cap \Sigma = \varnothing$ is a **finite** set of states.
  - Δ is a transition function.
  - $s_0 \in S$ is the start (initial) state.
  - $F \subseteq S$ is a set of final (accepting) states.
- DFA, NFA, and ε-NFA only differ in
  - For a DFA, $\delta : S \times \Sigma \to S$
  - For an NFA, $\delta : S \times \Sigma \to 2^S$
  - For an ε-NFA, $\delta : S \times (\Sigma \cup \{\varepsilon\}) \to 2^S$

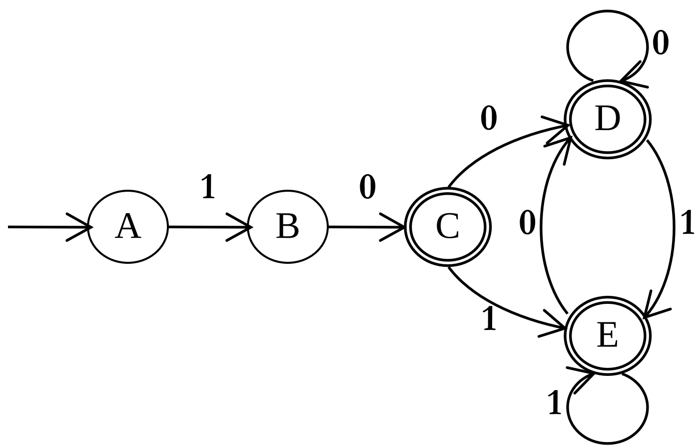
Language Defined by an FA

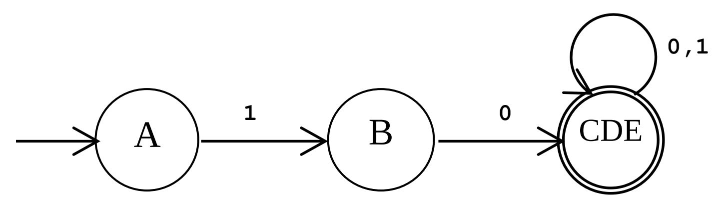
- **Overloading** the transition function
  - For any $a \in \Sigma$, $s \in S$ and $\omega \in \Sigma^*$,
    - $\delta(s, \varepsilon) = s$
    - $\delta(s, \omega a) = \delta(\delta(s, \omega), a)$
- Given a finite automaton M,
  - $L(M) = \{\omega \mid \omega \in \Sigma^* \wedge \exists s \in F.\ \delta(s_0, \omega) = s\}$

Transition Diagrams

- Graphical representation of FA

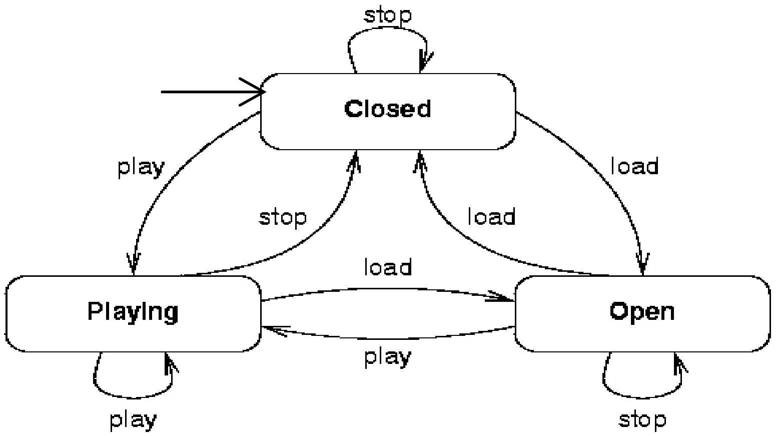






Commonly Used in Practice

- Description of the state transition of a stateful object



- UML (Unified Modeling Language)
  - De facto standard for object-oriented modeling and design.
  - Statechart of a class.

Internal Representation

- Transition table
  - Rows
    - DFA, NFA, and ε-NFA: State S
  - Columns
    - DFA and NFA: Alphabet Σ
    - ε-NFA: $\Sigma \cup \{\varepsilon\}$
  - Cells
    - DFA: an element of S
    - NFA and ε-NFA: a subset of S
- That is why NFA & ε-NFA are not suitable for implementation.

5. Transformation and Equivalence

- Regular Grammar to ε-NFA
- Regular Expression to ε-NFA
- Determination (of ε-NFA)
- Reduction (of DFA)

Regular Grammar to ε-NFA

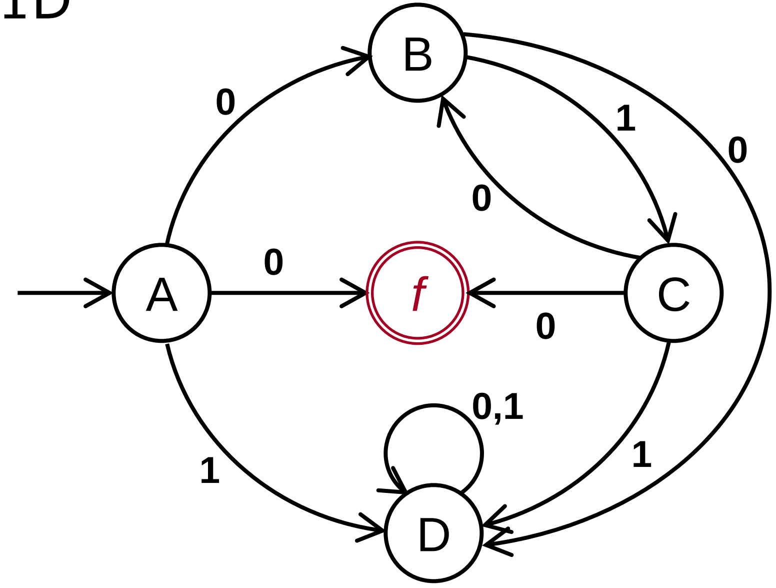
- Algorithm
  - Add a new final state *f*
  - For any $a \in \Sigma \cup \{\varepsilon\}$ and $A, B \in N$,
    - If $A \rightarrow a$ where $a \in \Sigma \cup \{\varepsilon\}$, let $\delta(A, a) = f$
    - If $A \rightarrow aB$, let $\delta(A, a) = B$

An Example

- Given a regular grammar

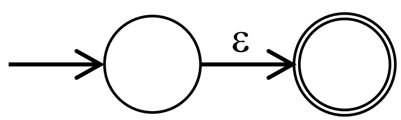
$$A \rightarrow 0 \mid 0B \mid 1D$$
$$B \rightarrow 0D \mid 1C$$
$$C \rightarrow 0 \mid 0B \mid 1D$$
$$D \rightarrow 0D \mid 1D$$

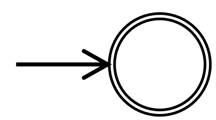
- We have an equivalent ε-NFA



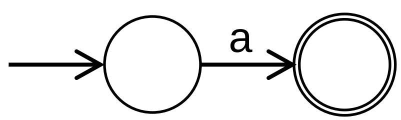
Regular Expression to ε-NFA

- Thompson's construction (McNaughton-Yamada-Thompson algorithm)
  - A constructive approach corresponding with the constructive definition of regular expressions
  - Basis:
    - ε

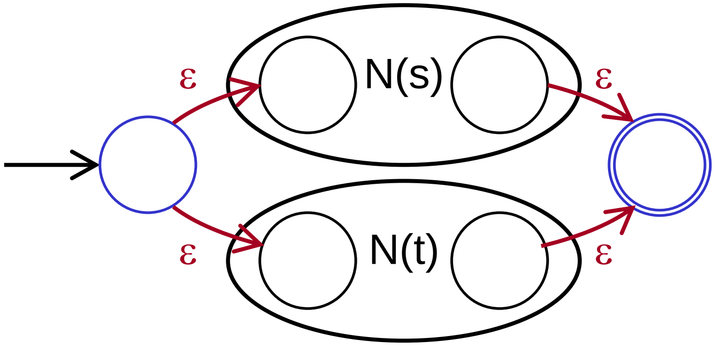




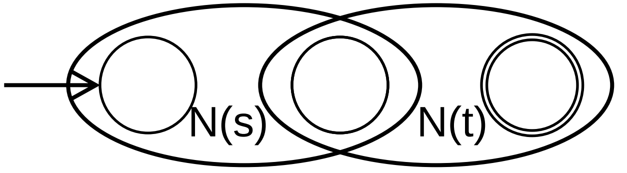
    - a $\in \Sigma$



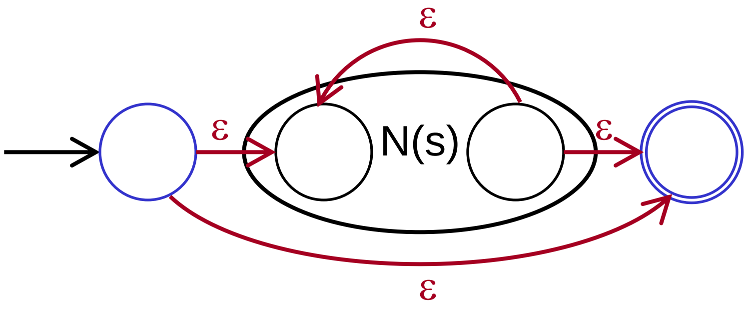
- Induction:
  - r = s | t
    2 new states
    4 new ε transitions



  - r = s t
    No new states
    or transitions

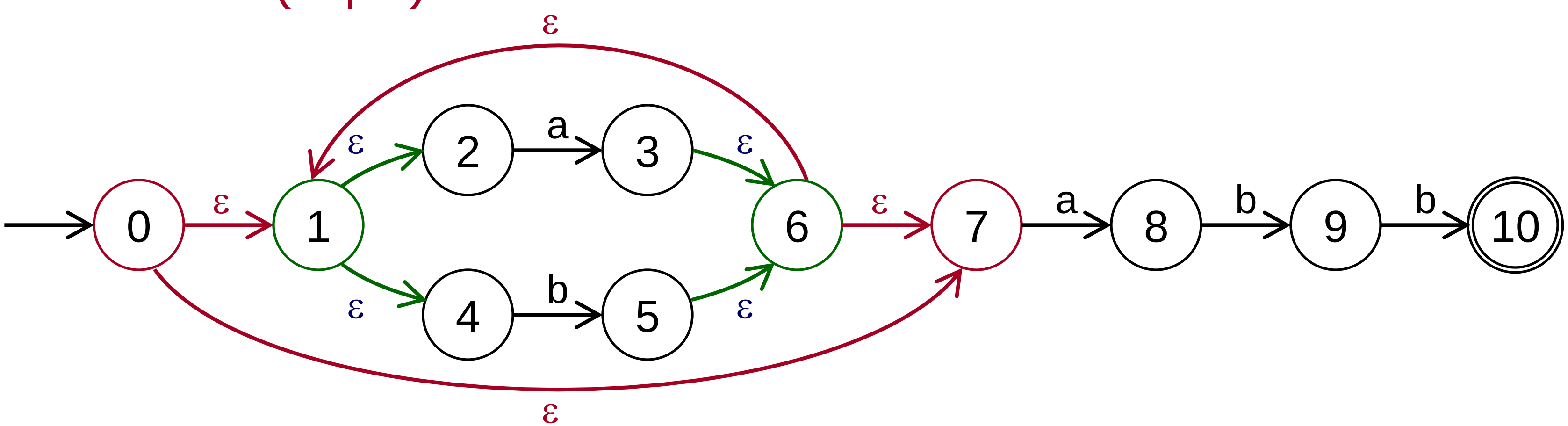


  - r = s*
    2 new states
    4 new ε transitions



An Example

- Given regular expression
  - $(a \mid b)^*abb$
- We have
  - $a \mid b$
  - $(a \mid b)^*$



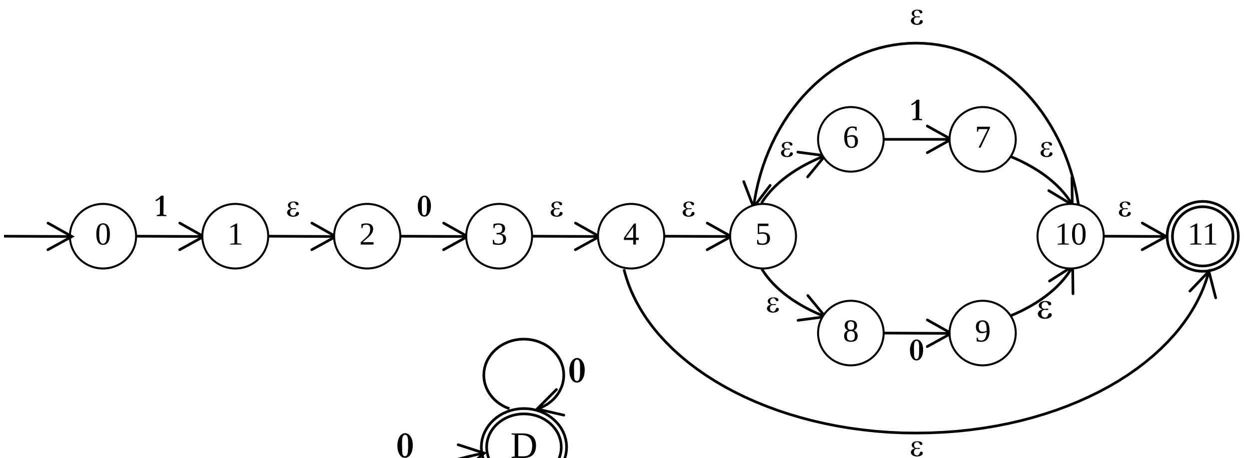
ε-NFA to DFA

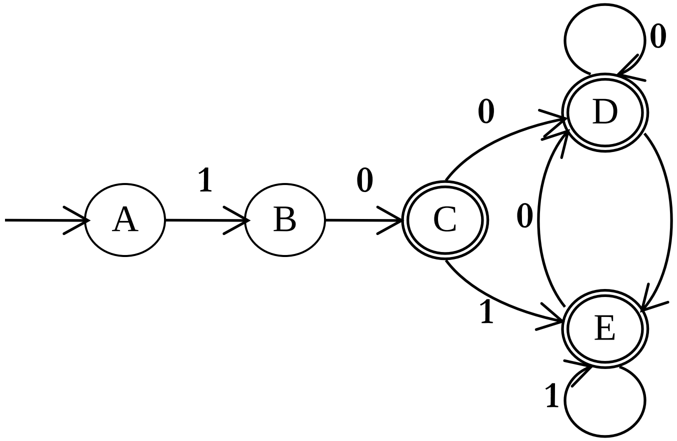
- Both 2 approaches are applied
  - Subset construction
  - ε-closure construction
- Subset
  - A state in the new DFA corresponds to a subset of states in the original NFA.
- ε-closure
  - The empty transition does not consume any input.

An Example

A = {0}, B = {1, 2}, C = {3, 4, 5, 6, 8, **11**}

D = {9, 10, **11**, 5, 6, 8}, E = {7, 10, **11**, 5, 6, 8}





Calculate the ε-closure

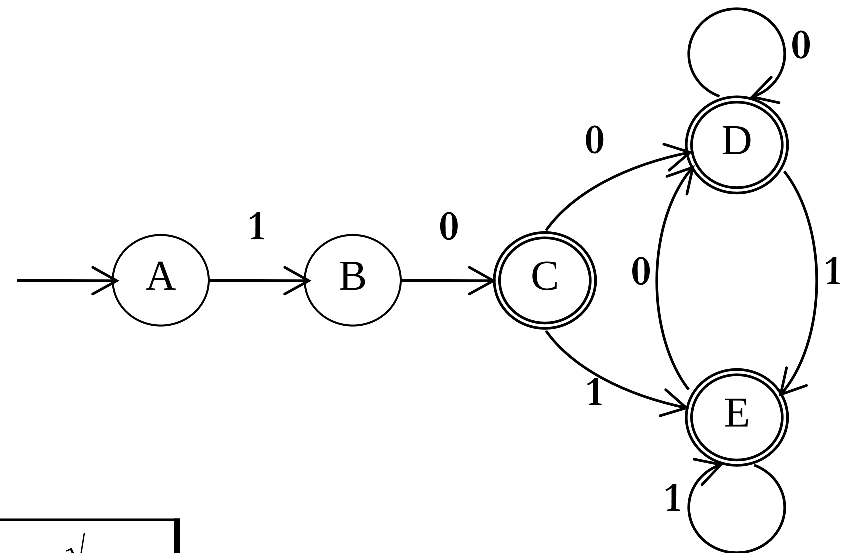
- Let ε-closure($\{0\}$) = $\{0\}$ = A, then
  - δ(A, 1) = ε-closure($\{1\}$) = $\{1, 2\}$ = B
  - δ(B, 0) = ε-closure($\{3\}$) = $\{3, 4, 5, 6, 8, 11\}$ = C
  - δ(C, 0) = ε-closure($\{9\}$) = $\{5, 6, 8, 9, 10, 11\}$ = D
  - δ(C, 1) = ε-closure($\{7\}$) = $\{5, 6, 7, 8, 10, 11\}$ = E
  - δ(D, 0) = ε-closure($\{9\}$) = D
  - δ(D, 1) = ε-closure($\{7\}$) = E
  - δ(E, 0) = ε-closure($\{9\}$) = D
  - δ(E, 1) = ε-closure($\{7\}$) = E

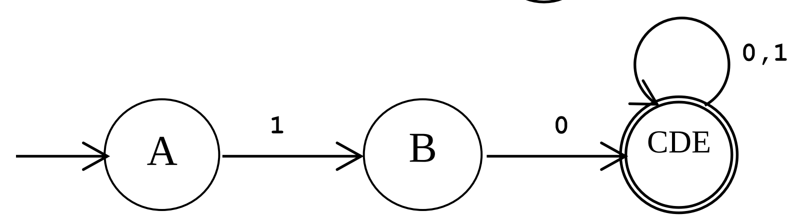
Reduction of DFA

- Find out equivalent classes of states
  - Approach 1:
    - Use a table to represent the equivalent relationship
    - Only a triangle is needed.
  - Approach 2:
    - Use partitions of the set of states

An Example

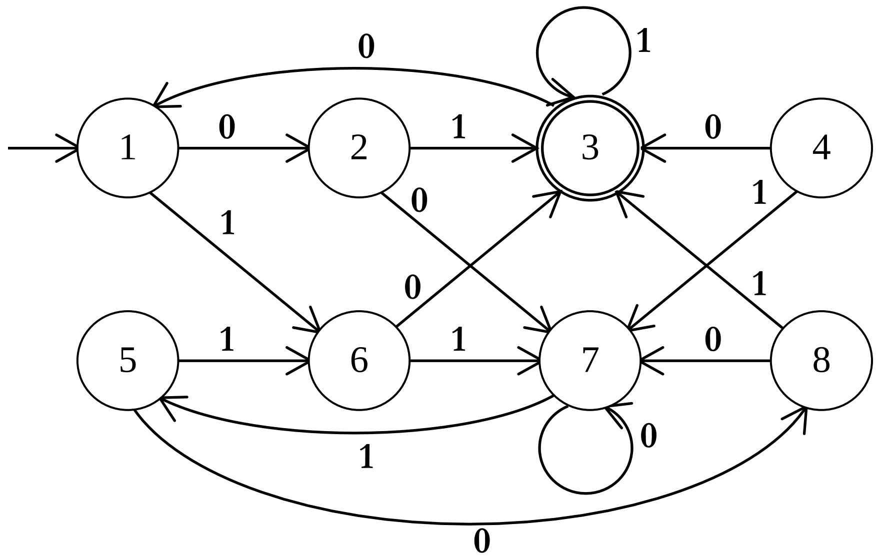
B	$\times_2$ $\delta(A, 1) = B$ $\delta(B, 1) = \perp$			
C	$\times_1$ Final	$\times_1$ Final		
D	$\times_1$ Final	$\times_1$ Final	$\surd_3$ $\delta(C, 0) = D$ $= \delta(D, 0)$ $\delta(C, 1) = E$ $= \delta(D, 1)$	
E	$\times_1$ Final	$\times_1$ Final	$\surd_4$ $\delta(C, 0) = D$ $= \delta(E, 0)$ $\delta(C, 1) = E$ $= \delta(E, 1)$	$\surd_5$ $\delta(D, 0) = D$ $= \delta(E, 0)$ $\delta(D, 1) = E$ $= \delta(E, 1)$
	A	B	C	D

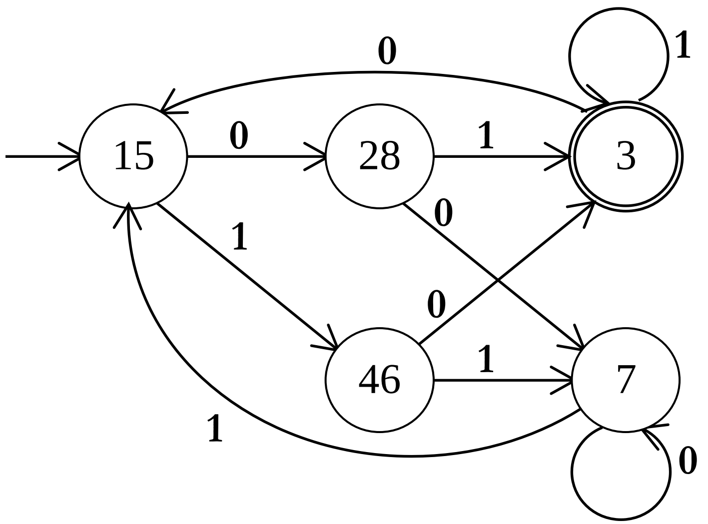




More Example

- {1, 2, 4, 5, 6, 7, 8} {3} (final and non-final, that is from ε)
- {1, 4, 5, 6, 7} {2, 8} {3} (from 1 to final and non-final)
- {1, 5, 7} {4, 6} {2, 8} {3} (from 0 to final and non-final)
- {1, 5} {7} {4, 6} {2, 8} {3} (from 0 to state {2, 8} and {7})
  {1, 5} {7} {4, 6} {2, 8} {3} (or from 1 to state {6} and {5})
- No difference found, terminate.





6. Limits of Regular Languages

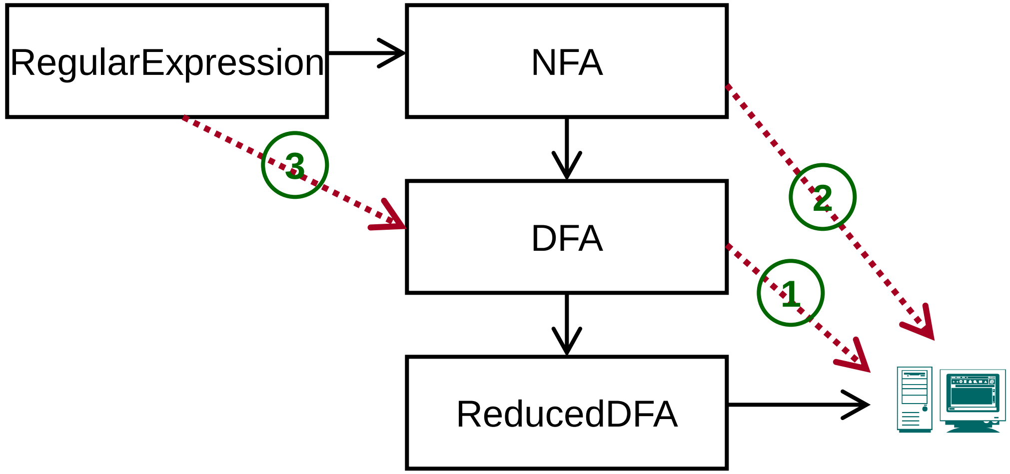
- Regular languages are not able to handle infinite counting and matching
  - e.g. $\{a^n b c^n \mid n \geq 1\}$ is NOT a regular set.
- The matching of parentheses in most programming languages
  - Can not be described by regular expressions
  - Can not be generated by regular grammars
  - Can not be recognized by finite automata

7. Lexical Analysis in Practice

- Scanner in practice: trained in our lab-time
  - Construct a scanner manually
    - Define the lexical rules
    - Construct a finite automaton
    - Write a scanner based on the blueprint of the finite automaton
  - Construct a scanner with a scanner generator
    - lex, Flex, and JFlex

String Processing

- Regular expression is a common way to describe a pattern in a string.
  - Trade-off and consequence in the implementation of regular expressions
    - Recognizing with an NFA
    - Regular expression to DFA directly



Exercise 3.1

- Give the recognized tokens of the following program in Pascal.

```
function max(i, j: integer): integer;
{return the maximum of integers i and j}
begin
  if i > j then max := i else max := j
end;
```

Exercise 3.2

- (DBv2, Ch.3, pp.125, ex.3.3.2) Describe the languages denoted by the following regular expressions:
  - $a\ (a \mid b)^*\ a$
  - $a^*\ b\ a^*\ b\ a^*\ b\ a^*$

Exercise 3.3

- (DBv2, Ch.3, pp.125, ex.3.3.4) Most Languages are case sensitive, so keywords can be written only one way, and the regular expressions describing their lexemes are very simple.
- However, some languages, like Pascal and SQL, are case insensitive. For example, the SQL keyword **`SELECT`** can also be written **`select`**, **`Select`**, or **`sELEcT`**.
- Show how to write a regular expression for a keyword in a case insensitive language. Illustrate your idea by writing the expression for **`SELECT`** in SQL.

Exercise 3.4

- Given the following regular expression

$$1^*(0 \mid 01)^*$$

(1) Transform it to an equivalent finite automaton.

(2) Construct an equivalent DFA for the result of exercise (1).

(3) Reduce the result of (2) and get a reduced DFA.

Exercise 3.5**

- Given the alphabet $\Sigma = \{ z, o, / \}$, a comment in a program over Σ begins with "/o" and ends with "o/". Embedded comments are not permitted.

  (1) Draw a DFA that recognizes nothing but all the comments in the source programs.

  (2) Write a single regular expression that exactly describes all the comments in the source programs.

Further Reading

- Dragon Book, 2nd Edition (DBv2)
  - Comprehensive reading:
    - Section 2.6 and 3.1–3.2 for introduction to scanner.
    - Section 3.3 for regular expressions and regular definitions.
    - Section 3.6–3.7, 3.9.6 for finite automata and related transformation.
  - Skip reading:
    - Section 3.4–3.5 and 3.8 for scanner generator.
    - Section 3.9.1–3.9.5 for regular expressions directly to DFAs.

Enjoy the Course!