Chapter 3

Scanning – Theory and Practice

Overview of scanner

- A scanner transforms a character stream of source file into a token stream.
- It is also called a **lexical analyzer**.
- Formal definitions allow a language designer to anticipate design flaws such as:
  - Virtually all languages specify certain kinds of **rational constants**. Such constants are often specified using decimal numerals such as 0.1 and 10.01.
  - Can .1 or 10. be allowed? C, C++, Java say YES
    But, Pascal and Ada say NO
    Why? 1..10 (range 1 to 10) would have been recognized as
    1. and .10 two contants.

What is a token anyway?

character stream

honeycone□□=5□□+□10□□*num ;

token stream

ID ‘=‘ INT ‘+’ INT ‘*’ ID ‘;’

A sample token data structure

```
struct token {
    int type ; // ID, INT, …..
    int ival ;  // integer value of a token
    string name ;  // store the token string
}
```

yacc/bison will give this data struc for you

Scanner.l

```
 %{
 #include "y.tab.h"
%}

  asgdada    {  yylval.ival = 5 ;
```

Regular expression

- **Regular expression** is a convenient way to specify various sets of strings and it can specify the structure of the tokens used in a programming language.

- A set of strings defined by a regular expression is called a **regular set**.

Regular expression (Cont.)

- The definition of regular expression starts with a finite character set, or **vocabulary** (denoted Σ)
- An **empty** (**null**) string is allowed (denoted λ). It represents an empty buffer in which no characters have yet been matched.

Regular expression (Cont.)

- Strings are built from characters in the character set Σ via **catenation**.

- As characters are catenated to a string, it grows in length.
  - For example, the string *do* is built by first catenating *d* to λ and then catenating *o* to the string *d*.
  - The null string λ, when catenated with any string s, yields s. That is, $s\lambda \equiv \lambda s \equiv s$.

Regular expression (Cont.)

- A meta-character is any punctuation character or regular expression operator.
- The following six symbols are meta-characters:

  ( ) ’ * + |

- The expression (‘(‘ | ’)’ | ; | ,) defines four single-character tokens:
  (**left parenthesis, right parenthesis, semicolon, and comma**).

Regular expression (Cont.)

- Alternation “|” can be extended to sets of strings.
  - Let P and Q be sets of strings. Then strings $s \in (P|Q)$ if, and only if, $s \in P$ or $s \in Q$.
- The operation, **Kleene closure**, is defined as:
  - The operator ***** is the postfix Kleene closure operator.
  - For example, let *P* be a set of strings. Then *P** represents all strings formed by the catenation of zero or more selections from *P*.

Regular expression (Cont.)

- 0 is a regular expression denoting the empty set (the set containing no strings).
– λ is a regular expression denoting the set that contains only the empty string.
– s is a regular expression denoting {s}: a set containing the single symbol $s \in \Sigma$

– If A and B are regular expressions,

then **A | B, AB, and A*** are also regular expressions.

They denote **3 operators**:

1) **alternation**, 2) **catenation**, and 3) **Kleene closure** of the corresponding regular sets.

Regular expression (Cont.)

- The following are additional **operators**:
  - **P+**, sometimes called positive closure, denotes all strings consisting of one or more strings in P catenated together: P*= (P+ | λ) and P+ = PP*.
  - If A is a set of characters, **Not(A)** denotes (Σ - A), that is, all characters in Σ , but not in A.
  - If *k* is a constant, then the set $\mathbf{A}^k$ represents all strings formed by catenating k (possibly different) strings from A.

Regular expression (Cont.)

- A basic pattern (such as “b”) can optionally be followed by **repetition operators**:

  b? for an optional b;
  b* for a possibly empty sequence of b;
  b+ for a non-empty sequence of b.

- There are two **composition operators**: catenation and alternatives:
  ab b follows a
  ab* | cd? ab* or cd?

Patterns of Regular Expression

Basic patterns:	*Matching:*
x	The character x
.	Any character, usually except a newline
[xyz...]	Any of the characters x, y, z, ...
Repetition operators:	
$R?$	An R or nothing (= optionally an R)
R^*	Zero or more occurrences of R
R^+	One or more occurrences of R
Composition operators:	
$R_1 R_2$	An R_1 followed by an R_2
$R_1 \| R_2$	Either an R_1 or an R_2
Grouping:	
(R)	R itself

Regular expression (Cont.)

Examples:

- (a|b)(a|b) will generate aa|ab|ba|bb
- ab* will generate a|ab|abb…
- (ab)* will generate λ | ab | abab|ababab…

The regular expression for "identifier" is:

letter → [a-z A-Z]
digit → [0-9]
underscore → _
letter_or_digit → letter | digit
underscored_tail →
underscore letter_or_digit+
identifier → letter letter_or_digit*
underscored_tail*

More Regular Expression Examples

^(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])$

matches a date in yyyy-mm-dd format from between 1900-01-01 and 2099-12-31, with a choice of four separators.

#.*$

matches a single-line comment starting with a # and continuing until the end of the line.

"[^"\r\n]*"

matches a single-line string that does not allow the quote character to appear inside the string.

^.*John.*$.

identify the whole line that contains “John” keyword.

The Applications of Regular Expression

- Regular expression is widely adopt in many libraries or programming language for you to parse strings
  - C standard library
  - C++ reg
  - Boost.regex
  - Php, python, perl…. Many many more
  - Unix tools : sed awk lex…
- It can be very handy when you need to check strings like IP address, html tags….

C standard library (long long time ago)

Yes, you can handle regular expressions at runtime. POSIX regular expressions are handled by two main functions, `regcomp()` and `regexec()` (plus `regfree()` and `regerror()`). In the example below, `regex_string` is something like "temp.*" and `string_to_match` is "temp that will match"

```
regex_t reg;
if(regcomp(&reg, regex_string, REG_EXTENDED | REG_ICASE) != 0) {
  fprintf(stderr, "Failed to create regex\n");
  exit(1);
}

if(regexec(&reg, string_to_match, 0, NULL, 0) == 0) {
  fprintf(stderr, "Regex matched!\n");
} else {
  fprintf(stderr, "Regex failed to match!\n");
}

regfree(&reg);
```

Regex in Posix

Patterns of POSIX.2 REGEX

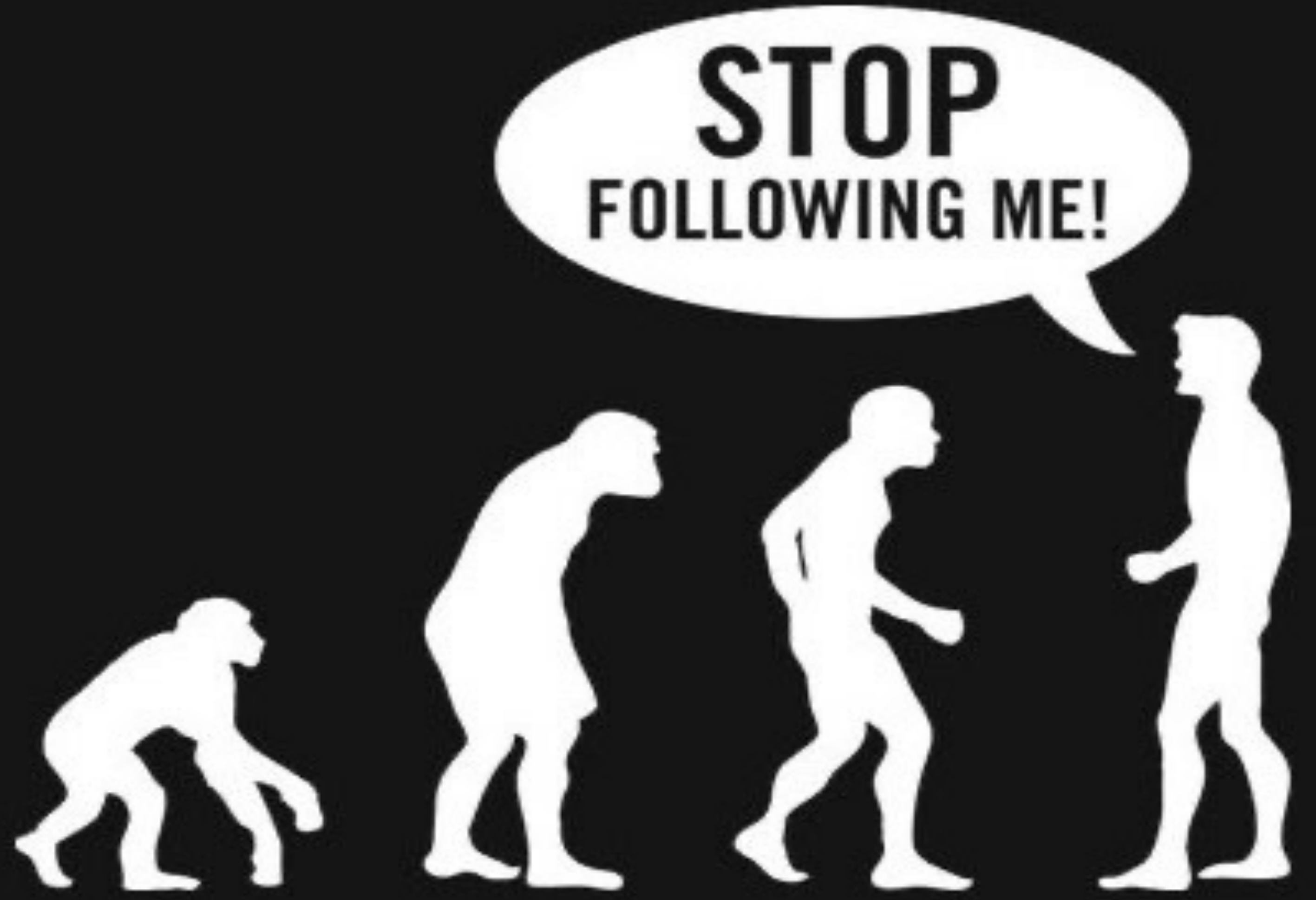
POSIX Regex 可用的規則樣式等於 PHP 的 ereg()/eregi() 。以下是一些可用的樣式規則:

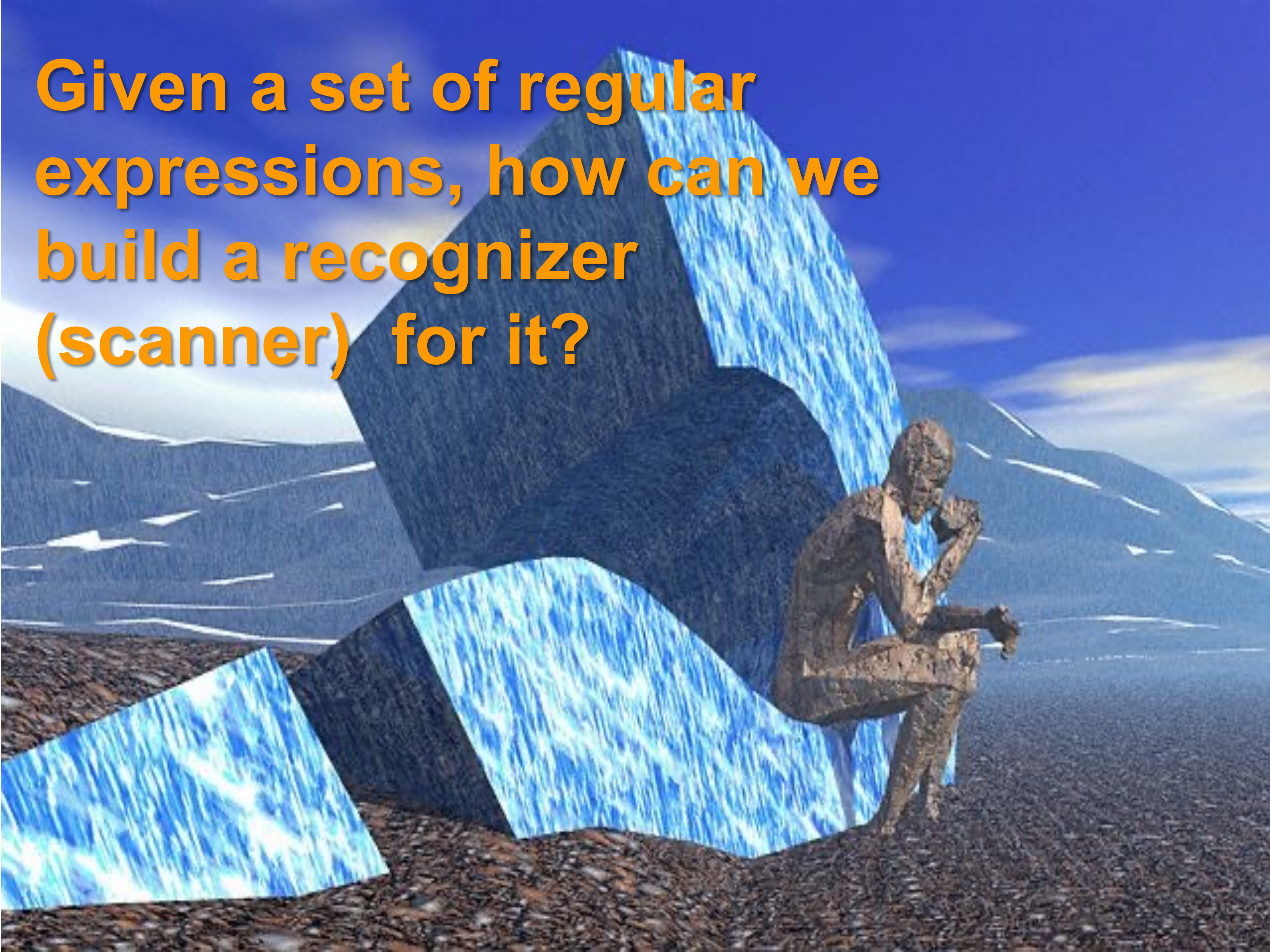
```
^   定位規則，文字開頭
$   定位規則，文字尾端
.   單一規則，代表任意字元
[chars]     單一規則，有 chars 裡其中一個字元
[^chars]    單一規則，沒有 chars 裡其中一個字元
?   倍數規則， 0 或 1 個的前導符號
*   倍數規則， 0 或多個的前導符號
+   倍數規則， 1 或多個的前導符號
{n,m}   表示前一符號在字串中的重覆次數。
    例如 A{2} 表示 'A' 重覆兩次 (即 'AA') ；
    A{2,} 表示字串含有 2 到無數多個 'A' ；
    A{2,5} 表示含有 2 到 5 個 'A' 。
\char   轉義，將 char 視為一般字元，而非樣式規則字元
(string)    子樣式規則，將 string 記憶起來，歸於一組。
            稍後可利用 \n 的方式，將第 n 組 string 提出。
```

Perl 另行擴充了一套樣式規則，如 \d, \w 等等；PHP 稱之為 PCRE。這些樣式規則不適用於此處。POSIX.2 之規則為 [:digit:], [:alnum:] 等，詳見 manpage: regex(7)。此外，PCRE 和 POSIX.2 REGEX 之敘述方式亦略有不同。PCRE 要求字樣規則前後以斜線(/)字元括起，如/[a-z]/；但 POXIS.2 則不需要，直接寫 [a-z] 即可。

Lex

- The most well-know scanner for compiler lexical analysis front-end.
  - **you need to return in every matched regular expression so yylex() will return one token at a time (otherwise, a yylex() will read the inputs until the end)**
  - **often you need to define a token data structure and store the tokens elsewhere**
- However, you can also use it for other general goals
  - ex. formatting log files into the one you need.
  - ex. file format translator
  - ex. preprocessing input data
  - …..


STOP
FOLLOWING ME!


Given a set of regular expressions, how can we build a recognizer (scanner) for it?

Finite Automata and Scanners

- A **finite automation** (FA) can be used to recognize the tokens specified by a regular expression.

- An FA consists of:
  - A finite set of *states*
  - A finite *vocabulary*, denoted Σ
  - A set of *transitions* (or moves) from one state to another, labeled with characters in Σ
  - A special state called the *start* state
  - A subset of the states called the *accepting*, or *final*, states.

- An FA can also be represented graphically using a transition diagram, composed of the components shown in Fig. 3.1.

Finite Automata and Scanners (Cont.)

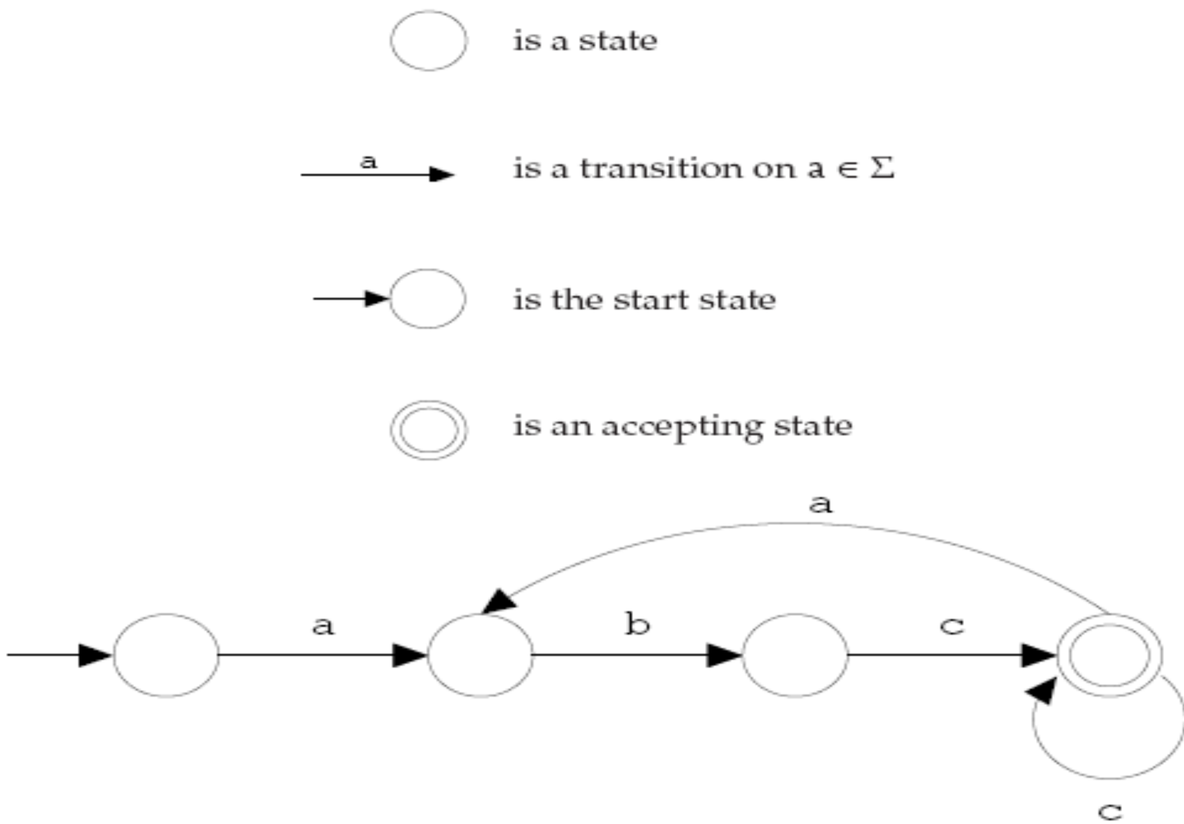


Figure 3.1: Components of a finite automaton drawing and their use to construct an automaton that recognizes $(a\,b\,c^{+})^{+}$.

Finite Automata and Scanners (Cont.)

Deterministic Finite Automata (DFA):

An FA that always allows a unique transition for a given state and character.

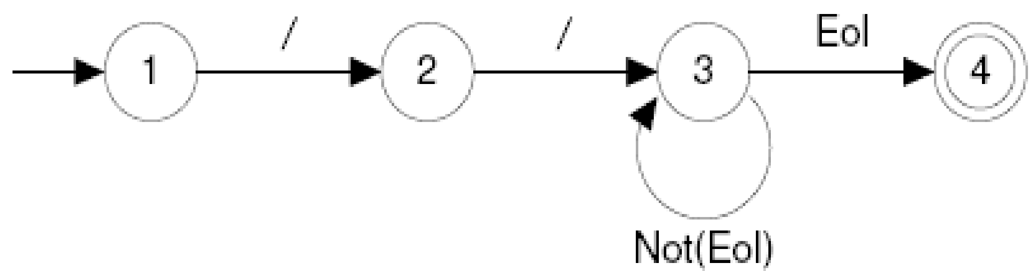
- DFAs are simple to program and are often used to drive a scanner.
- A DFA is conveniently represented in a computer by a **transition table**.
  - For example, the regular expression

    // (Not (eol))* eol

    which defines a Java or C++ **single-line comment**, might be recognized by the DFA shown in Fig. 3.2

Finite Automata and Scanners (Cont.)

(a)



(b)

State	Character				
	/	Eol	a	b	...
1	2				
2	3				
3	3	4	3	3	3
4					

Figure 3.2: DFA for recognizing a single-line comment. (a) transition diagram; (b) corresponding transition table.

Finite Automata and Scanners (Cont.)

- A DFA can be coded in one of two forms:
  - Table-driven
  - Explicit control

- In the *table-driven* form,

  the transition table that defines a DFA's **actions** is **explicitly represented** in a runtime table

  that is "interpreted" by a driver program (figure 3.3).

  Notably, end-of-file is represented by "eof".

Finite Automata and Scanners (Cont.)

```
/*   Assume CurrentChar contains the first character to be scanned   */
State ← StartState
while true do
    NextState ← T[State, CurrentChar]
    if NextState = error
    then break
    State ← NextState
    CurrentChar ← READ( )
if State ∈ AcceptingStates
then  /* Return or process the valid token */
else  /* Signal a lexical error */
```

Figure 3.3: Scanner driver interpreting a transition table.

Finite Automata and Scanners (Cont.)

- In the *explicit control* form,

  the transition table that defines

  a DFA's **actions** appears

  implicitly as the control logic

  of the program as shown in figure 3.4.

Finite Automata and Scanners (Cont.)

```
/* Assume CurrentChar contains the first character to be scanned */
if CurrentChar = '/'
then
    CurrentChar ← READ( )
    if CurrentChar = '/'
    then
        repeat
            CurrentChar ← READ( )
        until CurrentChar ∈ { Eol, Eof }
    else /* Signal a lexical error */
else /* Signal a lexical error */
if CurrentChar = Eol
then /* Finished recognizing a comment */
else /* Signal a lexical error */
```

Figure 3.4: Explicit control scanner.

Finite Automata and Scanners (Cont.)

- An FA that analyzes or transforms its input beyond simply accepting tokens is called **transducer**.
- The FAs shown in Fig. 3.5 recognize a particular kind of **constant and identifier**.
- A transducer that recognizes constants might be responsible for developing the appropriate bit pattern to represent the constant.
- A transducer that processes identifiers may only have to retain the name of the identifier.

Finite Automata and Scanners (Cont.)

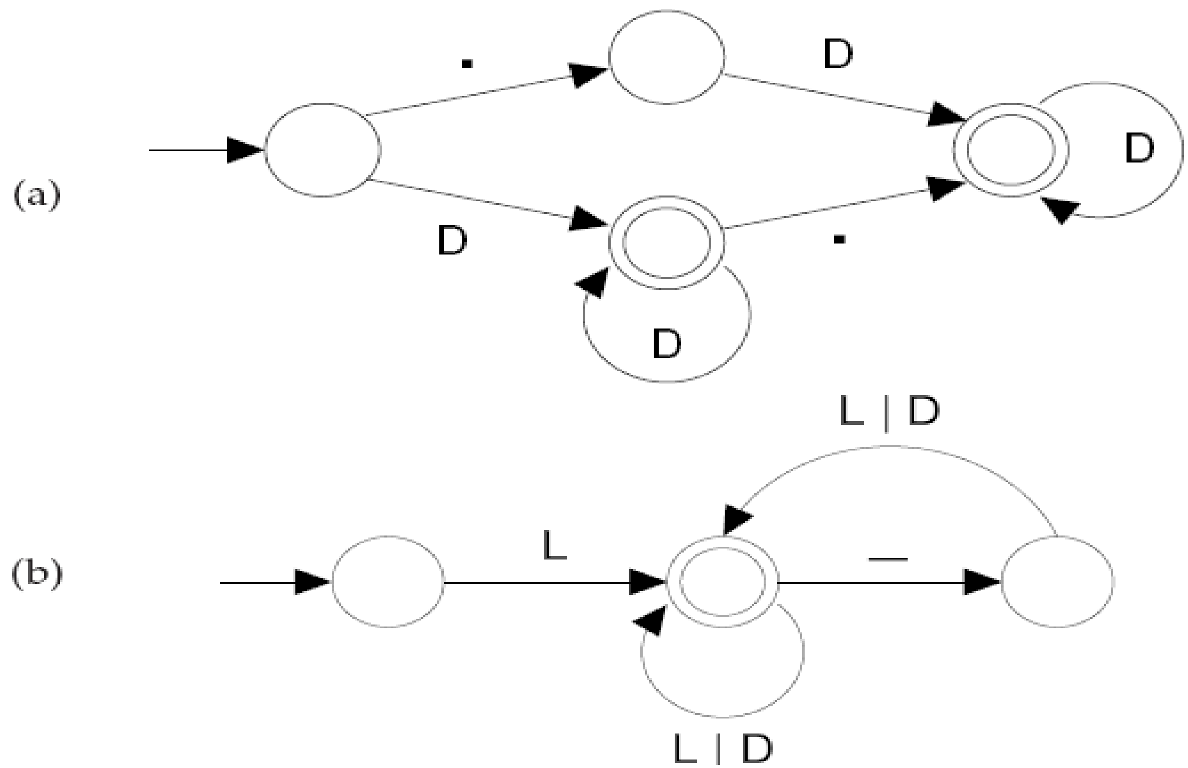


Figure 3.5: DFAs: (a) floating-point constant; (b) identifier with embedded underscore.

Regular Expressions and Finite Automata

- Regular expressions are equivalent to FAs.
- The main job of scanner is to transform

  a regular expression into

  an equivalent FA.

- First, transforming the regular expression into a

  nondeterministic finite automaton (NFA).

Regular Expressions and Finite Automata (Cont.)

- An NFA is a generalization of a DFA that allows

    1) multiple transitions from a state that have the same label

  as well as

    2) transitions labeled with λ

  as shown in Figs. 3.17 and 3.18, respectively.

Regular Expressions and Finite Automata (Cont.)

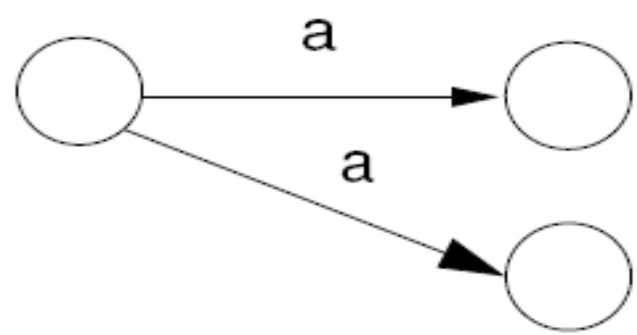


Figure 3.17: An NFA with two a transitions.

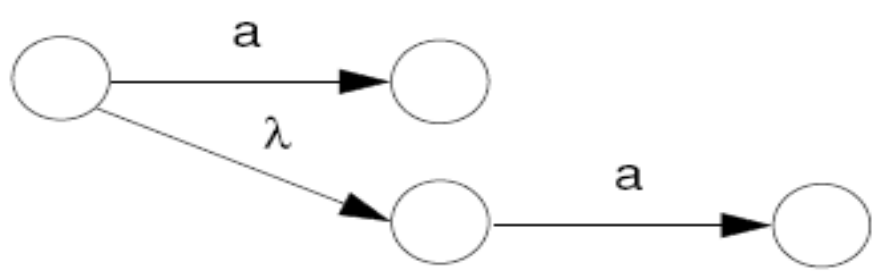
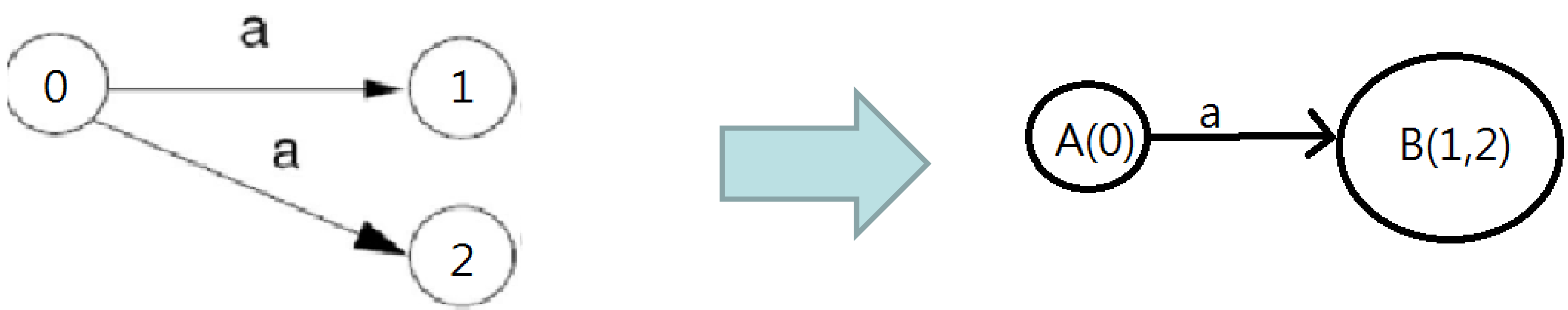
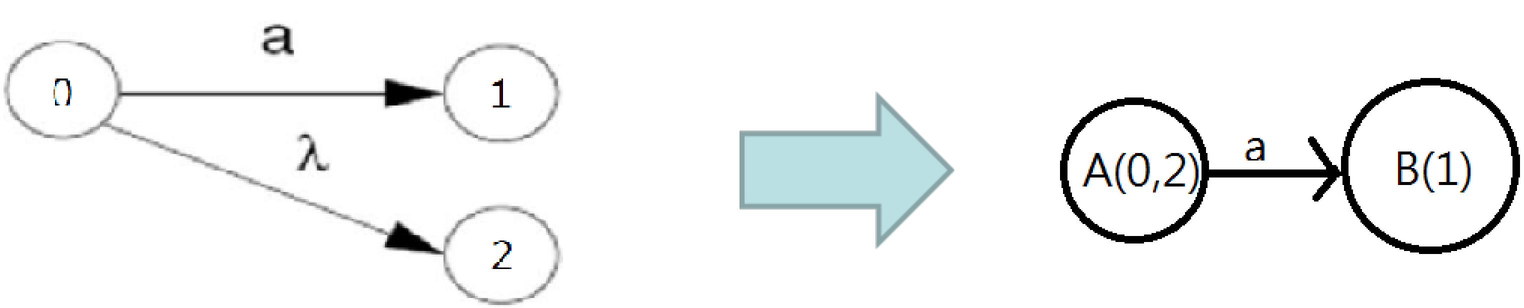


Figure 3.18: An NFA with a λ transition.

NFA → DFA





Transforming Regular Expression to NFA

A regular expression is built of:

the *atomic* regular expressions:

a (a character in Σ) and λ (see Fig. 3.19)

using the three operations:

AB, A|B, and A* (see Figs. 3.20, 3.21, 3.22)

Transforming Regular Expression to NFA (Cont.)

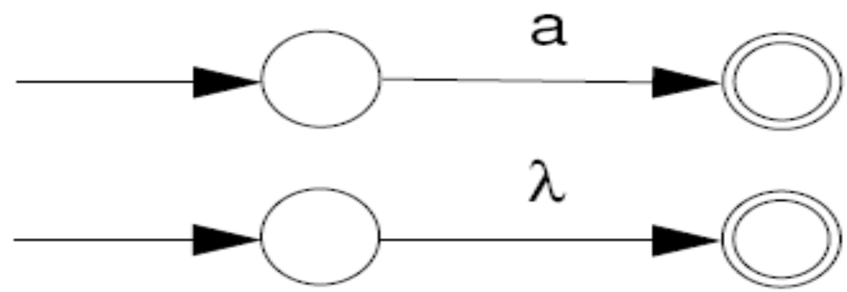


Figure 3.19: NFAs for a and λ.

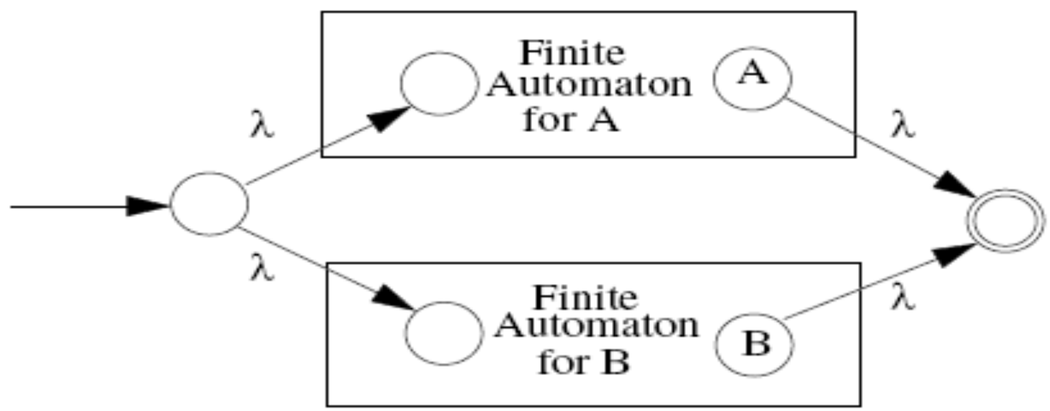


Figure 3.20: An NFA for $A \mid B$.

Transforming Regular Expression to NFA (Cont.)

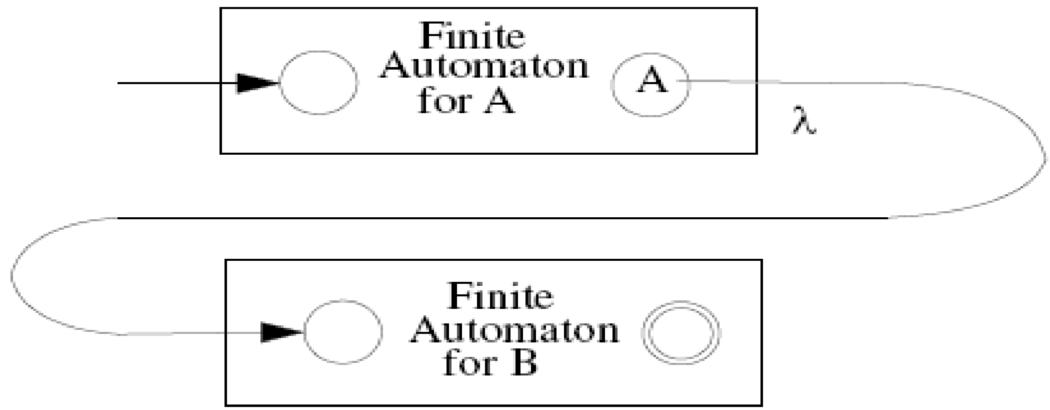


Figure 3.21: An NFA for AB.

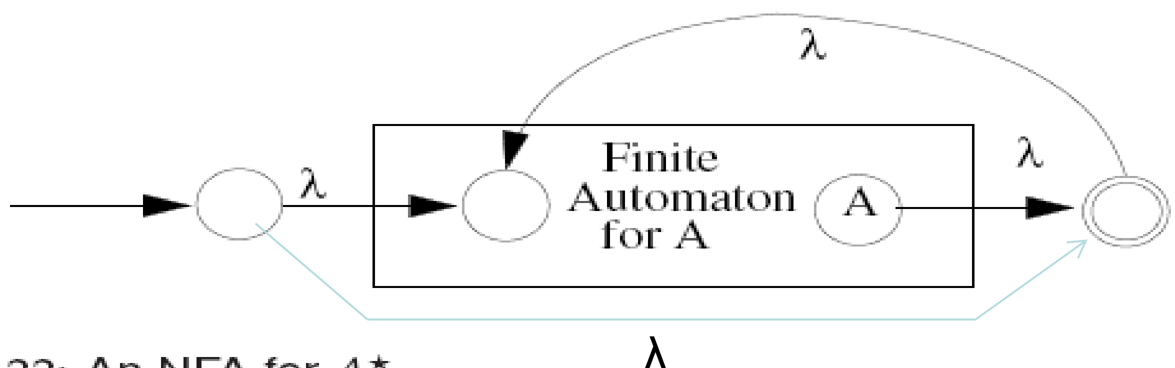


Figure 3.22: An NFA for $A^\star$.

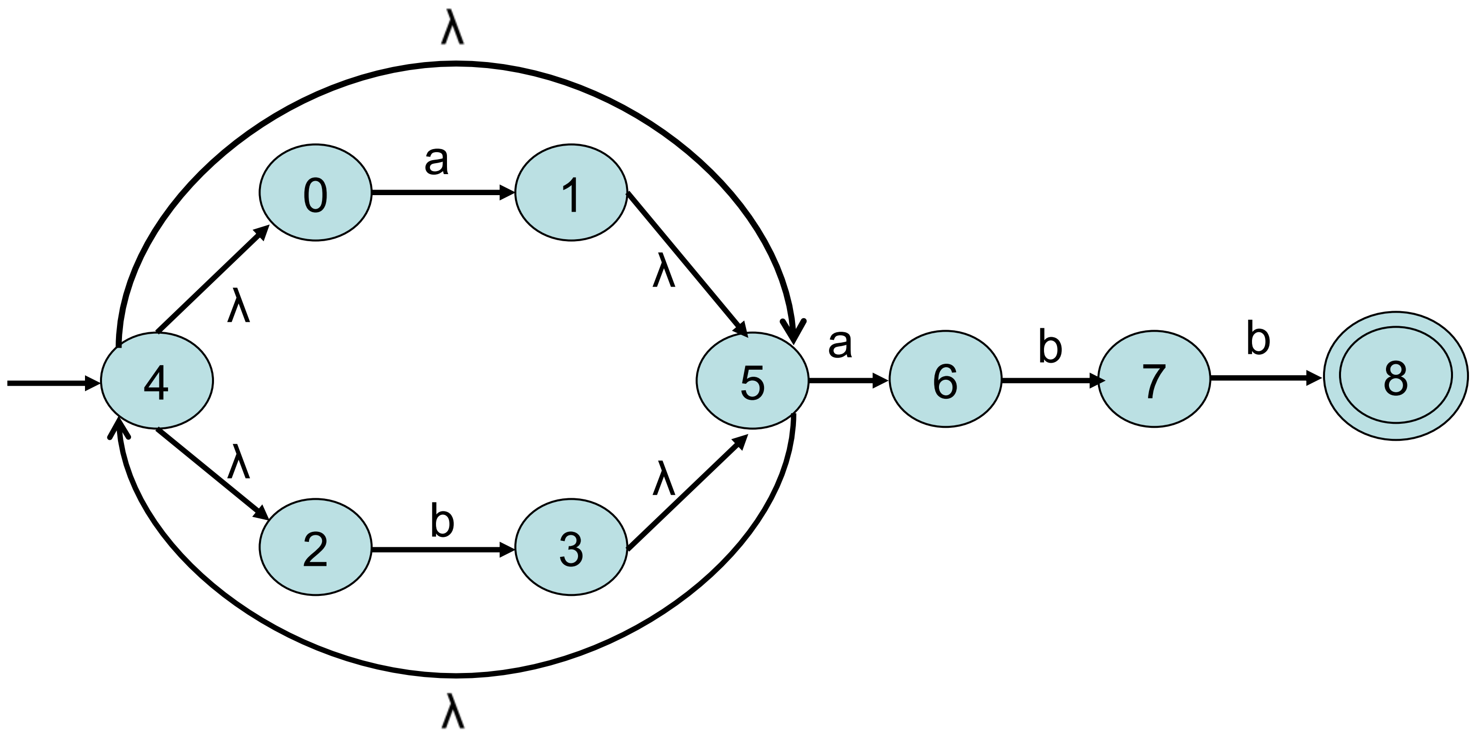
Transforming Regular Expression to NFA (Cont.)

For regular expression **(a|b)*abb**

First, we create the NFA for a, b, a|b, (a|b)*

Then, we create NFA for “abb”

See the animation next.


λ
a
0
1
λ
λ
4
5
a
6
b
7
b
8
λ
λ
b
2
3
λ

Transforming NFA to DFA

- The transformation from an NFA *N* to an equivalent DFA *D* works by the **subset construction** algorithm shown in Fig. 3.23.

- We construct each state of *D* with a *subset* of states of *N*. D will be in **the state {x,y,z}** after reading a given input character, if and only if N could be in ***any*** **of the states x,y,or z**.

Transforming NFA to DFA

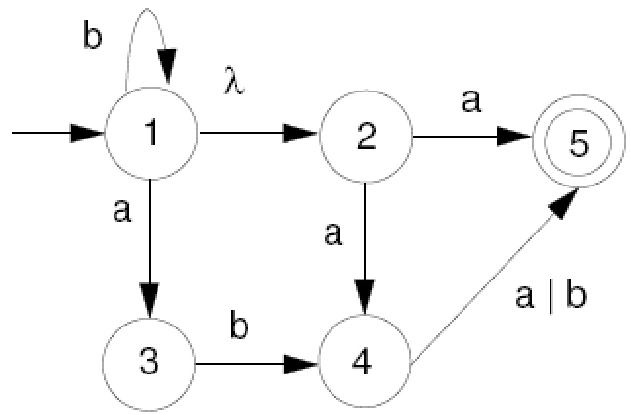


Figure 3.24: An NFA showing how subset construction operates.

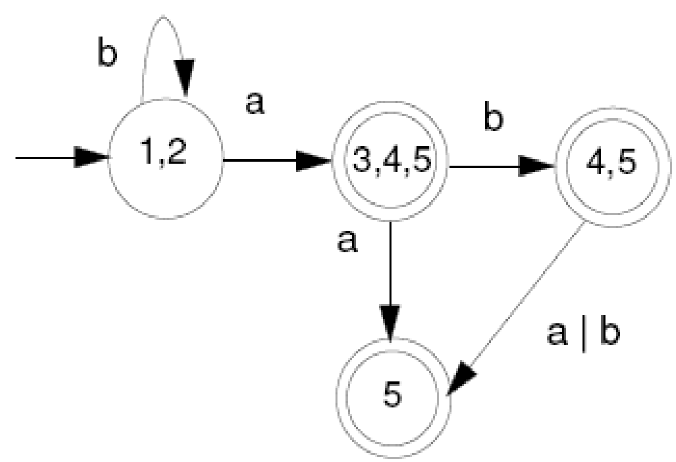


Figure 3.25: DFA created for NFA of Figure 3.24.

Creating the DFA (Cont.)

- Assume an NFA N shown in fig 3.24.
  - Start with state 1, the start state of N, and add its λ closure: state 2. Hence, D's start state is {1,2}.

  - Under a, {1,2}'s successor is {3, 4, 5}.
  - Under b, {1,2}'s successor is itself.

  - Under a and b, {3,4,5}'s successors are:

    {5} and {4,5}, respectively.
  - Under b, {4,5}'s successor is {5}.

  - Accepting states of D are those that contain N's accepting state 5. They are:

    {3,4,5} {4,5} and {5}   The resulting DFA is shown in fig 3.25.

Notion

N: NFA (non-deterministic finite automata)

D: DFA (deterministic finite automata)

$s \xrightarrow{c} t$: In N under char c, state s transits to t.

$S \xrightarrow{c} T$:In D under char c,state S transits to T.

S is a subset of {s | s in N}

```
function makeDeterministic(N) returns DFA
    D.StartState ← recordState({N.StartState})                         ①
    foreach S ∈ WorkList do                                             ②
        WorkList ← WorkList − {S}
        foreach c ∈ Σ do D.T(S, c) ← recordState(T ← ⋃_{s∈S} t(s, c))   ③
    D.AcceptStates ← {S ∈ D.States | S∩N.AcceptStates ≠ ∅}              ④
end
function close(S, T) return Set
    ans ← S
    repeat
        changed ← false
        foreach s ∈ ans do                                              ⑤
            foreach t ∈ T(s, λ) do                                      ⑥
                if t ∉ ans
                then
                    ans ← ans ∪ {t}                                     ⑦
                    changed ← true
    until not changed
    return (ans)
end
function recordState(S) return Set
    S ← close(S, T)                                                     ⑧
    If S ∉ D.States                                                     ⑨
    then
        D.States ← D.States ∪ {S}
        WorkList ← WorkList ∪ {S}
    return (S)
end
```

Revised Figure 3.23 Construction of a DFA *D* from an NFA *N*

Creating the DFA (Cont.)

We trace the subset algorithm to construct the start state of DFA:

- Start with state 1, the start state of N, and call RecordState(state 1) to find its λ-closure (Marker 1).
- RecordState() calls Close(state1, T). T includes states 2 and 3 (Marker 8).
- In Close(), set ans to state 1 (S).

  And then for state 1 in ans (Marker 5) find each t in T(s, λ) (Marker 6) and add t to ans, which is state 2 (Marker 7). After that, return the set, states {1,2}, to RecordState().
- Then, RecordState() will determine whether the set is in D.States. It is not, so it will be stored into D.States and WorkList (Marker 9).

- Now, we have constructed DFA ‘s **start state as states {1,2}.**

Creating the DFA (Cont.)

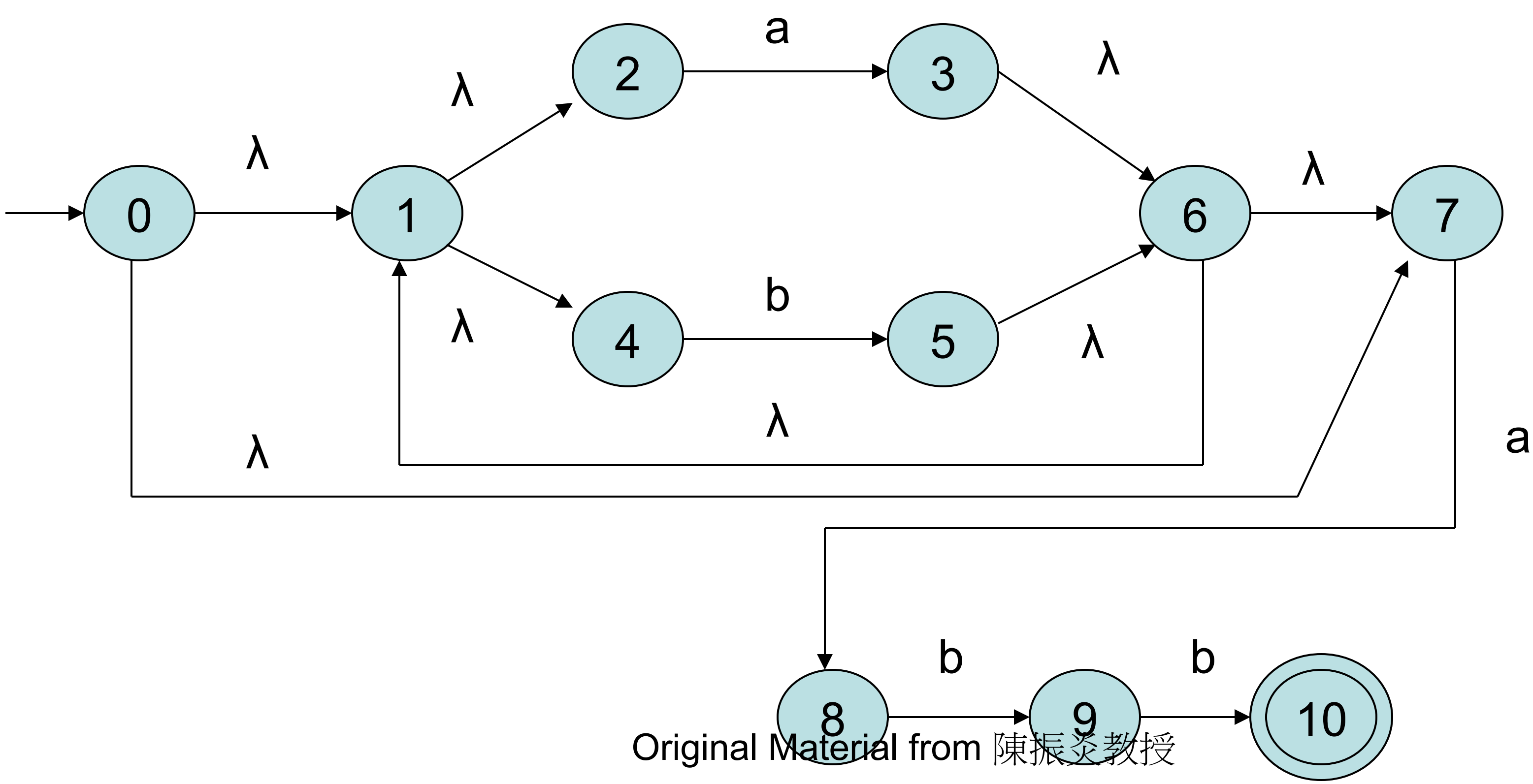
Next, we construct the successors of the start state S = {1, 2} of DFA:

for each S in WorkList (S = **{1, 2}**) do
  under char **"a"**
    set S's successor D.T(S, c) (S is **{1, 2}** and c is ***a***) to:
    state 1 transits to 3,
    state 2 transits to 4,
    state 2 transits to 5, we got T= **{3,4,5}**
   recordStates (**{3,4,5})**
    add {3,4,5} to D.States and workList

  under char **"b"**
    set S's successor D.T(S, c) (S is **{1, 2}** and c is ***b***) to:
    state 1 transits to 1, we got T=**{1}**
   recordStates (**{1})** calls close() we got T=**{1,2}**
    {1,2} is already in D.States, so do not add it to D.States and workList

Example

Given the NFA below, find its DFA.



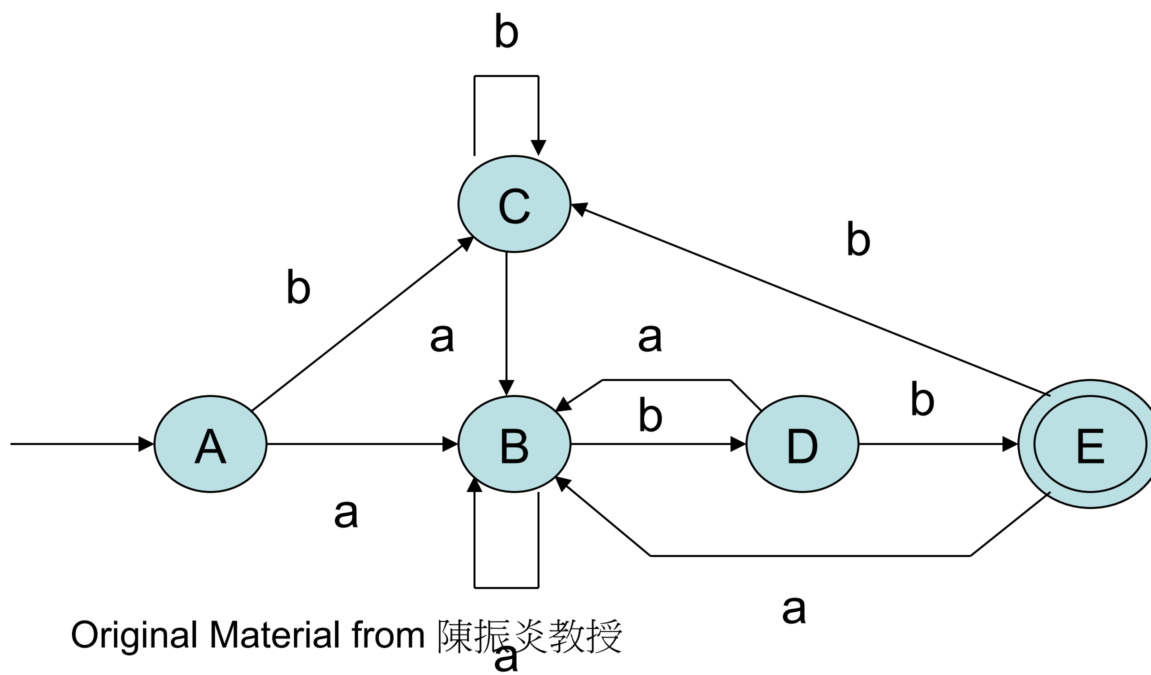
Example (Cont.)

The resulting DFA is:

A {0, 1, 2, 4, 7} B {1, 2, 3, 4, 6, 7, 8}

C {1, 2, 4, 5, 6, 7} D {1, 2, 4, 5, 6, 7, 9}

E {1, 2, 4, 5, 6, 7, 10}



IMPORTANT NOTES

- CFG is more powerful than regular expression
  - Ex. aaabbb which has the same number of a and b can not be expressed by regular expression but can be expressed by CFG
  - S -> a S b
  - S -> λ
- That is, DFA/NFA are not capable of remembering the occurrences of symbols

Important Notes

- Regular expression actually define a language **L**, where tokens are characters
- If an regular expression contains non-terminals, it can be expanded base on rewriting rules like a CFG (context free grammar)
- R* is equivalent to :
- Rs -> RRs
- Rs-> λ

Homework

- 3. Write the regular expressions for:

  (a) A floatdcl can be represented as either f or float, allowing a more Java-like syntax for declarations.

  (b) An intdcl can be represented as either i or int

  (c) A num may be entered in exponential (scientific) form. That is, an **ac** num may be suffixed with an optionally signed exponent (1.0e10, 123e-22 or 0.31415926535e1)

HW Solution

(a) (b)

Terminal	RegularExpression
floatdcl	“f” | (“f” “l” “o” “a” “t”)
intdcl	“i” | (“i” “n” “t”)

(c) inum $[0\text{-}9]^{+}$ e -? $[0\text{-}9]^{+}$

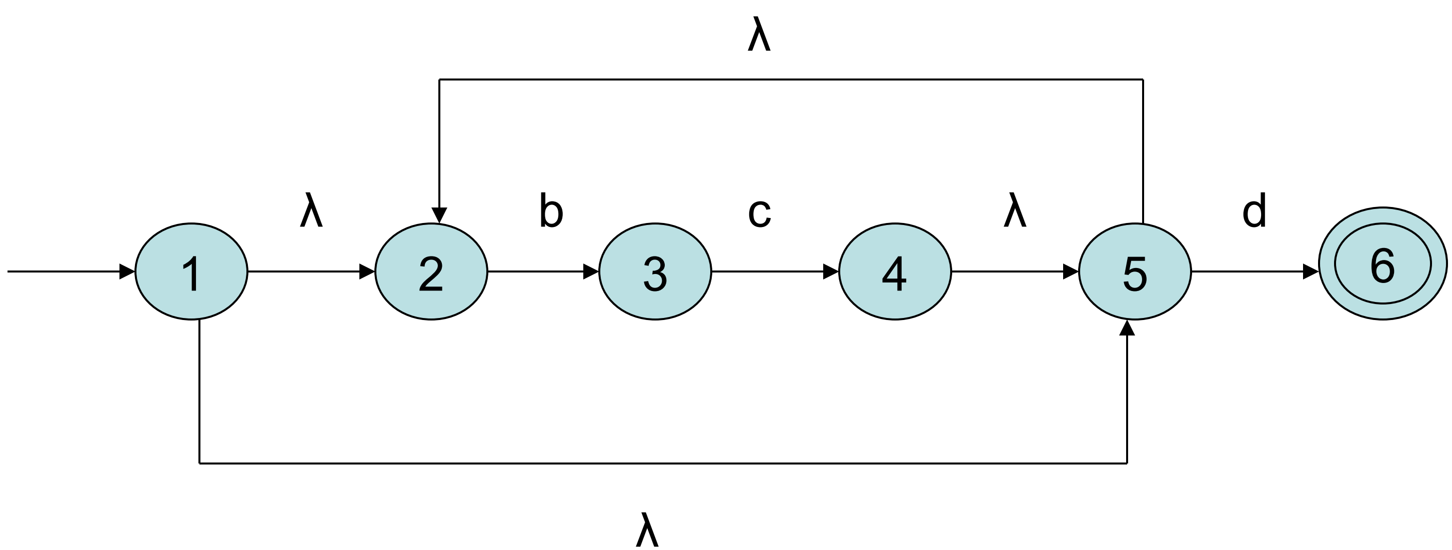
fnum $[0\text{-}9]$. $[0\text{-}9]^{+}$ e -?$[0\text{-}9]^{+}$

Homework

5(d) Write NFA, and then DFA that recognizes the tokens defined by the following regular expression:

(bc)*d

Homework Solution

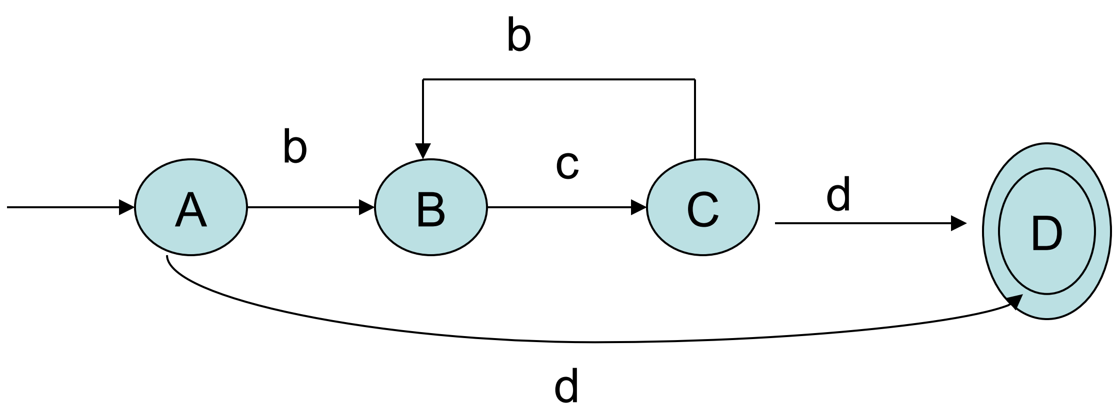


Homework Solution

From NFA to DFA

A {1,2,5}　　　B {3}

C {2,4,5}　　　D {6}



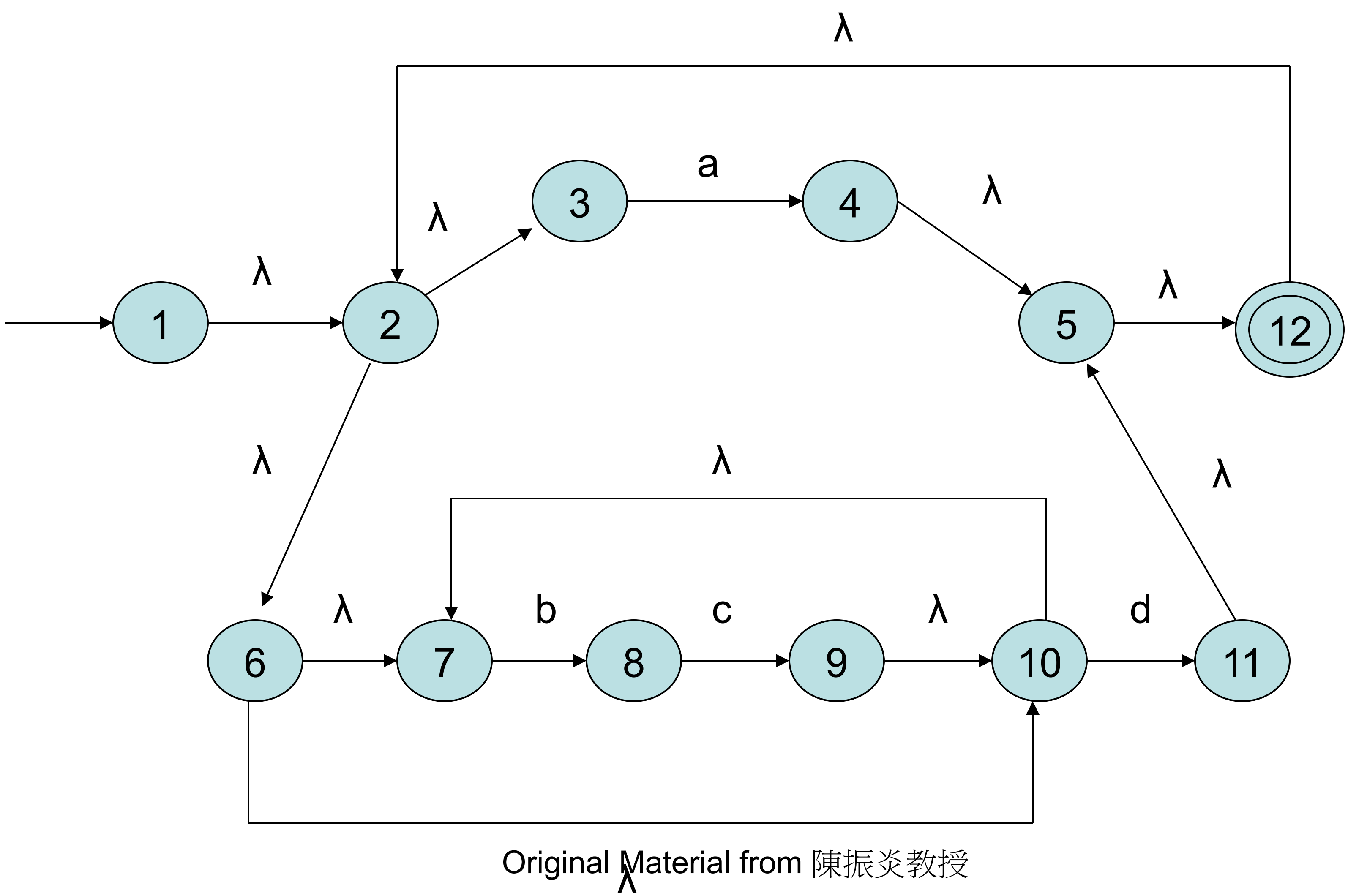
Home work

5(a) Write NFA and DFA that recognizes the tokens defined by the following regular expression:

$$(a|(bc)^*d)+$$

Homework Solution

From regular expression to NFA:



Homework Solution

From NFA to DFA:

A {1,2,3,6,7,10} B{2,3,4,5,6,7,10,12}

C {8} D{2,3,5,6,7,10,11,12} E{7,9,10}