

## Chapter 3 Scanner

Formal (and precise) specification  $\Rightarrow$  scanner program

Wuu Yang

National Chiao-Tung University, Taiwan, R.O.C.

first draft: March 1, 2010

current version: January 25, 2011

©January 25, 2011 by Wuu Yang. All rights reserved.

## Chapter outline: Scanning—Theory and practice

1. Overview of a scanner
2. Regular expressions
3. Examples
4. Finite automata and scanners
5. The lex scanner generator
6. Other scanner generators
7. Practical considerations of building scanners
8. Regular expressions and finite automata

Most programs require some kind of scanning, such as processing network packets, displaying web pages, interpreting audio and video media.

### §3.1. Overview

A scanner partitions a stream of characters into a stream of tokens. Though token structures are quite “simple,” their precise structures could be very subtle. Formal notations are necessary for implementors. Ex.

1. Can a string split across a line? Or can and how could a new-line character appear in a string?
2. Is a null string allowed?
3. Can and how could a double quote appear in a string?
4. Is a zero-length array allowed?
5. Is `.1` or `10.` ok?
6. the `1..10` problem: (a) as three tokens or (b) as two

floating-point numbers

Only formal notations are subject to rigorous checking for consistency, and hence help language designers and implementors.

Scanner generators (tables or programs)

What formal notation to use?

## §3.2. Regular Expressions

An example:

$$0(0|1|2|3|4|5|6|7)^*$$

Tokens are built from characters of a finite vocabulary (denoted as  $V$  or  $\Sigma$ ). Common vocabulary includes a-z, A-Z, 0-9, ;, =, <, >, etc. More examples:

$$(a|b)^*, a|b^*, ab|c, a(b|c), a|b|c|d, ab|cde|f$$

We use regular expressions to define structures of tokens, e.g.,

$$\text{Delimiter} = ( '( ' | ' ) ' | := | '+ ' | - | '* ' )$$

Because some characters are in the vocabulary and are used in regular expressions, they need to be properly *quoted* (meta-characters), e.g., ( ) + \* = | ' are meta characters.

## regular expressions

1.  $\emptyset$  (denoting the empty set)
2.  $\lambda$  (denoting the set of the null string)
3.  $c$  (any character from the vocabulary)
4. If  $A$  and  $B$  are regular expressions, so are  $(A|B)$  (alternation),  $(A \cdot B)$  (concatenation), and  $A^*$  (Kleene closure).

Examples of regular expressions:  $((((be)g)i)n)$ ,  $((en)d)$ ,  $(:=)$ ,  $0(((0|1)|2)|3)^*$ .

A regular expression denotes a set of strings. A string is a concatenation of characters from the vocabulary. We use  $\lambda$  or  $\epsilon$  to denote the empty string.

In order to omit parentheses, we assume that  $*$  has the highest precedence and  $|$  has the lowest precedence. The operators  $|$  and  $\cdot$  are associative. We always omit  $\cdot$ . Thus,  $((a \cdot (b^*)) \cdot c) | d$  is

abbreviated as  $ab^*c|d$ .

The above examples may be written as `begin, end, :=,`  
`0(0|1|2|3)*`.

This abbreviation is very similar to ordinary arithmetic notation.  
For instance, we may write  $((1 + ((2 * 3)/4)) - 5)$  as  $1 + 2 * 3/4 - 5$ .

Sometimes we may give a “name” to a regular expression and use that name in other regular expressions. For example,

$$\text{OCT} = 0(0|1|2|3|4|5|6|7)^*$$

Some notational conveniences.

$$P^+ == PP^*$$

$$\text{Not}(A) == V - A, \text{ where } A \text{ is a set of symbols.}$$

$$\text{Not}(S) == V^* - S, \text{ where } S \text{ is a set of strings.}$$

$$A^k == AA \dots A \text{ (} k \text{ times)}$$

### §3.3 Example.

Let  $D = (0|1|2|3|4| \dots |9)$  and  $L = (A|B| \dots |Z)$ .

`comment = --not(EOL)*EOL`

`decimal = D+.D+`

`identifier = L(L|D)*(_(L|D)+)*` or `L(L|D)*(_(L|D)+)*(L|D)+`

What if an identifier can end with `_`?

Comments are delimited by `## ... ##`, but we may use a single `#` inside comments.

`comments = ##((#|\lambda)not(#))* ##`

– not quite right; what if `#####`?

Is the following expression correct?

`comments = ##((#|\lambda)not(#))* (#|\lambda) ##`

– not quite right This does not allow `####`.

What about nested comments?

$$\{ \dots \{ \dots \} \dots \}$$

regular expressions vs. context-free grammars

Try exercise 5 on page 87. WRONG — (\* not ['\*')']\*)

All finite sets of strings are regular.

## Ambiguities

Ex. In Pascal, “1 + 2” could be scanned in two ways.

- Given  $\text{INT} = \text{D}^+$ , how to interpret 123456?

### **Longest match rule.**

Find the longest string that satisfies a token definition.

- Given the two token definitions:

$\text{OCT} = 0(0|1|2|3|4|5|6|7)^*$ ,

$\text{DEC} = \text{D}^+$

how to interpret 0?

### **First match rule.**

Find the first rule that the input matches.

Sometimes, the longest match rule is not good.

Ex. In Modula-2, “10.” and “.20” are valid real numbers.

Consider the case 10..20.

Ex. In C,  $x=y+++++z;$ .

## §3.4 Finite Automata (or Finite State Machines)

regular expressions  $\Leftrightarrow$  fa

An example of fa in Figure 3.1:

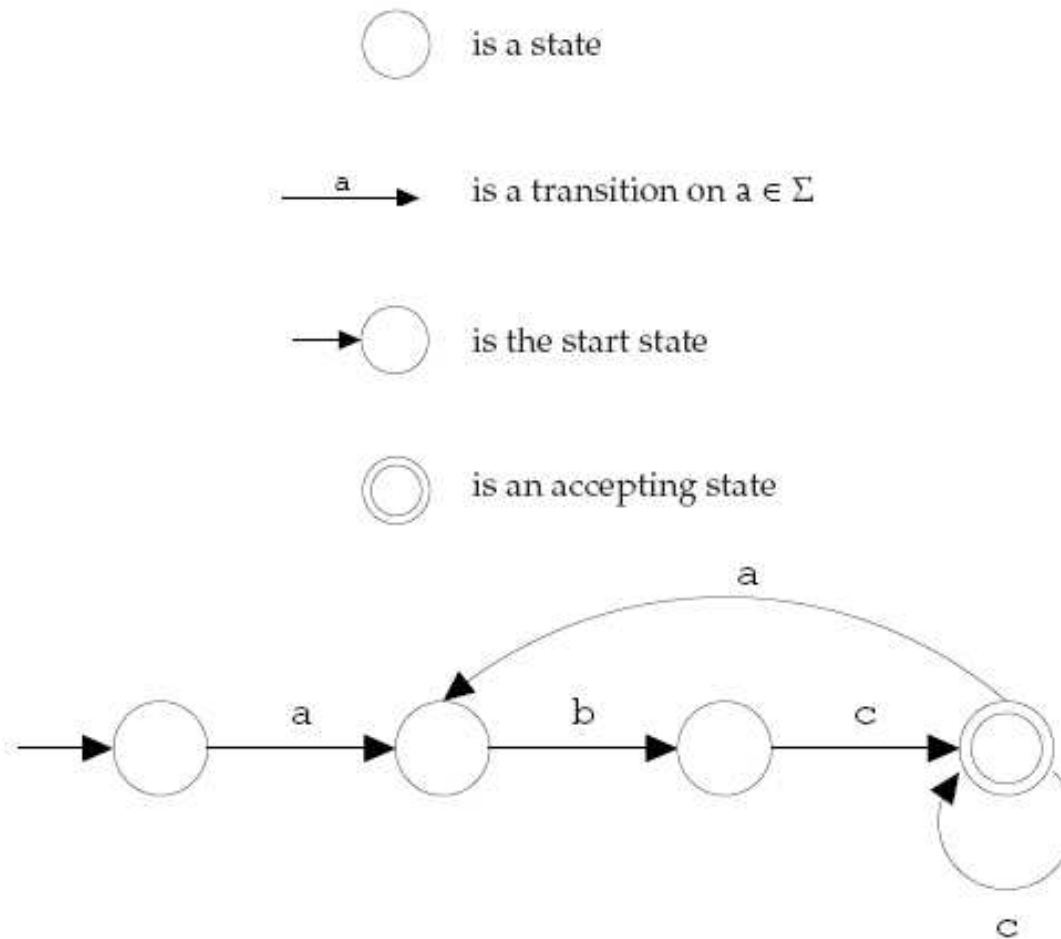
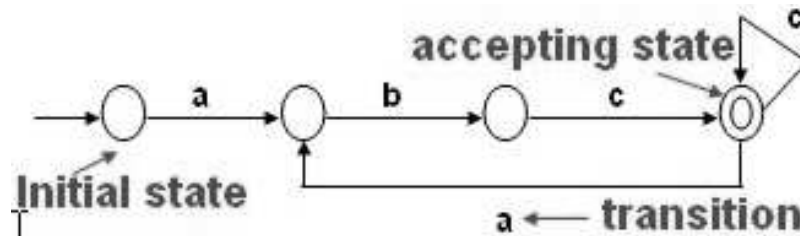


Figure 3.1: Components of a finite automaton drawing and their use to construct an automaton that recognizes  $(a b c^+)^+$ .

An fa consists of

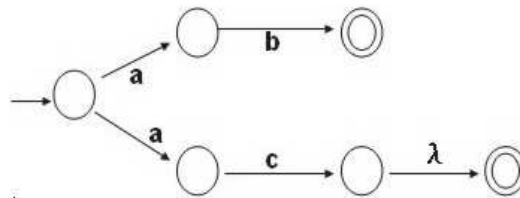
- a finite, non-empty set of *states*
- an *initial state*
- some *accepting states*
- *transitions* between states, labelled with characters from the vocabulary.

Ex. The following machine accepts **abccabc**, but it rejects **abcab**. This machine accepts all and only strings defined by the regular expression  $(abc^+)^+$ .



There are two kinds of fa:

1. *deterministic* (dfa): Next transition is unique.
2. *nondeterministic* (nfa): Otherwise. Include  $\lambda$  transitions and similar transitions.



In addition to a *transition diagram*, an fa may be represented by a

*transition table*, ex.

	a	b	c
1	2		
2		3	
3			4
4	2		4

The transition table is usually

sparse. We will discuss how to compress the table.

In the transition table, we implicitly use a *trap state*.

Alternatively, a dfa can be represented by a program.

Here is a scanner driver, which uses only 1-character look-ahead.

```
1.  Algorithm:
2.  state := initial state; curchar := getchar();
3.  while (TRUE) {
4.      nextState := T[state][curchar];
5.      if (nextState == ERROR) break;
6.      state := nextState;
7.      if (curchar == EOF) break;
8.      curchar := getchar();
9.  }
10. if (IsAcceptingState(state))
11.     return the token corresponding to state;
12. else lexicalError(curchar);
```

*Example.*

`comments= --not(eol)*eol`

A dfa for single-line comments is shown in Figure 3.2. Its scanner is shown in Figure 3.4.

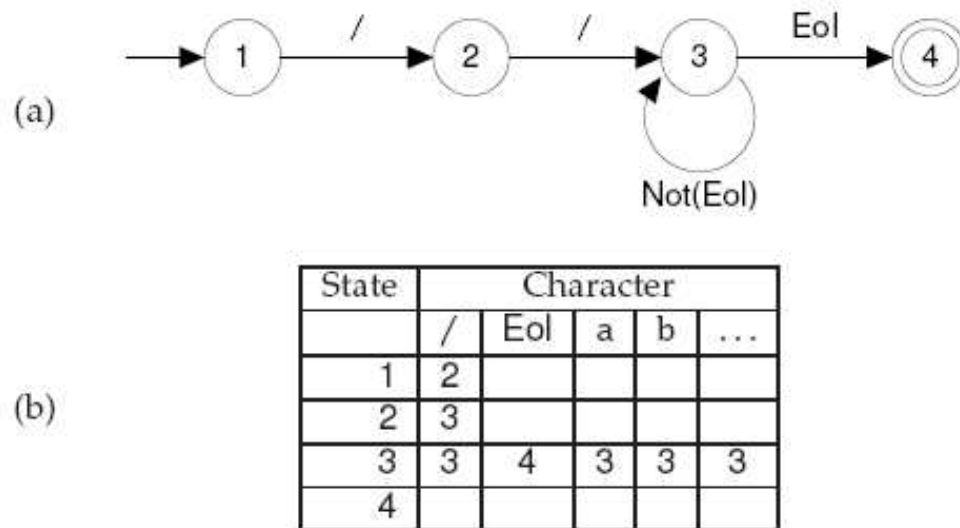


Figure 3.2: DFA for recognizing a single-line comment. (a) transition diagram; (b) corresponding transition table.

---

```
if CurrentChar == '/' then
  CurrentChar := Read()
  if CurrentChar == '/' then
    repeat
      CurrentChar := Read()
    until CurrentChar belongs to { Eol, Eof }
  else Error()
else Error()
if CurrentChar == Eol then // found a comment
else Error()
```

Figure 3.4 A scanner for single-line comments.

```

/* Assume CurrentChar contains the first character to be scanned */
if CurrentChar = '/'
then
    CurrentChar ← getNextChar()
    if CurrentChar = '/'
    then
        repeat
            CurrentChar ← getNextChar()
        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 */

```

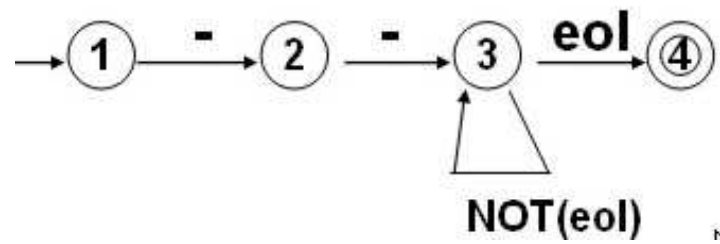
Figure 3.4: Explicit control scanner.

---

Example. `comments= --not(eol)*eol` The transition table is as

follow:

	-	eol	a	b	...
1	2				
2	3				
3	3	4	3	3	...
4					



Example.

$\text{RealLit} = (D^+(\lambda \mid \cdot)) \mid (D^* \cdot D^+)$

$\text{ID} = L (L \mid D)^* (\_ (L \mid D)^+)^*$

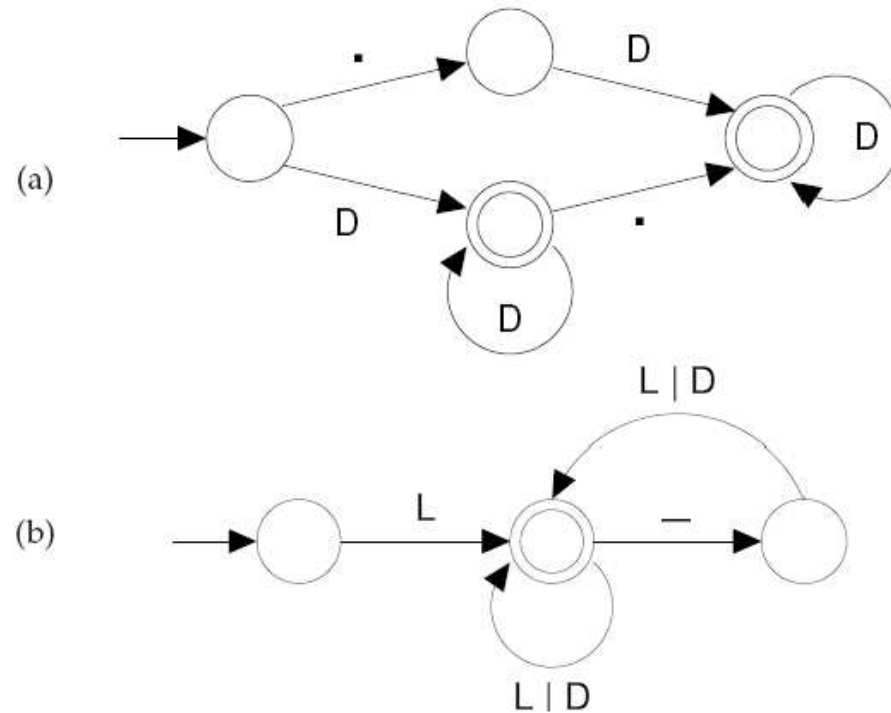
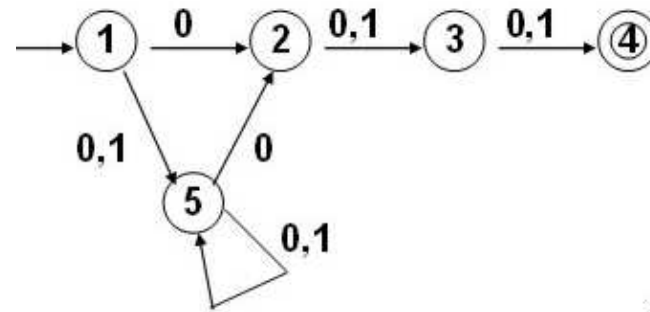
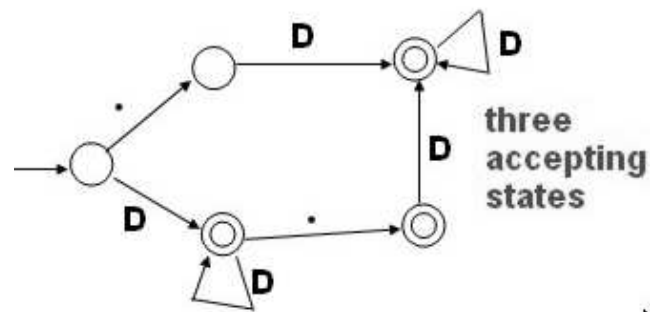


Figure 3.5: DFAs: (a) floating-point constant; (b) identifier with embedded underscore.

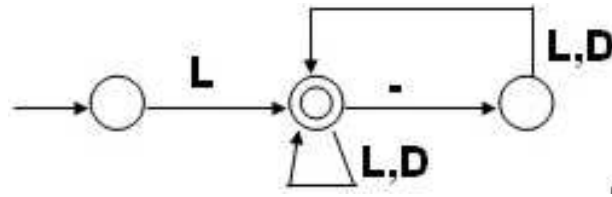
Example.  $(0|1)^*0(0|1)(0|1)$



Example.  $\text{RealLit} = (D^+(\lambda|.)) | (D^*.D^+) = D^+ | D^+. | D^*.D^+$



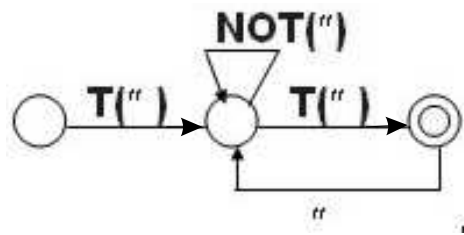
Example.  $ID = L(L|D)^*(\_ (L|D)^+)^*$  (What if an ID may end with  $\_?$ )



We may perform some actions during state transition, e.g., toss away the input characters.

Ex.  $String = "(not(" | ")*)"$

T means *toss-away*. Try this fa with input `""Hi""`. Finally the buffer will contain `"Hi"`.



Some actions may be attached to a scanner during state transitions. The result is called a *transducer*. A transducer translates an input string to an output string.

### §3.5 Lex: a scanner generator

All scanners share much code. It is a waste to re-implement a complete scanner. So we use a generator that *generates* new scanners by re-using existing code. A user only has to supply a specification of the token structures. The specification is regular expressions, written in the format dictated by the scanner generators.

Lex/flex/JFlex are famous scanner generators.

Figure 3.7 is a partial lex input for ac. Figure 3.8 is the format for lex input.

```
%%  
f      { return(FLOATDCL); }  
i      { return(INTDCL); }  
p      { return(PRINT); }  
%%
```

Figure 3.7: A Lex definition for ac's reserved words.

---

```
declarations  
%%  
regular expression rules  
%%  
subroutine definitions
```

Figure 3.8: The structure of Lex definition files.

---

Figure 3.9 is the character class definition.

Character Class	Set of Characters Denoted
[abc]	Three characters: a, b, and c
[cba]	Three characters: a, b, and c
[a-c]	Three characters: a, b, and c
[aabbcc]	Three characters: a, b, and c
[^abc]	All characters except a, b, and c
[\^-\]]	Three characters: ^, -, and ]
[^]	All characters
"[abc]"	Not a character class. This is an example of one five-character <i>string</i> : [abc].

Figure 3.9: Lex character class definitions.

---

Figure 3.10 is the definition of ac's identifiers.

```
%%  
[a-eghj-oq-z]      { return(ID); }  
%%
```

Figure 3.10: A Lex definition for ac's identifiers.

---

Figure 3.11 and Figure 3.12 are lex input for ac.

```
%%
(" ")+          { /* delete blanks */}
f               { return(FLOATDCL); }
i               { return(INTDCL); }
p               { return(PRINT); }
[a-eghj-oq-z]  { return(ID); }
([0-9]+)|([0-9]+ "." [0-9]+) { return(NUM); }
"="            { return(ASSIGN); }
"+"           { return(PLUS); }
"_"           { return(MINUS); }
%%
```

Figure 3.11: A Lex definition for ac's tokens.

---

```

%%
Blank          " "
Digits        [0-9]+
Non_f_i_p     [a-eghj-oq-z]
%%
{Blank}+      { /* delete blanks */}
f             { return(FLOATDCL); }
i             { return(INTDCL); }
p             { return(PRINT); }
{Non_f_i_p}   { return(ID); }
{Digits}|({Digits}"."{Digits}) { return(NUM); }
"="          { return(ASSIGN); }
"+"          { return(PLUS); }
"-"          { return(MINUS); }
%%

```

Figure 3.12: An alternative definition for ac's tokens.

## §3.7 Practical Considerations

### §3.7.1 Reserved Words

Usually, all keywords are reserved in order to simplify parsing.

What if `begin` and `end` may be used as variable names, e.g.,

```
. . . begin = . . .
```

In Pascal, we could even write

```
begin
  end; end; begin; begin;
end
if else then if = else;
```

The problem with reserved words is that they are too numerous.

For example, COBOL has several hundred reserved words!

Modula-2 has 40 reserved keywords and 13 standard identifiers.

Pascal has 35 reserved keywords. C has 29 reserved words.

## How to handle reserved words?

1. Use regular expressions – very complicated! E.g.,

$ID = L(L|D)^*$

$NID = NOT(NOT(ID) | begin | end | \dots)$

Use a *tunnel automaton*.

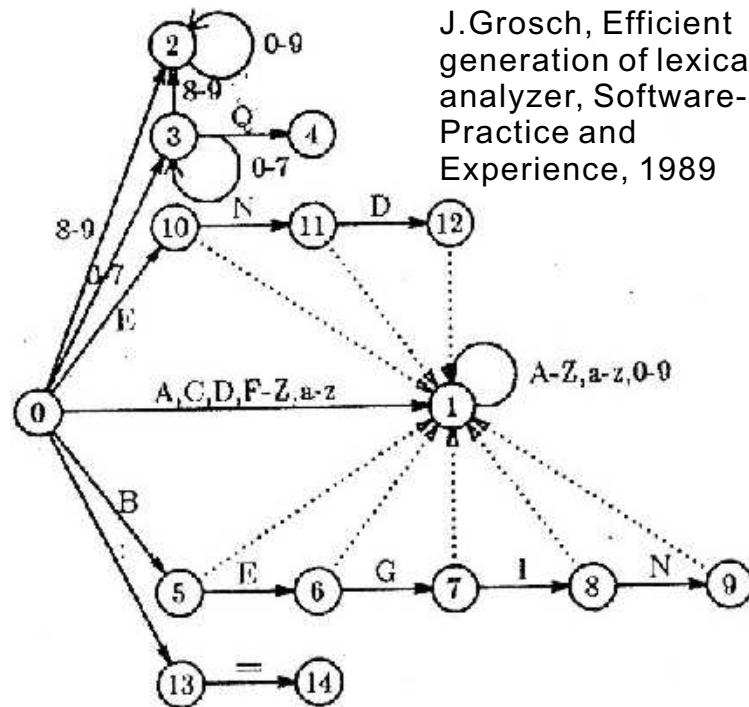


Figure 3. Tunnel automaton for the running example

2. Treat reserved words as ordinary identifiers and use a table-lookup to detect them. The table can be organized as
  - (a) a sorted list (binary search)
  - (b) a hash table (perfect hash is possible because the set of reserved words is fixed)
  - (c) create a distinct r.e. for each reserved word (big fa)
  - (d) trie

LEXAGEN

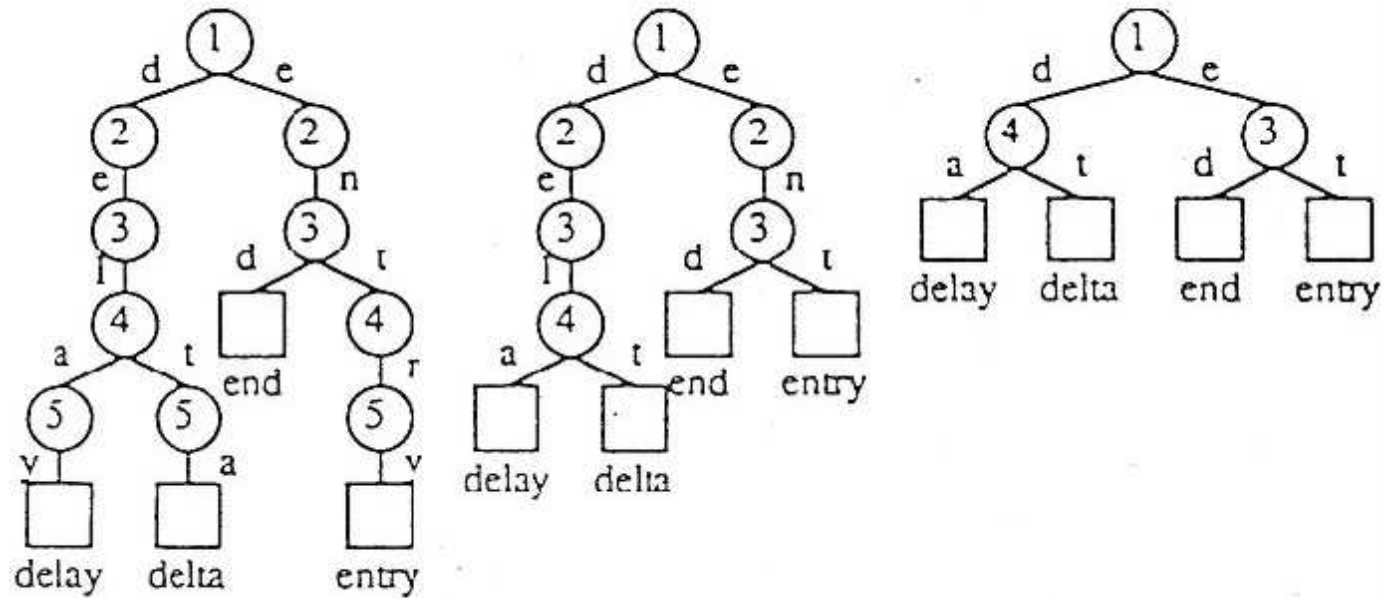
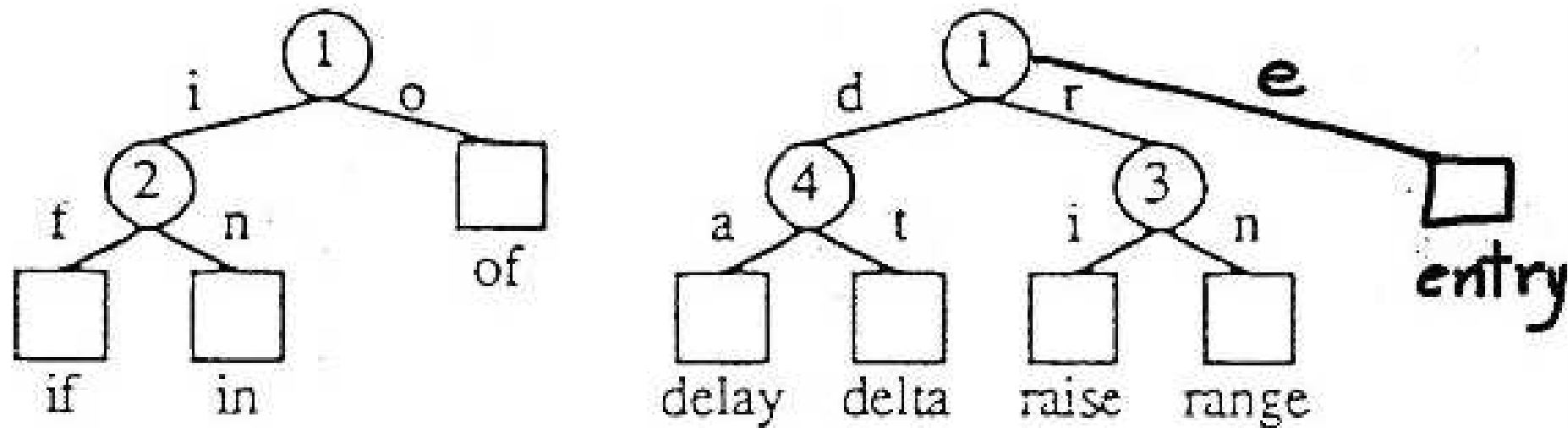


Figure 12. A full trie, a pruned trie and a pruned O-trie

Keyword recognition



*Figure 13. A pruned O-trie forest.*

↓

## Minimum perfect hashing

Given  $n$  keywords  $K = \{K_1, K_2, \dots, K_n\}$ . Find a 1-1, onto function  $f : K \rightarrow \{1, 2, \dots, n\}$ .

Newton's interpolating polynomial Given  $n$  points  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ , let

$$\begin{aligned} P(x) = & [ e_n(x - x_{n-1})(x - x_{n-2}) \dots (x - x_1) + \\ & + e_{n-1}(x - x_{n-2}) \dots (x - x_1) + \dots + \\ & + e_2(x - x_1) + e_1 ] \text{ mod } q \end{aligned}$$

and  $P(x_i) = y_i$ , for all  $i = 1, \dots, n$ , where  $q$  is a prime,  $q > x_i, q > y_i$ .

The solution to the above equations is

$$e_1 = y_1 \text{ mod } q,$$

$$e_2 = [(y_2 - e_1)/(x_2 - x_1)] \text{ mod } q$$

$$e_3 = [(y_3 - (e_1 + e_2(x_3 - x_1)))/((x_3 - x_2)(x_3 - x_1))] \text{ mod } q$$

$$\dots$$

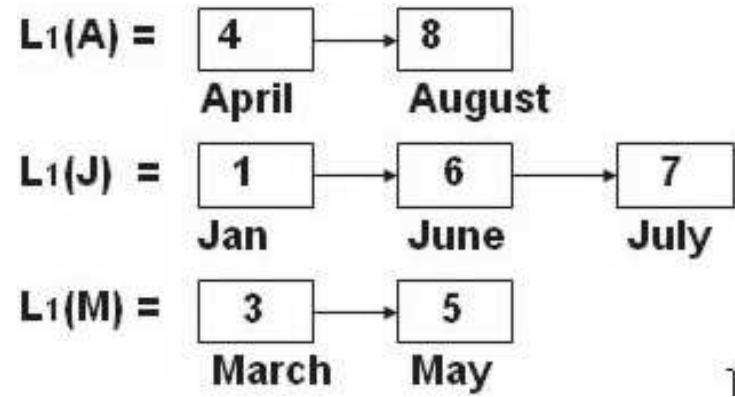
$$e_n = \frac{y_n - \sum_{s=1}^{n-1} e_s \prod_{t=1}^{s-1} (x_n - x_t)}{\prod_{s=1}^{n-1} (x_n - x_s)} \pmod{q}.$$

Ex. Given the 12 pairs (see next page), assume the encoding of A is 1, that of B is 2, etc. According to the 3rd column, we have the following equation:

$$\begin{aligned}
P_1(x) = & 3(x - F)(x - M)(x - A)(x - S)(x - 0)(x - N)(x - D) + \\
& 2(x - M)(x - A)(x - S)(x - 0)(x - N)(x - D) + \\
& 19(x - A)(x - S)(x - 0)(x - N)(x - D) + \\
& 9(x - S)(x - 0)(x - N)(x - D) + \\
& 23(x - 0)(x - N)(x - D) + \\
& 4(x - N)(x - D) + \\
& 4(x - D) + \\
& 0
\end{aligned}$$

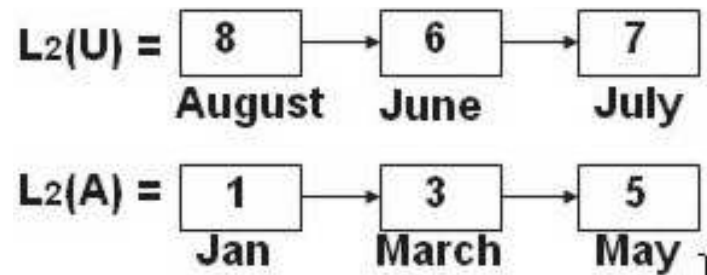
month	choose the 1st character	repetition
(January , 1)	(J,1)	(J,0)
(February, 2)	(F,2)	(F,2)
(March, 3)	(M,3)	(M,0)
(April,4)	(A,4)	(A,0)
(May,5)	(M,5)	
(June,6)	(J,6)	
(July,7)	(J,7)	
(August,8)	(A,8)	
(September,9)	(S,9)	(S,9)
(October,10)	(O,10)	(O,10)
(November,11)	(N,11)	(N,11)
(December,12)	(D,12)	(D,12)

We also have the following three lists:



For the repetitions (i.e.,  $P_1(x) = 0$ ), consider the 2nd characters (see the next table). We can establish the 2nd equation and two lists:

$$P_2(y) = 10(y - U)(y - A) + 8(y - A) + 0$$



month	choose the 2nd character	repetition
(January , 1)	(A,1)	(A,0)
(March, 3)	(A,3)	
(April,4)	(P,4)	(P,4)
(May,5)	(A,5)	
(June,6)	(U,6)	(U,0)
(July,7)	(U,7)	
(August,8)	(U,8)	

Now consider each of the  $L_1$  list and of the  $L_2$  list. We have

$$L_1(M) \cap L_2(A) = (\text{March}, \text{May})$$

$$L_1(J) \cap L_2(U) = (\text{June}, \text{July})$$

We will consider the 3rd character:

month	choose the 3rd character	repetition
(March, 3)	(R,3)	
(May,5)	(Y,5)	
(June,6)	(N,6)	(U,0)
(July,7)	(L,7)	

There is no repetition. We have

$$P_3(z) = 4(z - Y)(z - N)(z - L) + 6(z - N)(z - L) + 14(z - L) + 7$$

The hashing function is

$h(C_1C_2C_3 \dots) =$  if  $P_1(C_1) \neq 0$  then  $P_1(C_1)$

else if  $P_2(C_2) \neq 0$  then  $P_2(C_2)$

else if  $L_1(C_1) \cap L_2(C_2) = \{k_2\}$  (single element) then  $k_2$

else if  $P_3(C_3) \neq 0$  then  $P_3(C_3)$

else if  $L_1(C_1) \cap L_2(C_2) \cap L_3(C_3) = \{k_3\}$  (single element) then  $k_3$

else if  $P_4(C_4) \neq 0$  then  $P_4(C_4)$

else ...

## §3.7.2 Compiler Directives and Pragas

These are NOT part of program texts. For example,

```
#pragma omp parallel for shared(n, a) private(i)
#pragma omp parallel for private(x)
#pragma omp parallel for schedule(dynamic, 16)
#pragma comment(linker, "/
export:SomeFunc=DllWork.SomeFunc")
```

compiler options e.g. optimization, profiling, etc.

- handled by scanner or semantic routines.
- Complex pragmas are treated like other statements.

source inclusion, e.g. `#include` in C

- handled by preprocessor or scanner.

conditional compilation, e.g. `#if`, `#endif` in C

- useful for creating program versions.

```
#define sun
#if vax
int i;
#endif
#if sun
float i;
#endif
```

- `#if exp` must be parsed and evaluated during compilation.
- What about nested conditional compilation?

```
#if . . .
#if . . .
#endif
#endif
```

linker/loader commands

```
// Function forwarders to functions in DllWork  
#pragma comment(linker, "  
export:SomeFunc=DllWork.SomeFunc")
```

This pragma tells the linker that the stub DLL should export a function called `SomeFunc`, but that the actual implementation for the function is in the function `SomeFunc` contained in the `DllWork.dll`.

source listing

- trivial, but be careful!
- issues:
  - What if errors are detected at a very late stage?
  - editing source lines
  - correspondence between source lines and listing lines

- Token positions must be remembered.
- UNIX simplifies scanners by ignoring all these *trivia*.

### §3.7.3 Symbol Table

Who is responsible for entering symbols into the symbol table?

- scanner? Consider the example

```
{ int abc
    . . . abc . . .
    { int abc; . . . abc . . . }
}
```

Scanner puts the symbol in a string space.

Do upper case and lower case matter?

- No in Pascal.
- Yes in C.

### §3.7.4 Scanner Termination

How to handle end-of-file?

1. Create a special EOF token.
2. Good for parser, too.

### §3.7.5 Multiple-Character Lookahead

e.g. FORTRAN ignores blanks.

```
DO 2 I = 1, 3
```

```
DO 2 I = 1.3
```

FORTRAN compilers need to look ahead many characters.

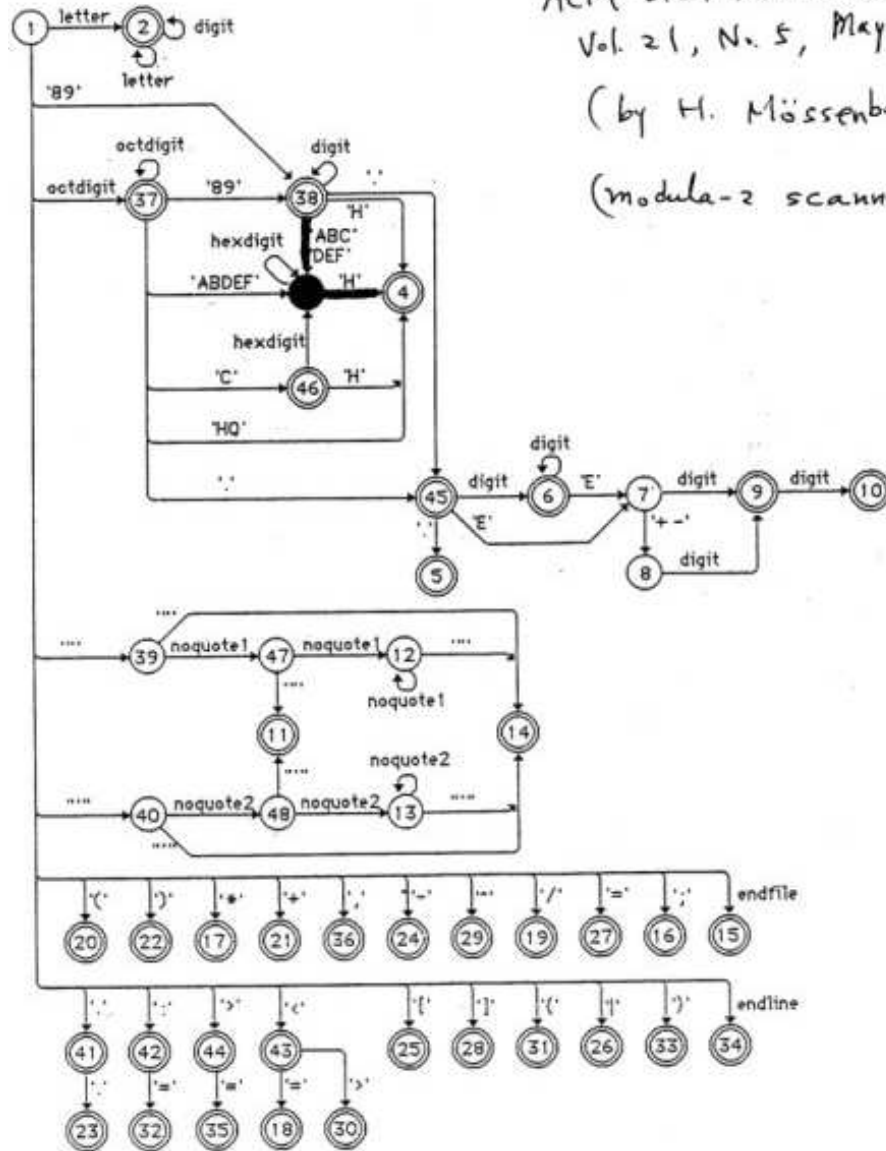
We mentioned that if a token is a prefix of another token, then the scanner needs to look ahead multiple characters. Lookahead is caused by the longest-match rule.

In Pascal or Ada, we need to scan `10..20` as three tokens `10`, `..`, and `20`. 2-character lookahead is necessary (and sufficient) for Pascal and Ada. But not for Modula-2.

ACM SIGPLAN Notices  
 Vol. 21, No. 5, May 1986

(by H. Mössenböck)

(modula-2 scanner)

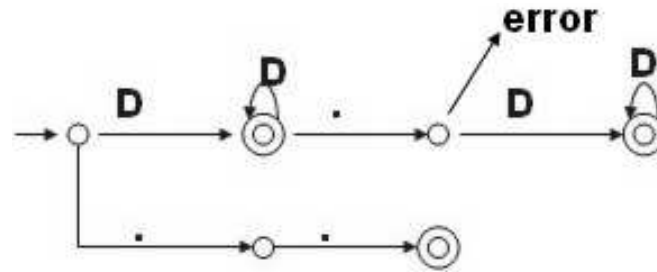


Are there other situations that require multiple-character lookahead?



How to handle multiple-character lookahead?

1. Use pseudo tokens.



Make the error state a pseudo accepting state, which corresponds to the regular expression  $D^+..$

2. Use suffix automata.

Tic (') in Ada

Two usages

1. attributes, e.g., `Array1'length`.
2. character literals, e.g., `'a'`.

We need to scan `typename'('a', 'b')` as seven tokens:

`typename` `'` `(` `'a'` `,` `'b'` `)`

Problem is in the first `'`.

Even multiple-character lookahead cannot help. Need help from the parser.

This is a case where the longest-match rule does not apply.

## §3.7.6 Lexical Error Recovery

Purpose of error recovery

1. Correct errors?
2. Continue compilation and avoid cascading error messages.

When a lexical error occurs, we may discard all characters read so far or delete only the 1st character.

Usually lexical errors are caused by some illegal character, which usually appears at the beginning of a token.

### **runaway string**

Use error tokens to recognize run-away strings.

```
STR = "(NOT("|EOL) | ")*"
```

```
RASTR = "(NOT("|EOL) | ")*EOL
```

Insert a " before EOL.

## runaway comments

two cases:

1. { . . .  
. . . EOF

Use an error token.

COMMENT = { NOT(})\* }

RACOMMENT = { NOT(})\* EOF

2. { . . .  
. . . ( missing } )  
. . .  
{ . . .  
. . . }

Use two kinds of comments. COM1 = { NOT({|})\* }

COM2 = { (NOT({|})\* { NOT({|})\*})\* }

COM2 causes a warning message.

## Nested comments

When do we need nested comments?

Solutions:

1. Use several kinds of comment delimiters such as

{ . . . }

/\* . . . \*/

(\* . . . \*)

New Pascal standard . . .

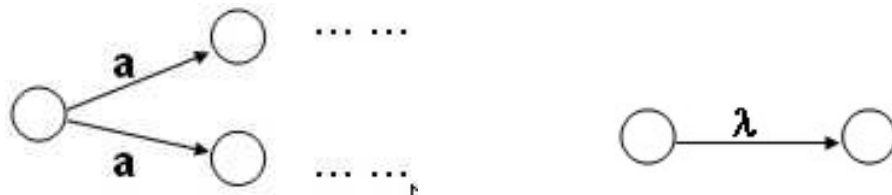
2. Use a counter.

### §3.8 r.e. $\Rightarrow$ fa

Four steps:

1. r.e.  $\Rightarrow$  nondeterministic fa
2. nondeterministic fa  $\Rightarrow$  deterministic fa
3. minimize the number of states (for reducing table size)
4. construct suffix automata by splitting states (for look-aheads)

Two cases an fa is nondeterministic: similar transitions and  $\lambda$ -transitions.



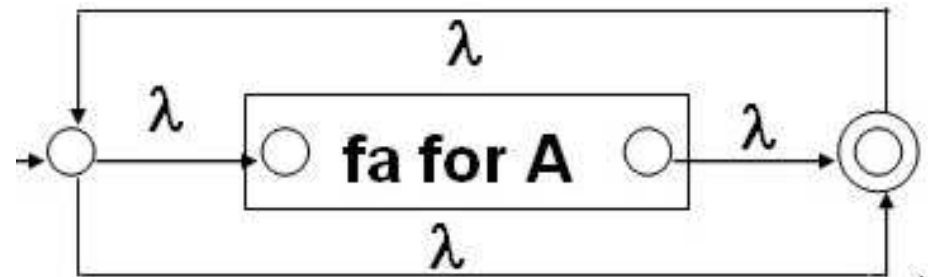
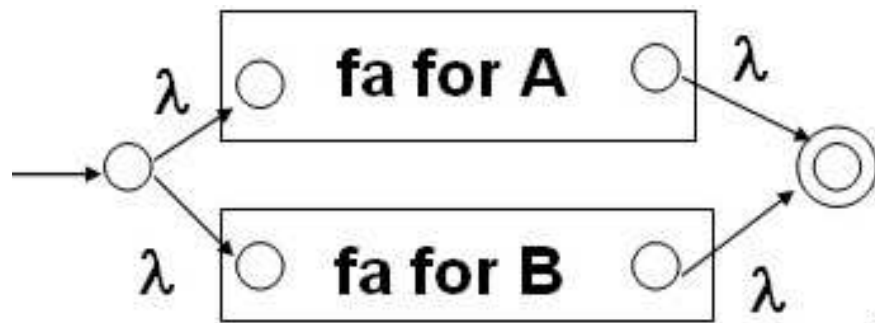
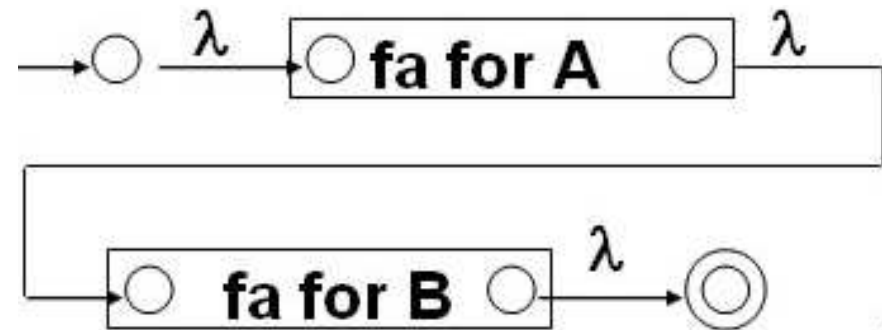
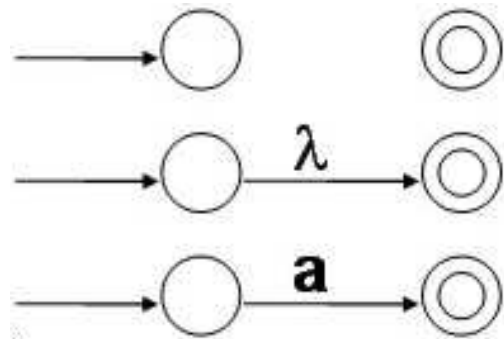
By definition, *an nfa accepts an input string* if and only if there is at least one path in the nfa that begins at the initial state and ends up in an accepting state by scanning the input string.

A theoretical result:  $\text{nfa} = \text{dfa}$  (in the sense that they accept exactly the same set of strings).

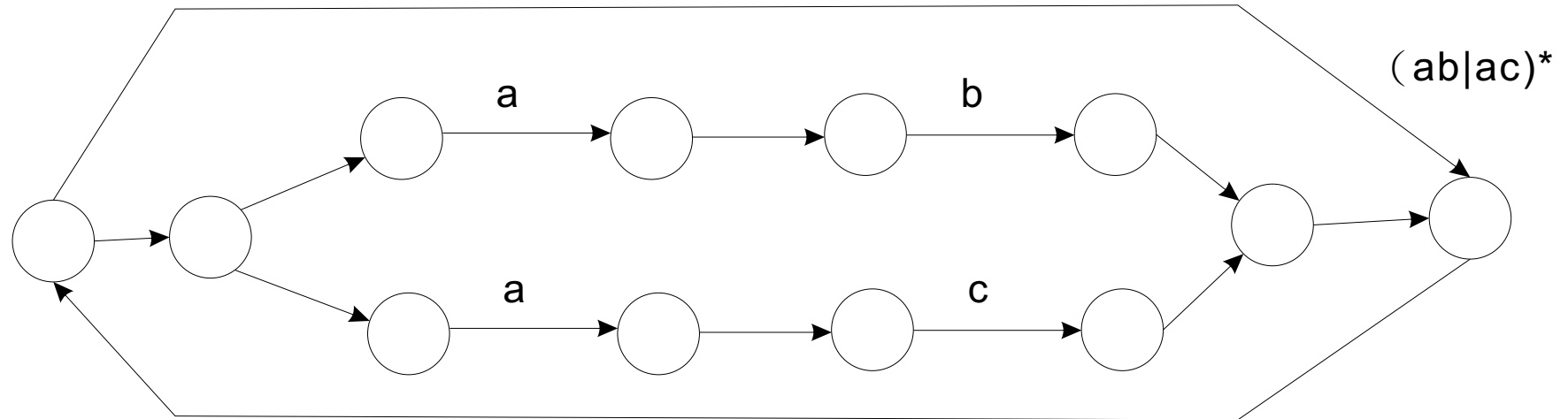
## Step 1. r.e. $\Rightarrow$ nfa.

We first review the definition of r.e.:

1.  $\emptyset$  (denoting the empty set)
2.  $\lambda$  (denoting the set of the null string)
3.  $a$  (any character of the vocabulary)
4.  $A|B$  (or)
5.  $A \cdot B$  (concatenation)
6.  $A^*$  (repetition)



Example. Consider the regular expression  $(ab|ac)^*$ .



## Step 2. nfa $\Rightarrow$ dfa.

Use *subset construction*.

Let  $N$  be an nfa and  $D$  be the equivalent dfa.

Each state of  $D$  corresponds to a set of states of  $N$ . If we run  $N$  and  $D$  on the same input string,  $D$  is in state  $\{x, y, z\}$  if and only if  $N$  is in state  $x$ ,  $y$ , or  $z$ .

Thus,  $D$  keeps track of all possible paths  $N$  might take.

A state  $\{x, y, \dots\}$  of  $D$  is an accepting state if at least one of  $x, y, \dots$  is an accepting state of  $N$ .

The initial state of  $D$  is the set of states reachable from the initial state of  $N$  by  $\lambda$ -transitions.

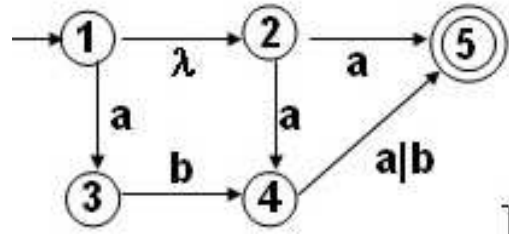
Given an nfa  $N$ , let  $S$  be a set of states of  $N$ . We define the operation  $\text{closure}(S)$  as

$$\text{closure}(S) = \{q \mid \text{there is a state } p \in S, p \xrightarrow{\lambda, *} q\}$$

(that is, reachable by one or more  $\lambda$ -transitions.)

1. Algorithm:
2.  $S := \text{closure}(\{ \text{initial state of } N \} )$
3. Add state  $S$  to the dfa.
4. repeat {
5.     Choose a state  $X$  from the dfa and a character  $c$ .
6.      $T := \{ q \mid \text{there is a state } p \text{ in } X \text{ and } p \xrightarrow{c} q \}$
7.      $T := \text{closure}(T)$  /\*  $T$  is a new state \*/
8.     Add  $T$  to the dfa.
9. } until no more states can be added to the dfa.

Example.



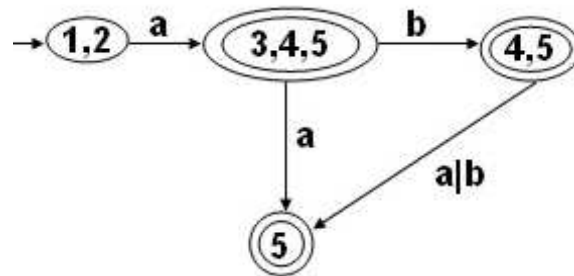
Initial state  $\text{closure}(\{1\}) = \{1, 2\}$ .

$\{1, 2\} \xrightarrow{a} \{3, 4, 5\}$ .  $\text{closure}(\{3, 4, 5\}) = \{3, 4, 5\}$ .

$\{3, 4, 5\} \xrightarrow{a} \{5\}$ .  $\text{closure}(\{5\}) = \{5\}$ .

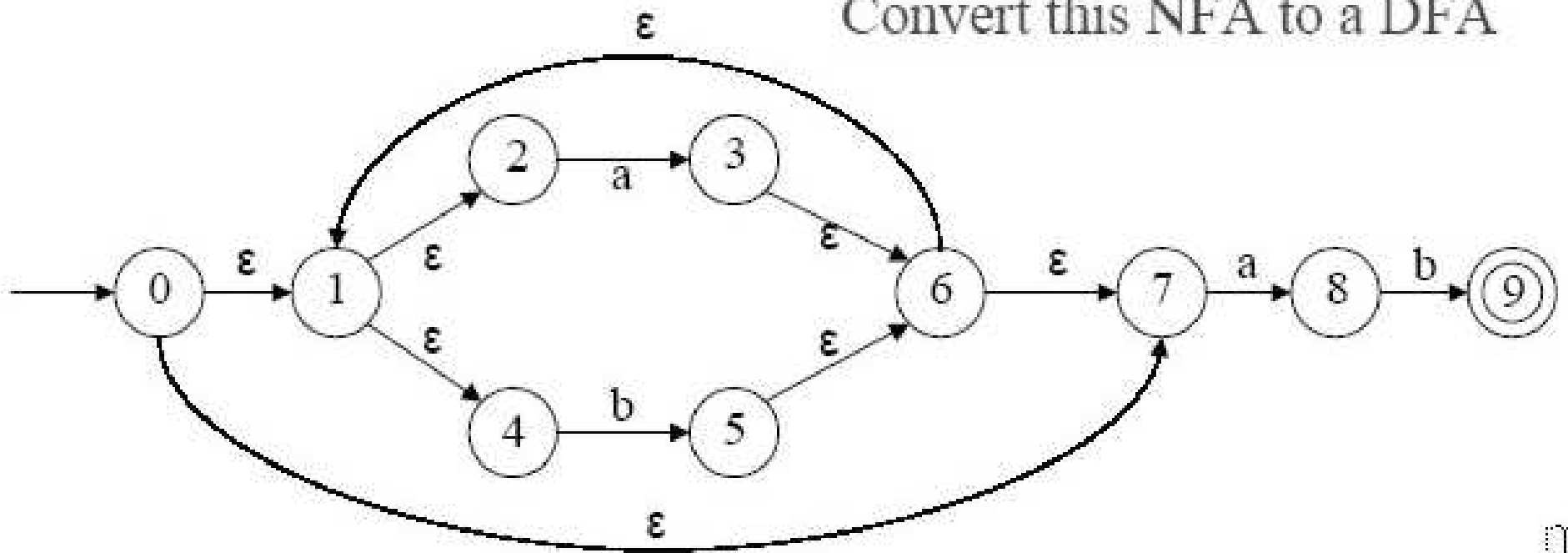
$\{3, 4, 5\} \xrightarrow{b} \{4, 5\}$ .  $\text{closure}(\{4, 5\}) = \{4, 5\}$ .

$\{4, 5\} \xrightarrow{a} \{5\}$ .  $\{4, 5\} \xrightarrow{b} \{5\}$ .



Example. Convert NFA to a DFA.

Convert this NFA to a DFA



*Example.* Two DFAs.

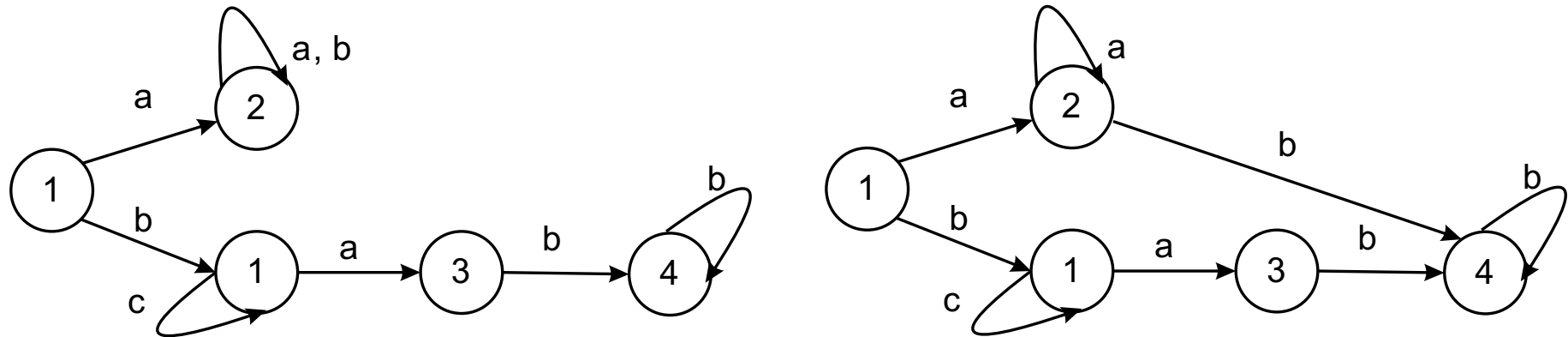


Figure 1: FAs for  $(a(a|b)^*|bc^*ab^+)$  and  $(a+b^*|bc^*ab^+)$ .

Subset construction always terminates since there is only a finite number of subsets.

The dfa is equivalent to the nfa in that they accept the same set of strings.

The dfa may contain  $2^n - 1$  states, where  $n$  is the number of states of the nfa. E.g. (exercise 18, p89)  $(a|b)^*a(a|b)(a|b)\dots(a|b)$ .

Fortunately, this exponential blowup seldom happens in practice.

### Step 3. Minimize the number of states.

There are many dfa's that accept the same language. In order to reduce the table size in the implementation of a dfa, we wish to use the dfa with the least number states (i.e., the *smallest* dfa).

Every dfa has a unique smallest equivalent dfa.

Given a dfa  $M$ , we use splitting to construct the equivalent minimal dfa.

1. Algorithm:
2. Initially, there are two sets, one containing all
3.     the accepting states of  $M$ , the other the
4.     remaining states.
5. repeat {
6.     Choose a set  $A = \{s_1, s_2, \dots, s_n\}$  and a character  $c$ .
7.     Split  $A$  into  $A_1, A_2, \dots, A_k$  so that for all  $i$ ,
8.     if  $s_j, s_k \in A_i$  and  $s_j \xrightarrow{c} t_j$  and  $s_k \xrightarrow{c} t_k$

9.            then  $t_j$  and  $t_k$  are in the same set  $A_m$ .
10.    } until no more set can be split.

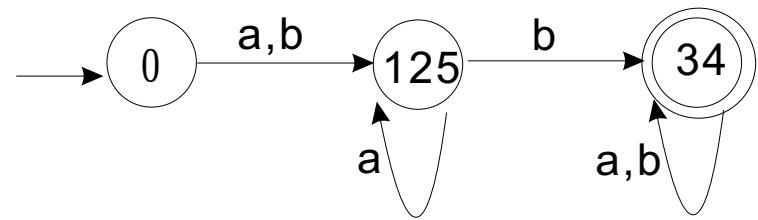
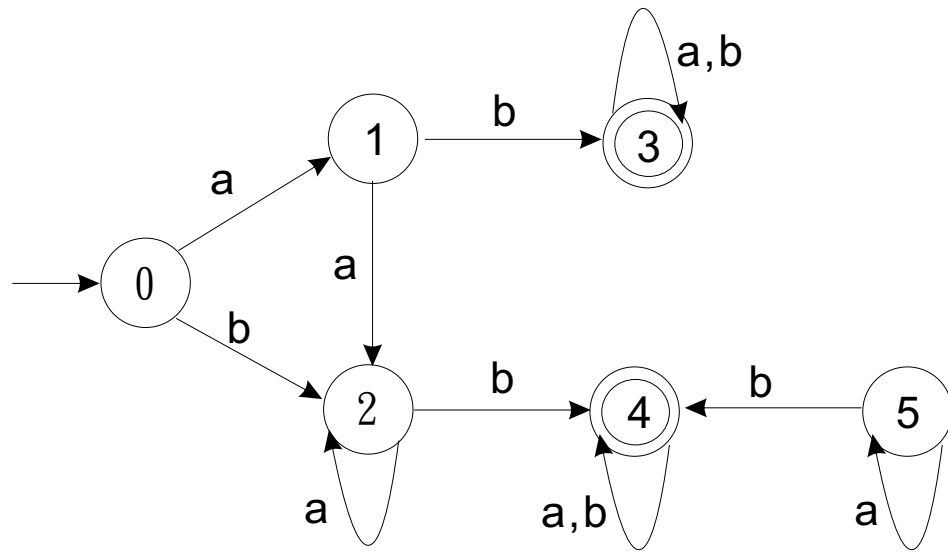
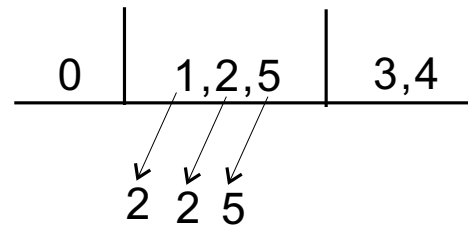
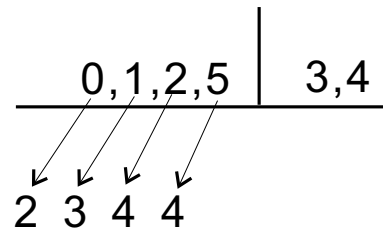
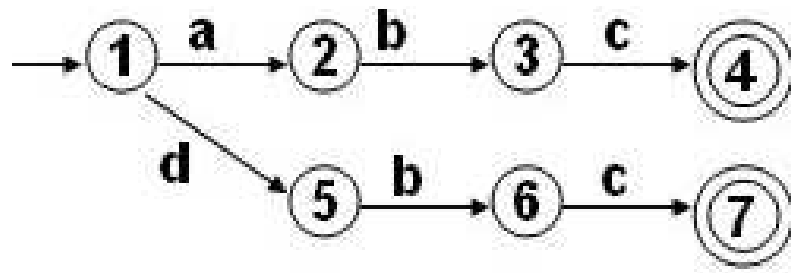


Fig 2.17 Two equivalent dfas



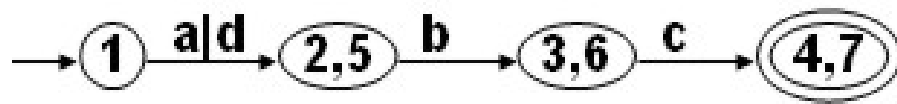


Initially, there are two sets  $\{1, 2, 3, 5, 6\}$  and  $\{4, 7\}$ .

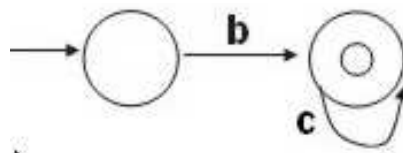
$\{1, 2, 3, 5, 6\}$  splits into  $\{1, 2, 5\}$  and  $\{3, 6\}$  on  $c$ .

$\{1, 2, 5\}$  splits into  $\{1\}$  and  $\{2, 5\}$  on  $b$ .

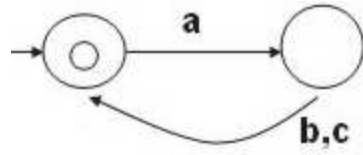
So finally we have



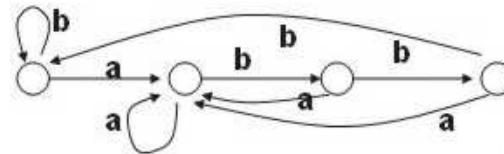
Ex.  $bc^*$ .



Ex.  $(ab|ac)^*$ .



Ex.  $(a|b)^*abb$ .



Exercise. Let  $V = a, b$ . Define

Token  $T1 = a\{b\}^*a$

Token  $T2 = ab\{b\}^*a$ .

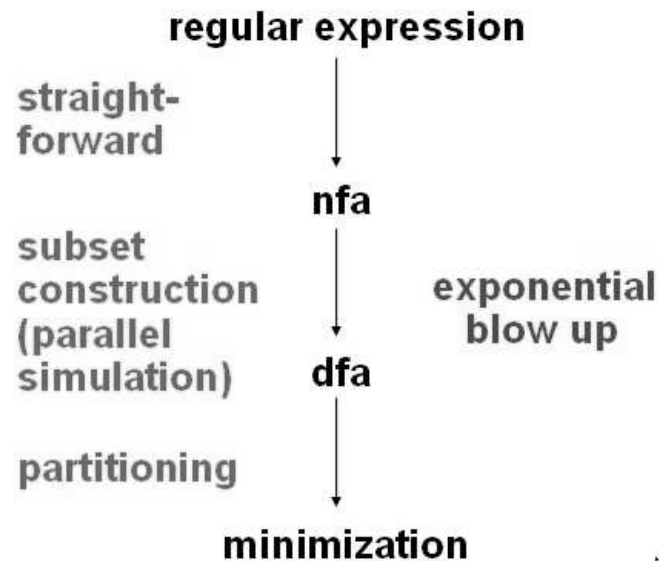
Construct the minimal dfa.

Observations:

- The nfa has at most twice as many states as the number of characters and operators in the regular expression.

Proof. By structural induction.  $\square$

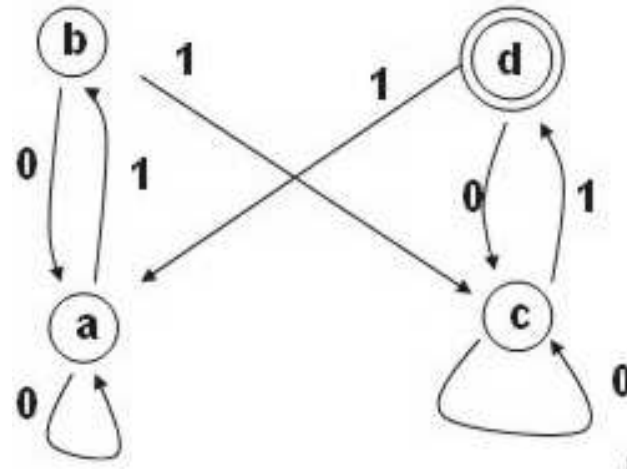
- The construction of nfa from a regular expression  $r$  takes  $O(|r|)$  time.
- The size of the transition table of the constructed nfa is  $O(|r|)$ , given a fixed input vocabulary.



Issues:

1. exponential blow up:  $(0|1)^*0(0|1)(0|1)\dots(0|1)$
2. balanced parentheses problem: CFG  $\{[{}^i]{}^i\}$
3. Ada's attributes: A'b
4. ambiguity
5. lookahead problem

How to construct regular expressions from finite automata?



$$A = \lambda \mid A0 \mid B0 \mid D1$$

$$B = A1$$

$$C = C0 \mid B1 \mid D0$$

$$D = C1$$

**Theorem.** If  $A = \alpha \mid A\beta$  then  $A = \alpha\beta^*$ , where  $\alpha$  and  $\beta$  do not involve  $A$ .

**Ambiguity:** two cases:

1. A string and its prefix both satisfy a regular expression, e.g.,

Totalweight := . . ..

- Use the longest-match rule.
- Lookahead problem.

2. A string may satisfy two (or more) regular expressions. ex.

The string `abbba` satisfies both token definitions.

TOKEN T1 = `abb*a`

TOKEN T2 = `a*b*a*`

- Use the first-match rule.

## Checklist for scanners.

1. How does the scanner handle the string `10..20`?
2. How does the scanner handle the string `"abc""def"`?
3. How does the scanner handle run-away strings (or cross-line strings) and run-away comments? Note that in Ada/CS, there is no run-away comments since every comment is terminated by end-of-line.
4. How does the scanner handle the string `10.3E+5`, `10.3E+BC`, `10.3E+2.34`, and `10.3EBC`? An acceptable solution to `10.3EBC` could be two tokens: a real number `10.3` and an identifier `EBC`. But you may make your own decision.
5. How does the scanner handle a very long identifier? How long can an identifier be? What if the maximum is exceeded?
6. How does the scanner handle a very long string constant? How

long can a string be? What if the maximum is exceeded?

7. How does the scanner handle reserved words? How do you use a table for this purpose?
8. How does the scanner handle illegal characters, such as !, [ and ], which do not appear in Ada/CS?
9. How does the scanner handle the empty string "" ?
10. How does the scanner handle the ' sign as in a character constant 'a' and in an attribute such as d'First?
11. How does the scanner handle end-of-file?
12. How does the scanner recover from errors? What characters are deleted when errors occur?
13. Can the scanner handle nested comments? This is not related to Ada/CS due the format of a comment. But in general, nested comments could be an issue in a scanner.

## *Review*

Regular expressions cannot define the set of balanced parentheses,

$$\{(^i)^i \mid i = 0, 1, \dots\}$$

exponential blowup  $(0|1) * (0|1)(0|1) \dots (0|1)$

the lookahead problem

faster scanner

equivalence of regular expressions

*Question.* Given regular expressions for tokens of a programming language, determine the minimum number of characters of lookahead for the scanner.

# RE2C: A More Versatile Scanner Generator

Table I. Comparison of Generated C Scanners

program	user	time		total	text	space		
		sys				data	bss	total
<i>R4000 / gcc2.3.3 -O</i>								
flex -Cem	10.36	0.87	11.23	5200	4192	48	9440	
flex -Cf	5.44	0.72	6.16	4688	64384	48	69120	
lcc	3.19	0.67	3.86	7328	1216	8256	16800	
gla	2.89	0.63	3.52	11552	3056	144	14752	
re2c	2.54	0.68	3.22	13264	512	0	13776	
re2c -s	2.38	0.67	3.05	11056	4528	0	15584	
<i>R4000 / cc2.11.2 -O -Olimit 5000</i>								
flex -Cem	9.97	0.89	10.86	4704	4240	32	8976	
flex -Cf	6.19	0.72	6.91	4256	64432	32	68720	
lcc	2.74	0.72	3.46	9664	864	8256	18784	
gla	2.46	0.69	3.15	19232	2992	128	22352	
re2c	2.97	0.63	3.60	15088	528	0	15616	
re2c -s	2.94	0.61	3.55	16080	11808	0	27888	

space  
time  
hand-  
coded



*Example.* Find a nondeterministic finite automaton, deterministic finite automaton, and a minimum deterministic finite automaton for the following regular expression:

$$a(bc^*)^+ \mid aa(c^*b)$$

*Answer.* There are 5 states in the minimal deterministic fa. See Figure 2.

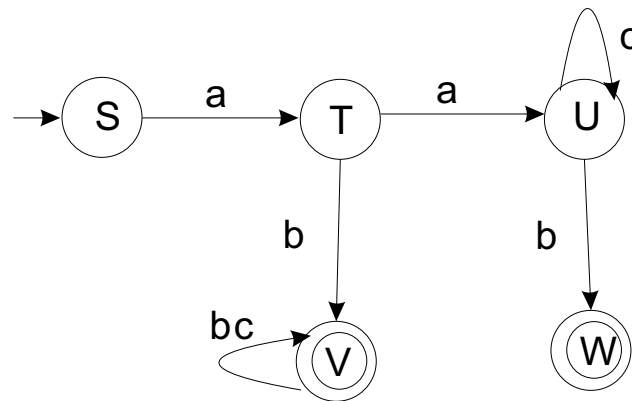


Figure 2: mindfa01.eps