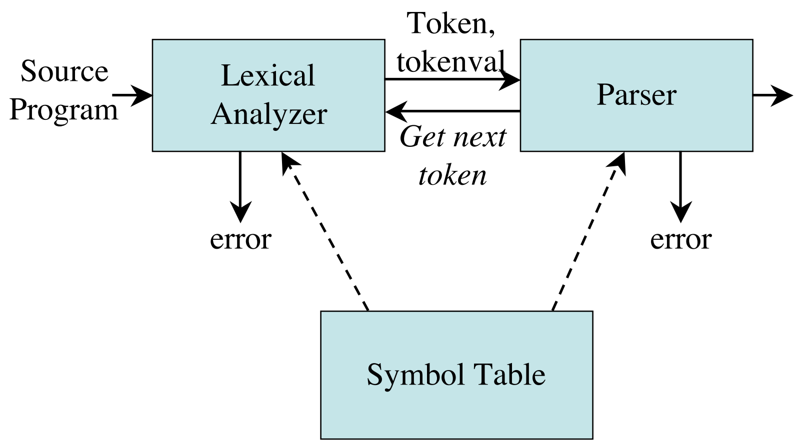
Lexical Analysis and Lexical Analyzer Generators

Chapter 3

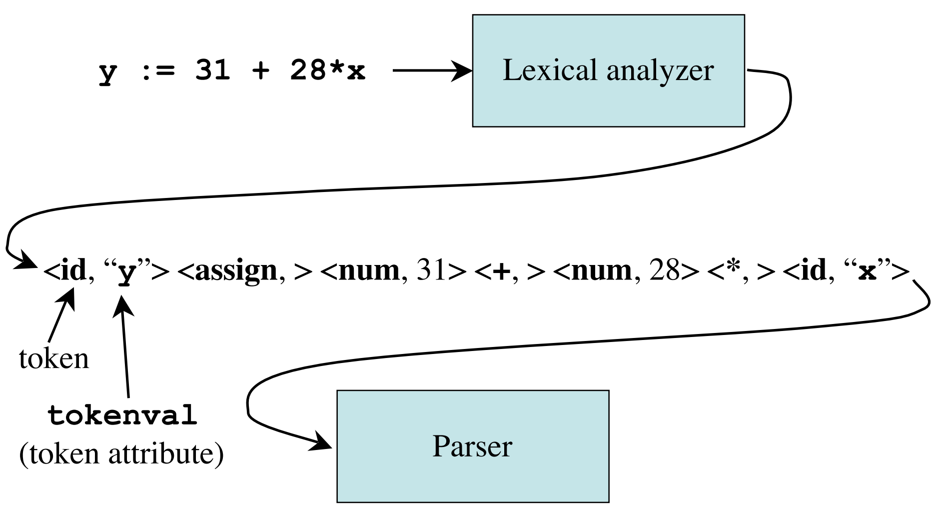
The Reason Why Lexical Analysis is a Separate Phase

- Simplifies the design of the compiler
  - LL(1) or LR(1) with 1 lookahead would not be possible
- Provides efficient implementation
  - Systematic techniques to implement lexical analyzers by hand or automatically
  - Stream buffering methods to scan input
- Improves portability
  - Non-standard symbols and alternate character encodings can be more easily translated

Interaction of the Lexical Analyzer with the Parser



Attributes of Tokens



5

Tokens, Patterns, and Lexemes

- A *token* is a classification of lexical units
  - For example: **id** and **num**
- *Lexemes* are the specific character strings that make up a token
  - For example: **`abc`** and **`123`**
- *Patterns* are rules describing the set of lexemes belonging to a token
  - For example: "*letter followed by letters and digits*" and "*non-empty sequence of digits*"

6

Specification of Patterns for Tokens: Terminology

- An *alphabet* Σ is a finite set of symbols (characters)
- A *string s* is a finite sequence of symbols from Σ
  - $|s|$ denotes the length of string *s*
  - ε denotes the empty string, thus $|\varepsilon| = 0$
- A *language* is a specific set of strings over some fixed alphabet Σ

7

Specification of Patterns for Tokens: String Operations

- The *concatenation* of two strings x and y is denoted by xy
- The *exponentation* of a string s is defined by

  $s^0 = \varepsilon$

  $s^i = s^{i-1}s$ for $i > 0$

  (note that $s\varepsilon = \varepsilon s = s$)

8

Specification of Patterns for Tokens: Language Operations

- *Union*

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

  $LM = \{xy \mid x \in L \text{ and } y \in M\}$
- *Exponentiation*

  $L^0 = \{\varepsilon\}$; $L^i = L^{i-1}L$
- *Kleene closure*

  $L^* = \cup_{i=0,\ldots,\infty} L^i$
- *Positive closure*

  $L^+ = \cup_{i=1,\ldots,\infty} L^i$

Specification of Patterns for Tokens: Regular Expressions

- Basis symbols:
  - ε is a regular expression denoting language $\{\varepsilon\}$
  - $a \in \Sigma$ is a regular expression denoting $\{a\}$
- If r and s are regular expressions denoting languages $L(r)$ and $M(s)$ respectively, then
  - $r \mid s$ is a regular expression denoting $L(r) \cup M(s)$
  - rs is a regular expression denoting $L(r)M(s)$
  - r^* is a regular expression denoting $L(r)^*$
  - (r) is a regular expression denoting $L(r)$
- A language defined by a regular expression is called a *regular set*

Specification of Patterns for Tokens: Regular Definitions

- Naming convention for regular expressions:

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\ldots$$
$$d_n \rightarrow r_n$$

  where r_i is a regular expression over

$$\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$$

- Each d_j in r_i is textually substituted in r_i

11

Specification of Patterns for Tokens: Regular Definitions

- Example:

letter → **`A`** | **`B`** | … | **`Z`** | **`a`** | **`b`** | … | **`z`**
digit → **`0`** | **`1`** | … | **`9`**
id → **letter** (**letter** | **digit**)*

- Cannot use recursion, this is illegal:

digits → **digit digits** | **digit**

12

Specification of Patterns for Tokens: Notational Shorthands

- We frequently use the following shorthands:
  $r^+ = rr^*$
  $r? = r \mid \varepsilon$
  [**`a`**-**`z`**] = **`a`** | **`b`** | **`c`** | … | **`z`**
- For example:
  digit → [**`0`**-**`9`**]
  num → **digit**$^+$ (**. digit**$^+$)? (**E** (**+**|**-**)? **digit**$^+$)?

Regular Definitions and Grammars

Grammar

stmt → **if** *expr* **then** *stmt*

| **if** *expr* **then** *stmt* **else** *stmt*

| ε

expr → *term* **relop** *term*

| *term*

term → **id**

| **num**

Regular definitions

if → `if`

then → `then`

else → `else`

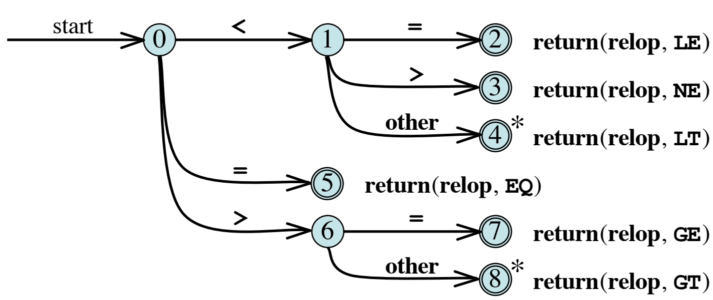
relop → **< | <= | <> | > | >= | =**

id → **letter** (**letter** | **digit**)*

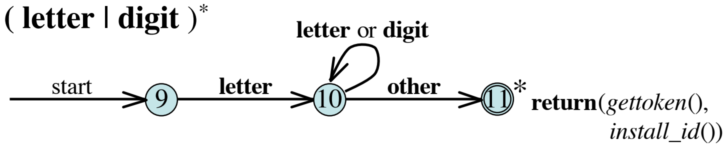
num → **digit**$^+$ (**. digit**$^+$)? (**E** (**+**|**-**)? **digit**$^+$)?

Implementing a Scanner Using Transition Diagrams

relop → **< | <= | <> | > | >= | =**



id → **letter** (**letter** | **digit**)*



15

Implementing a Scanner Using Transition Diagrams (Code)

```
token nexttoken()
{ while (1) {
    switch (state) {
    case 0: c = nextchar();
       if (c==blank || c==tab || c==newline) {
         state = 0;
         lexeme_beginning++;
       }
       else if (c=='<') state = 1;
       else if (c=='=') state = 5;
       else if (c=='>') state = 6;
       else state = fail();
       break;
     case 1:
       …
     case 9: c = nextchar();
       if (isletter(c)) state = 10;
       else state = fail();
       break;
     case 10: c = nextchar();
       if (isletter(c)) state = 10;
       else if (isdigit(c)) state = 10;
       else state = 11;
       break;
     …
```

Decides what other start state is applicable

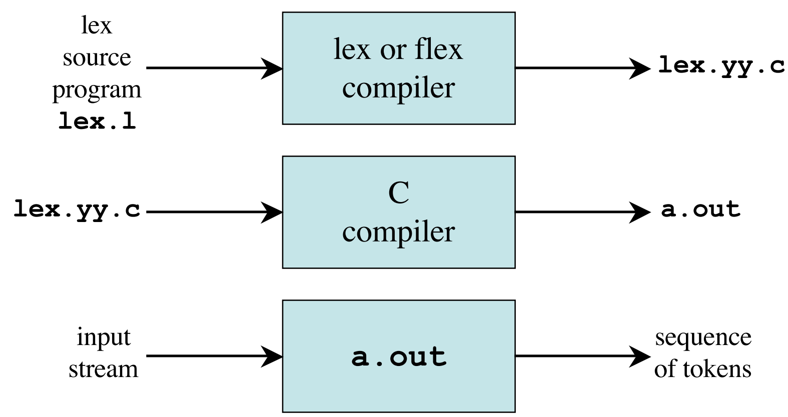
```
int fail()
{ forward = token_beginning;
  swith (start) {
  case  0: start =  9; break;
  case  9: start = 12; break;
  case 12: start = 20; break;
  case 20: start = 25; break;
  case 25: recover(); break;
  default: /* error */
  }
  return start;
}
```

16

The Lex and Flex Scanner Generators

- *Lex* and its newer cousin *flex* are scanner generators
- Systematically translate regular definitions into C source code for efficient scanning
- Generated code is easy to integrate in C applications

Creating a Lexical Analyzer with Lex and Flex



Lex Specification

- A *lex specification* consists of three parts:
  regular definitions, C declarations in **`%{ %}`**
  `%%`
  translation rules
  `%%`
  user-defined auxiliary procedures
- The *translation rules* are of the form:
  p_1 { $action_1$ }
  p_2 { $action_2$ }
  …
  p_n { $action_n$ }

Regular Expressions in Lex

x	match the character **x**
\\.	match the character **.**
"*string*"	match contents of string of characters
.	match any character except newline
^	match beginning of a line
$	match the end of a line
[xyz]	match one character **x**, **y**, or **z** (use **** to escape **-**)
[^xyz]	match any character except **x**, **y**, and **z**
[a-z]	match one of **a** to **z**
r*****	closure (match zero or more occurrences)
r**+**	positive closure (match one or more occurrences)
r**?**	optional (match zero or one occurrence)
$r_1 r_2$	match r_1 then r_2 (concatenation)
$r_1 \mid r_2$	match r_1 or r_2 (union)
(r **)**	grouping
$r_1 \backslash r_2$	match r_1 when followed by r_2
{d**}**	match the regular expression defined by d

Example Lex Specification 1

Translation rules

Contains the matching lexeme

```
%{
#include <stdio.h>
%}
%%
[0-9]+   { printf("%s\n", yytext); }
.|\n     { }
%%
main()
{ yylex();
}
```

Invokes the lexical analyzer

```
lex spec.l
gcc lex.yy.c -ll
./a.out < spec.l
```

Example Lex Specification 2

Translation rules

Regular definition

```
%{
#include <stdio.h>
int ch = 0, wd = 0, nl = 0;
%}
delim      [ \t]+
%%
\n         { ch++; wd++; nl++; }
^{delim}   { ch+=yyleng; }
{delim}    { ch+=yyleng; wd++; }
.          { ch++; }
%%
main()
{ yylex();
  printf("%8d%8d%8d\n", nl, wd, ch);
}
```


Example Lex Specification 3

Translation rules

Regular definitions

```
%{
#include <stdio.h>
%}
digit      [0-9]
letter     [A-Za-z]
id         {letter}({letter}|{digit})*
%%
{digit}+   { printf("number: %s\n", yytext); }
{id}       { printf("ident: %s\n", yytext); }
.          { printf("other: %s\n", yytext); }
%%
main()
{ yylex();
}
```

23

Example Lex Specification 4

```
%{ /* definitions of manifest constants */
#define LT (256)
…
%}
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
%%
{ws}       { }
if         {return IF;}
then       {return THEN;}
else       {return ELSE;}
{id}       {yylval = install_id(); return ID;}
{number}   {yylval = install_num(); return NUMBER;}
"<"        {yylval = LT; return RELOP;}
"<="       {yylval = LE; return RELOP;}
"="        {yylval = EQ; return RELOP;}
"<>"       {yylval = NE; return RELOP;}
">"        {yylval = GT; return RELOP;}
">="       {yylval = GE; return RELOP;}
%%
int install_id()
…
```

Return token to parser

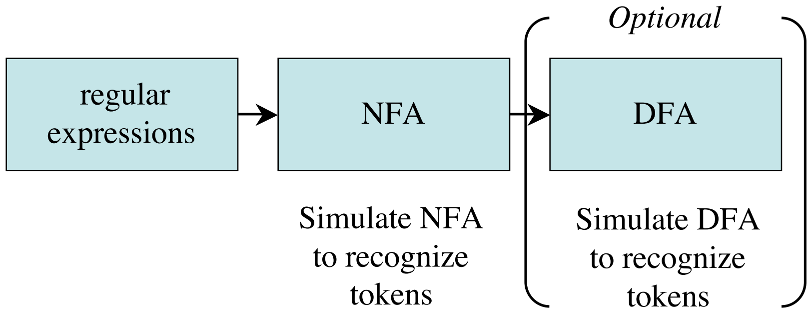
Token attribute

Install **`yytext`** as identifier in symbol table

24

Design of a Lexical Analyzer Generator

- Translate regular expressions to NFA
- Translate NFA to an efficient DFA



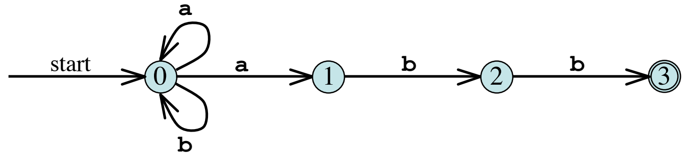
Nondeterministic Finite Automata

- Definition: an NFA is a 5-tuple (S,Σ,δ,s_0,F) where

  S is a finite set of *states*
  Σ is a finite set of *input symbol alphabet*
  δ is a *mapping* from $S \times \Sigma$ to a set of states
  $s_0 \in S$ is the *start state*
  $F \subseteq S$ is the set of *accepting (*or *final) states*

Transition Graph

- An NFA can be diagrammatically represented by a labeled directed graph called a *transition graph*



$S = \{0,1,2,3\}$
$\Sigma = \{\mathbf{a},\mathbf{b}\}$
$s_0 = 0$
$F = \{3\}$

27

Transition Table

- The mapping δ of an NFA can be represented in a *transition table*

$\delta(0,\mathbf{a}) = \{0,1\}$
$\delta(0,\mathbf{b}) = \{0\}$
$\delta(1,\mathbf{b}) = \{2\}$
$\delta(2,\mathbf{b}) = \{3\}$

State	*Input* **a**	*Input* **b**
0	{0, 1}	{0}
1		{2}
2		{3}

28

The Language Defined by an NFA

- An NFA *accepts* an input string *x* **iff** there is some path with edges labeled with symbols from *x* in sequence from the start state to some accepting state in the transition graph
- A state transition from one state to another on the path is called a *move*
- The *language defined by* an NFA is the set of input strings it accepts, such as (**a**|**b**)***abb** for the example NFA

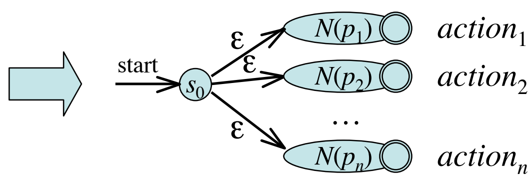
29

Design of a Lexical Analyzer Generator: RE to NFA to DFA

Lex specification with regular expressions

p_1 { $action_1$ }
p_2 { $action_2$ }
…
p_n { $action_n$ }

NFA



Subset construction (optional)

DFA

30

From Regular Expression to NFA (Thompson's Construction)

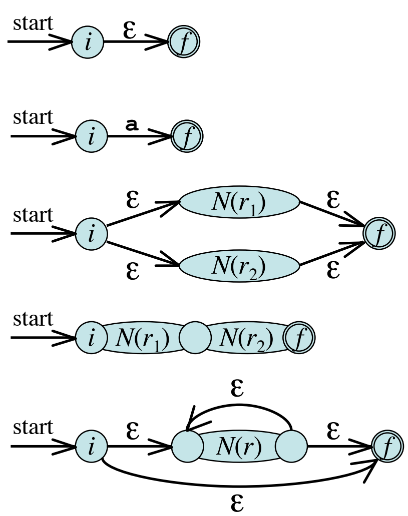
ε

a

$r_1 \mid r_2$

$r_1 r_2$

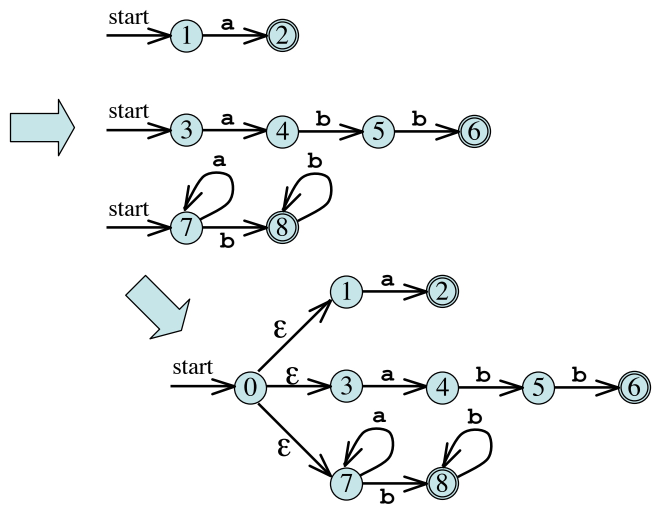
r^*



31

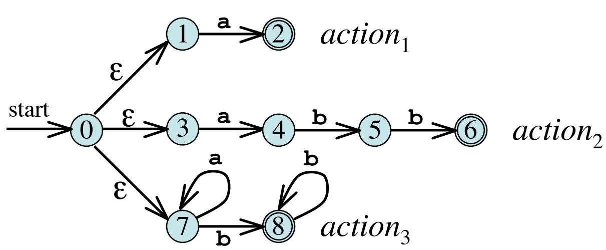
Combining the NFAs of a Set of Regular Expressions

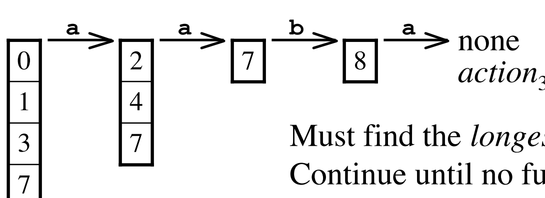
a { *action*$_1$ }
abb { *action*$_2$ }
a*b+ { *action*$_3$ }



32

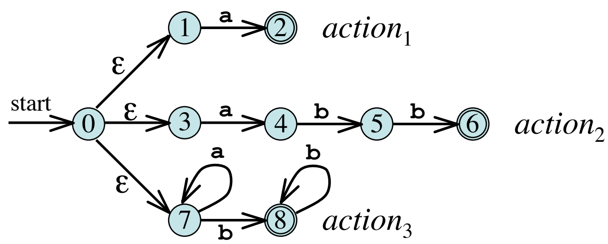
Simulating the Combined NFA Example 1

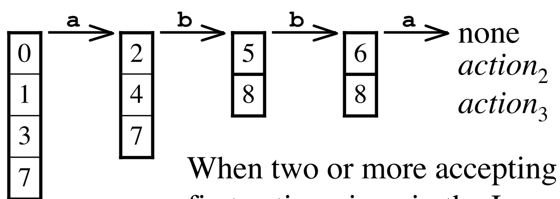




Must find the *longest match*:
Continue until no further moves are possible
When last state is accepting: execute action

Simulating the Combined NFA Example 2





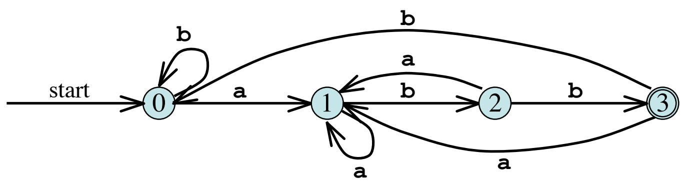
When two or more accepting states are reached, the first action given in the Lex specification is executed

Deterministic Finite Automata

- A *deterministic finite automaton* is a special case of an NFA
  - No state has an ε-transition
  - For each state *s* and input symbol *a* there is at most one edge labeled *a* leaving *s*
- Each entry in the transition table is a single state
  - At most one path exists to accept a string
  - Simulation algorithm is simple

Example DFA

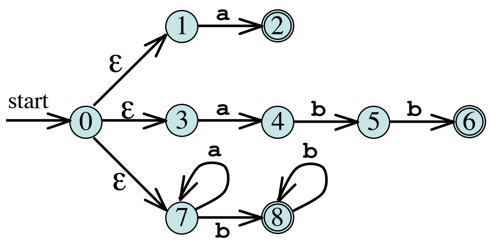
A DFA that accepts (**a**|**b**)***abb**



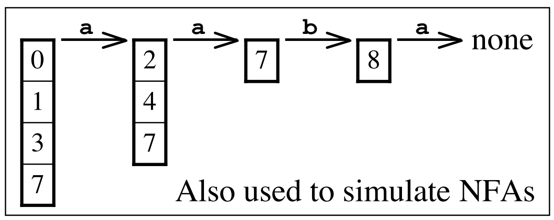
Conversion of an NFA into a DFA

- The *subset construction algorithm* converts an NFA into a DFA using:
  $\varepsilon\text{-}closure(s) = \{s\} \cup \{t \mid s \rightarrow_\varepsilon \ldots \rightarrow_\varepsilon t\}$
  $\varepsilon\text{-}closure(T) = \cup_{s \in T}\, \varepsilon\text{-}closure(s)$
  $move(T,a) = \{t \mid s \rightarrow_a t \text{ and } s \in T\}$
- The algorithm produces:
  Dstates is the set of states of the new DFA consisting of sets of states of the NFA
  Dtran is the transition table of the new DFA

ε-*closure* and *move* Examples



ε-*closure*({0}) = {0,1,3,7}
move({0,1,3,7},**a**) = {2,4,7}
ε-*closure*({2,4,7}) = {2,4,7}
move({2,4,7},**a**) = {7}
ε-*closure*({7}) = {7}
move({7},**b**) = {8}
ε-*closure*({8}) = {8}
move({8},**a**) = $\emptyset$



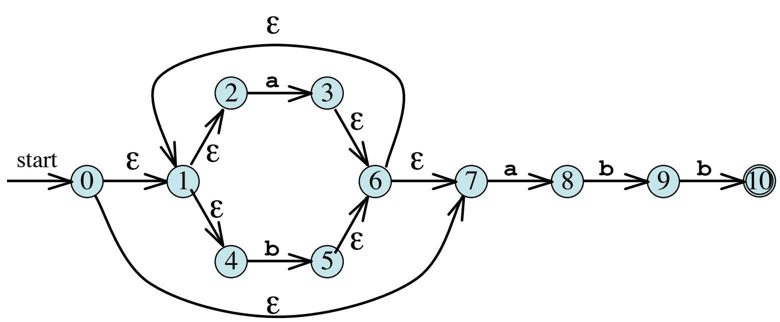
Simulating an NFA using ε-*closure* and *move*

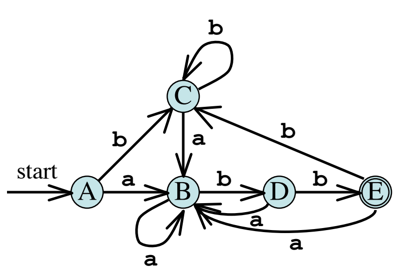
$S := \varepsilon\text{-}closure(\{s_0\})$
$S_{prev} := \emptyset$
$a := nextchar()$
while $S \neq \emptyset$ **do**
  $S_{prev} := S$
  $S := \varepsilon\text{-}closure(move(S,a))$
  $a := nextchar()$
end do
if $S_{prev} \cap F \neq \emptyset$ **then**
  execute *action in* S_{prev}
  return "yes"
else **return** "no"

The Subset Construction Algorithm

Initially, ε-*closure*(s_0) is the only state in *Dstates* and it is unmarked
while there is an unmarked state *T* in *Dstates* **do**
  mark *T*
  for each input symbol $a \in \Sigma$ **do**
    U := ε-*closure*(*move*(*T*,*a*))
    if *U* is not in *Dstates* **then**
      add *U* as an unmarked state to *Dstates*
    end if
    Dtran[*T*,*a*] := *U*
  end do
end do

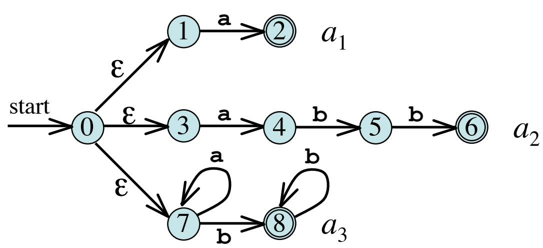
Subset Construction Example 1

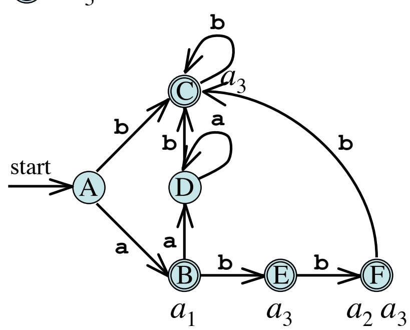




Dstates
A = {0,1,2,4,7}
B = {1,2,3,4,6,7,8}
C = {1,2,4,5,6,7}
D = {1,2,4,5,6,7,9}
E = {1,2,4,5,6,7,10}

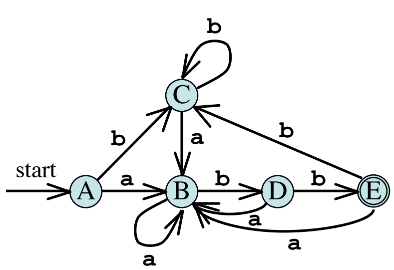
Subset Construction Example 2

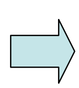


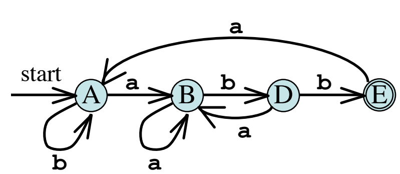


Dstates
A = {0,1,3,7}
B = {2,4,7}
C = {8}
D = {7}
E = {5,8}
F = {6,8}

Minimizing the Number of States of a DFA







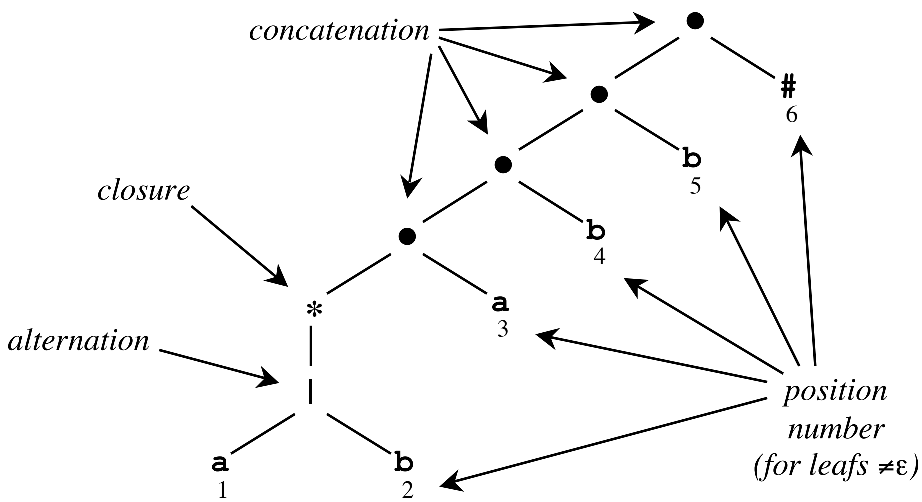
From Regular Expression to DFA Directly

- The *important states* of an NFA are those without an ε-transition, that is if *move*({*s*},*a*) ≠ ∅ for some *a* then *s* is an important state
- The subset construction algorithm uses only the important states when it determines ε-*closure*(*move*(*T*,*a*))

From Regular Expression to DFA Directly (Algorithm)

- Augment the regular expression *r* with a special end symbol # to make accepting states important: the new expression is *r#*
- Construct a syntax tree for *r#*
- Traverse the tree to construct functions *nullable*, *firstpos*, *lastpos*, and *followpos*

From Regular Expression to DFA Directly: Syntax Tree of (a|b)*abb#



From Regular Expression to DFA Directly: Annotating the Tree

- *nullable*(*n*): the subtree at node *n* generates languages including the empty string
- *firstpos*(*n*): set of positions that can match the first symbol of a string generated by the subtree at node *n*
- *lastpos*(*n*): the set of positions that can match the last symbol of a string generated be the subtree at node *n*
- *followpos*(*i*): the set of positions that can follow position *i* in the tree

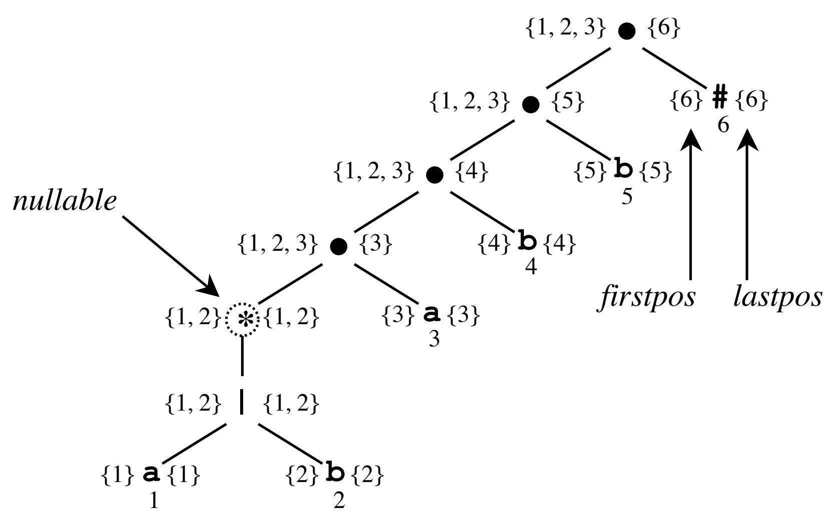
47

From Regular Expression to DFA Directly: Annotating the Tree

Node *n*	*nullable*(*n*)	*firstpos*(*n*)	*lastpos*(*n*)
Leaf ε	true	∅	∅
Leaf *i*	false	{*i*}	{*i*}
| / \ c_1 c_2	*nullable*(c_1) or *nullable*(c_2)	*firstpos*(c_1) ∪ *firstpos*(c_2)	*lastpos*(c_1) ∪ *lastpos*(c_2)
• / \ c_1 c_2	*nullable*(c_1) and *nullable*(c_2)	**if** *nullable*(c_1) **then** *firstpos*(c_1) ∪ *firstpos*(c_2) **else** *firstpos*(c_1)	**if** *nullable*(c_2) **then** *lastpos*(c_1) ∪ *lastpos*(c_2) **else** *lastpos*(c_2)
* | c_1	true	*firstpos*(c_1)	*lastpos*(c_1)

48

From Regular Expression to DFA Directly: Syntax Tree of `(a|b)*abb#`



From Regular Expression to DFA Directly: *followpos*

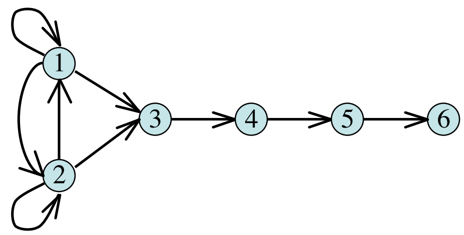
for each node n in the tree **do**
    if n is a cat-node with left child c_1 and right child c_2 **then**
        for each i in $lastpos(c_1)$ **do**
            $followpos(i) := followpos(i) \cup firstpos(c_2)$
        end do
    else if n is a star-node
        for each i in $lastpos(n)$ **do**
            $followpos(i) := followpos(i) \cup firstpos(n)$
        end do
    end if
end do

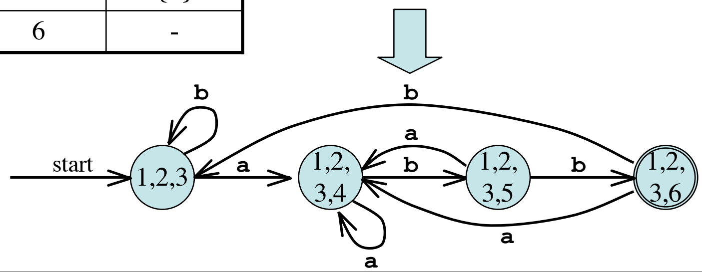
From Regular Expression to DFA Directly: Algorithm

$s_0 := firstpos(root)$ where *root* is the root of the syntax tree
Dstates := $\{s_0\}$ and is unmarked
while there is an unmarked state T in *Dstates* **do**
    mark T
    for each input symbol $a \in \Sigma$ **do**
        let U be the set of positions that are in $followpos(p)$
            for some position p in T,
            such that the symbol at position p is a
        if U is not empty and not in *Dstates* **then**
            add U as an unmarked state to *Dstates*
        end if
        $Dtran[T,a] := U$
    end do
end do

From Regular Expression to DFA Directly: Example

Node	*followpos*
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	-





Time-Space Tradeoffs

Automaton	*Space (worst case)*	*Time (worst case)*
NFA	$O(\|r\|)$	$O(\|r\|\times\|x\|)$
DFA	$O(2^{\|r\|})$	$O(\|x\|)$