

Chapter 3
Lexical Analysis



Outline

- The role of the lexical analyzer
- Input buffering
- Specification of tokens
- Recognition of tokens
- The lexical-analyzer generator Lex
- Finite automata
- From regular expressions to automata
- Design of a lexical-analyzer generator
- Optimization of DFA-based pattern matchers



The Role of the Lexical Analyzer

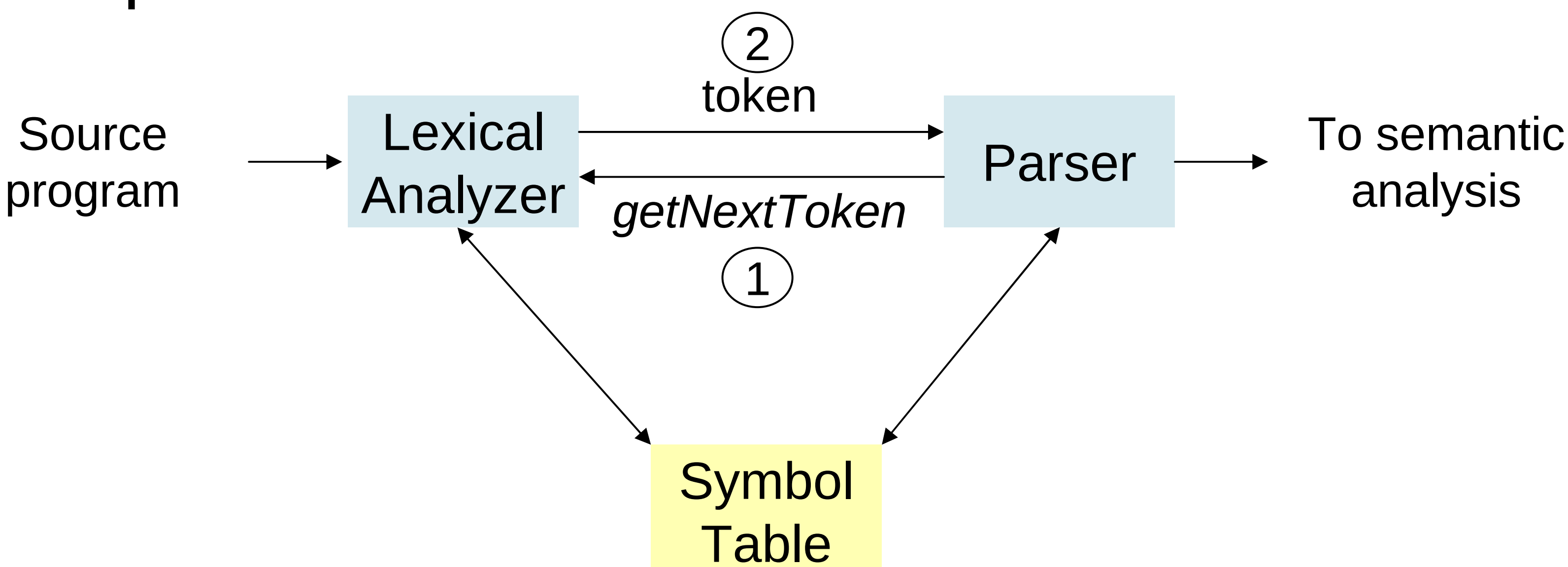


Lexical Analyzer (Scanner)

- The main tasks of the lexical analyzer
  - Read the input characters of the source program,
  - Group them, and
  - Produce a sequence of tokens for each lexeme in the source program.
    - When a lexeme constituting an identifier is found, the lexeme is put to the symbol table.
- Other tasks of the lexical analyzer
  - Strip out comments and whitespace (blank, newline, tab, and other characters that separate tokens in the input)
  - Correlate error messages generated by the compiler with the source program. (e.g., line number for error message)

Lexical Analyzer (Cont.)

- The parser calls the lexical analyzer that reads characters from its input until it can identify the next lexeme and produce the next token for the compiler.





Lexical Analyzer (Cont.)

- Lexical analyzers are sometimes divided into two processes:
  - Scanning:
    - Consist of the simple processes (that do not require tokenization of the input).
    - E.g., deletion of comments, compaction of consecutive whitespace characters into one
  - Lexical analysis:
    - Produce the sequence of tokens as output.

Lexical Analysis vs. Parsing

- Reasons to separate lexical analysis from syntax analysis:
  - Simplicity of the design
    - A parse that has to deal with comments and whitespace would be considerably more complex.
  - Compiler efficiency
    - A separate lexical analyzer allows to adopt specialized buffering techniques to speed up reading input characters.
  - Compiler portability
    - Input-device-specific peculiarities (特質) can be restricted to the lexical analyzer.



Tokens, Patterns, and Lexemes

- Token
  - A token is a pair consisting of a token name and an optional attribute value.
    - The token name is an abstract symbol representing a kind of lexical unit (e.g., keyword) or a sequence of input characters (e.g., identifier).
    - The token names are the input symbols that the parser processes.
- Pattern
  - A pattern is a description of the lexeme forms that a token may take.
    - In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.
    - For identifiers, the pattern is a more complex structure matched by many string.
- Lexeme
  - A lexeme
    - Is a sequence of characters in the source program matches the pattern for a token, and
    - Is identified by the lexical analyzer as an instance of that token.



Example of Tokens

Token name (or referred to as **token**)	Informal description	Sample lexemes
if	characters i, f	if
else	characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters and digits	pi, score, D2
number	Any numeric constant	3.12159, 0, 6.02e23
literal	Anything but surrounded by “	“core dumped”

- We often refer to a token by its token name.
- We generally write token names in **boldface**.



Classes of Tokens

- Classes for most tokens:
  - One token for one keyword
    - The pattern for a keyword is the same as the keyword itself.
  - Tokens for the operators
    - Either individually or in classes (e.g., **comparison**)
  - One token representing all identifiers
    - E.g., **id**
  - One or more tokens representing constants
    - E.g., **number** for numeric constants, and **literal** for strings constants
  - Tokens for each punctuation symbol
    - E.g., left and right parentheses, comma, and semicolon
- E.g., printf(“Total = %d\n”, score);
  - printf and score are lexemes matching the pattern for token **id**.
  - “Total = %d\n” is a lexeme matching token **literal**.



Attributes of Tokens

- When more than one lexeme can match a pattern, the lexical analyzer must provide additional information.
  - In many cases, the lexical analyzer returns to the parser
    - Not only a token name,
    - But also an attribute value that describes the lexeme represented by the token.
  - Assume each token has at most one associated attribute:
    - The attribute may have a structure that combines several pieces of information.
    - E.g., The token **id** whose attribute is a pointer pointing to the symbol table for its corresponding information (e.g., a structure for its lexeme, its type, and the location at which it is first found)



Token Names and Associated Attribute Values

- A Fortran statement: E = M * C ** 2 can be written as a sequence of pairs (i.e., tokens):

Token name

attribute

<**id**, pointer to symbol-table entry for E>
<**assign_op**>
<**id**, pointer to symbol-table entry for M>
<**mult_op**>
<**id**, pointer to symbol-table entry for C>
<**exp_op**>
<**number**, integer value 2>

Operators, punctuation, and keywords don’t need an attribute value.

In practice, a typical compiler stores a character string for the constant with a pointer pointing to the string.



Lexical Errors

- Lexical analyzers are hard to tell source-code errors without the aid of other components.
  - E.g., fi (a == f(x)) …
    The string *fi* is a transposition of the keyword **if** or a valid lexeme for the token **id**?
  - The parse could help identify a transposition error.



Error-Recovery Strategies

- When the lexical analyzer is unable to proceed because no pattern for tokens matches any prefix of the remaining input, the following error-recovery strategies could be adopted:
  1. Delete successive characters from the remaining input until the lexical analyzer can find a well-formed token. (panic mode recovery)
  2. Delete one character from the remaining input.
  3. Insert a missing character into the remaining input.
  4. Replace a character by another character.
  5. Transpose two adjacent characters.
     - The simplest strategy is to see whether a prefix of the remaining input can be transformed into a valid lexeme by a single transformation. (because most lexical errors involve a single character)



Input Buffering



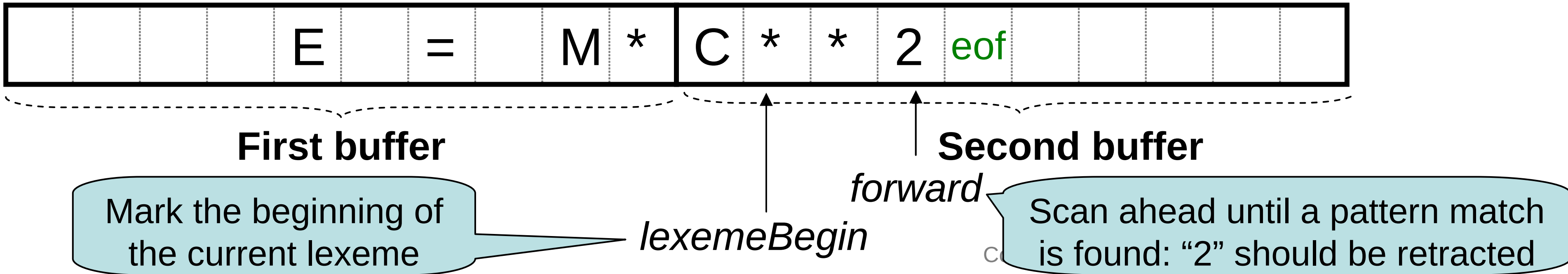
The Problem of Recognizing Lexemes

- We often have to look one or more characters beyond the next lexeme.
  - E.g., an identifier can’t be identified until we see a character that is not a letter or digit.
  - E.g., In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=.
- Two-buffer scheme could handle large lookaheads safely so as to improve the speed on reading the source program.



Two-Buffer Scheme

- Two buffers reloaded alternately to reduce the amount of overhead required to process a single input character.
  - E.g., Each buffer is of the same size *N*, and *N* is usually the size of a disk block (e.g., 4096 bytes).
    - One system read command can read N characters into a buffer.
    - If fewer than *N* characters remain in the input file, then a special character (i.e., **eof**) marks the end of the source file.
  - Once the next lexeme is determined,
    - The lexeme is recorded as an attribute value of a token returned to the parser.
    - Then, *forward* is set to the character at its right end, and *lexemeBegin* is set to the character immediately after the lexeme just found.

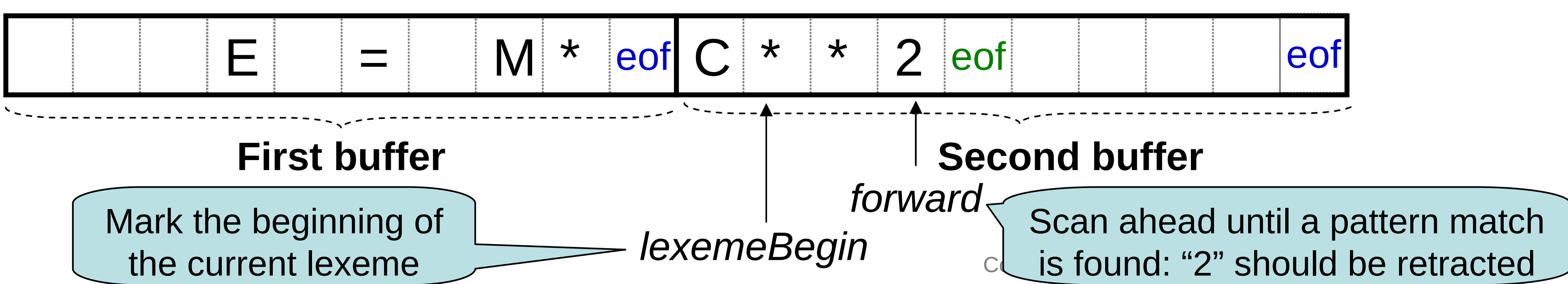




Two-Buffer Scheme with Sentinels

- Whenever we advance forward, we make two tests:
  - Test the end of the buffer
  - Then, determine what character is read (a multiway branch)
- To combine the two tests in one, we can add a sentinel character (Sentinel is a special character that can't be part of the source program.)
  - at the end of each buffer and
  - at the end of the entire input.

If a long string (> N) is encountered, we can treat the long string as a concatenation of strings to prevent buffer overflow.





Lookahead Code with Sentinels

```
Switch (*forward++) {
    case eof:
        if (forward is at the end of the first buffer) {
            reload secnod buffer;
            forward = beginning of second buffer;
        }
        else if (forward is at end of the second buffer) {
            reload first buffer;
            forward = beginning of first buffer;
        }
        else  // eof within a buffer marks the end of input
        break;
    Cases for the other characters
}
```

Multiway branch:
In practice, a multiway branch depending on the input character that is the index of an array of addresses. Only a jump to the indexed address is needed for a selected case.



Specification of Tokens



Strings and Languages

- An alphabet (字母) is any finite set of symbols. Typical examples are:
  - Binary (alphabet): the set {0, 1}
  - ASCII (alphabet): important alphabet used in many systems
  - Unicode (alphabet): including approximately 100,000 characters
- A string over an alphabet is a finite sequence of symbols drawn from that alphabet.
  - In language theory, “sentence” and “word” are often used as synonyms for “string.”
  - $|s|$ is the length of a string *s*.
    - E.g., **banana** is a string of length six.
    - E.g., ε denotes the empty string whose length is zero.
- A language is any countable set of strings over some fixed alphabet.



Terms for Parts of Strings

- A prefix of string *s* is any string obtained by removing zero or more symbols from the end of *s*.
  - E.g., ban, banana, and ε are prefixes of banana.
- A suffix of string *s* is any string obtained by removing zero or more symbols from the beginning of *s*.
  - E.g., nana, banana, and ε are suffixes of banana.
- A substring of *s* is obtained by deleting any prefix and suffix from *s*.
  - E.g., banana, nan, and ε are substrings of banana.
- A proper prefix, suffix or substring of a string *s* doesn't include ε and *s*.
- A subsequence of string *s* is any string fromed by deleting zero or more characters from *s*.
  - E.g., baan is a subsequence of banana.

Operations on Languages

Operation	Definition and Notation
Union of *L* and *M*	*L* U *M* = {*s* | *s* is in *L* or in *M*}
Concatenation of *L* and *M*	*LM* = {*st* | *s* is in *L* and t is in *M*}
(*Kleene*) *closure* of *L*	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of *L*	$L^+ = \cup_{i=1}^{\infty} L^i$

Concatenation of *L* one or more times

Concatenation of *L* zero or more times

- E.g., *x* = dog and *y* = house
  - *xy* = doghouse
- For any string *s*,
  - $\varepsilon s = s\varepsilon = s$
  - $s^0 = \varepsilon$
  - $s^1 = s$, $s^2 = ss$, $s^3 = sss$, $s^i = s^{i-1}s$



Operations on Languages (Cont.)

- E.g.,
  - Let ***L*** be the set of letters {A, B, …, Z, a, b, …, z} and let ***D*** be the set if digits {0, 1, …, 9}
    - $L \cup D$ is the set of letters and digits (62 strings of length one)
    - LD is the set of 520 strings of length two (one letter and one digit)
    - L^4 is the set of all 4-letter strings
    - $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter
    - D^+ is the set of all strings of one or more digits

Regular Expressions

- Regular expressions are an important notation for specifying lexeme patterns.
- For example:
  - If *letter_* is established to stand for any letter or the underscore, and *digit* is established to stand for any digit, then the language of C identifiers is described as

    *letter_(letter_ | digit)**
- Two basic rules over some alphabet $\sum$:
  - ε is a regular expression, and $L(\varepsilon)$ is $\{\varepsilon\}$, the empty string.
  - If **a** is a symbol in the alphabet $\sum$, then **a** is regular expression and $L(\mathbf{a}) = \{a\}$.

Boldface for regular expression

italics for regular symbols



Induction (歸納) of Regular Expressions

- Suppose *r* and *s* are regular expressions denoting languages *L*(*r*) and *L*(*s*), respectively.
  - (*r*)|(*s*) denotes the language *L*(*r*) U *L*(*s*)
  - (*r*)(*s*) denotes the language *L*(*r*)*L*(*s*)
  - $(r)^*$ denotes the language $(L(r))^*$
  - (*r*) denotes *L*(*r*) → we can add additional pairs of parentheses around expressions
- Unnecessary pairs of parentheses can be dropped if we adopt the following conventions:
  - The unary operator * has the highest precedence and is left associative.
  - Concatenation has the second highest precedence and is left associative.
  - | has the lowest precedence and is left associative.

E.g., $(a)|((b)^*(c)) = a|b^*c$



Induction of Regular Expressions (Cont.)

- Let the alphabet $\sum$ = {a, b}
  - The regular expression **a|b** denotes the language {*a*, *b*}.
  - (**a**|**b**)(**a**|**b**) denotes {*aa*, *ab*, *ba*, *bb*}, the language of all strings of length two over the alphabet $\sum$. (= aa|ab|ba|bb)
  - a^* denotes the language consisting of all strings of zero or more a's. I.e., {ε, *a*, *aa*, *aaa*, …}
  - $(\mathbf{a}|\mathbf{b})^*$ denotes the language consisting of zero or more instances of a or *b*. I.e., all strings of *a*'s and *b*'s {ε, *a*, *b*, *aa*, *ab*, *ba*, *bb*, *aaa*, …} (= $(a^*b^*)^*$)



Algebraic Laws for Regular Expressions

- A language that can be defined by a regular expression is call a regular set.
  - If two regular expressions r and s denote the same regular set, then r = s. E.g., (**a**|**b**) = (**b**|**a**)

<table>
<tr><th>Law</th><th>Description</th></tr>
<tr><td>$r|s = s|r$</td><td>| is commutative (交換律)</td></tr>
<tr><td>$r|(s|t) = (r|s)|t$</td><td>| is associative (結合律)</td></tr>
<tr><td>r(st) = (rs)t</td><td>Concatenation is associative</td></tr>
<tr><td>$r(s|t) = rs|rt;\ (s|t)r = sr|tr$</td><td>Concatenation distributions (分配律) over |</td></tr>
<tr><td>$\varepsilon r = r\varepsilon = r$</td><td>ε is the identity for concatenation</td></tr>
<tr><td>$r^* = (r|\varepsilon)^*$</td><td>ε is guaranteed in a closure</td></tr>
<tr><td>$r^{**} = r^*$</td><td>* is idempotent (i.e., applied multiple times without changing the result)</td></tr>
</table>



Regular Definitions

- If ∑ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$$\begin{aligned} d_1 &\rightarrow r_1 \\ d_2 &\rightarrow r_2 \\ &\dots \\ d_n &\rightarrow r_n \end{aligned}$$

  where
  - Each d_i is a new symbol that is not in ∑ and not the same as any other of the d's, and
  - Each r_i is a regular expression over the alphabet $\sum \cup \{d_1, d_2, \ldots, d_{i-1}\}$.
- By restricting r_i to ∑ and the previous defined d's,
  - The recursive definitions can be avoided, and
  - A regular expression can be constructed for each r_i over ∑ alone.



Regular Definition for C Languages

- Regulation definition for C's Identifiers

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

- Regulation definition for C's unsigned numbers (e.g., 5280, 0.01234, 6.3366E4, or 1.89E-4)

digit	→	0 | 1 | … | 9
digits	→	*digit digit**
optionalFraction	→	. *digits* | ε
optionalExponent	→	(E (+ | - | ε) *digits*) | ε
number	→	*digits optionalFraction optionalExponent*

At least one digit must follow the dot



Extensions of Regular Expressions

- Extensions of the regular expression introduced by Kleene:
  - One or more instances
    - The unary postfix operator $^{+}$ represents the positive closure of a regular expression and its language.
    - i.e., if r is a regular expression, then $(r)^{+}$ denotes the language $(L(r))^{+}$.
    - Two useful algebraic laws:
      - $r^{*} = r^{+} \mid \varepsilon$
      - $r^{+} = rr^{*} = r^{*}r$
  - Zero or one instance
    - The unary postfix operator **?** means "zero or one occurrence."
    - E.g., $r? = r \mid \varepsilon$ or $L(r?) = L(r) \cup \{\varepsilon\}$
    - ? has the same precedence and associativity as * and $^{+}$
  - Character classes (shorthand regular expression)
    - A regular expression $a_1 \mid a_2 \mid \ldots \mid a_n = [a_1a_2\ldots a_n]$
    - If $a_1, a_2, \ldots, a_n$ form a logical sequence (e.g., consecutive uppercase letters), then $a_1 \mid a_2 \mid \ldots \mid a_n = [a_1\text{-}a_n]$



Regular Definition with Shorthand

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

*letter_* → [A-Za-z_]
digit → [0-9]
id → *letter_* (*letter_* | *digit*)*

---

digit → 0 | 1 | … | 9
digits → *digit digit**
optionalFraction → . *digits* | ε
OptionalExponent → (E (+ | - | ε) *digits*) | ε
number → *digits optionalFraction OptionalExponent*

digit → [0-9]
digits → *digit*$^+$
number → *digits* (. digits)? (E [+-]? *digits*)?



Recognition of Tokens



Patterns for Tokens

- The terminals of the grammar (which are **if**, **then**, **else**, **relop**, **id**, and **number**) are the names of tokens as far as the lexical analyzer is concerned.

stmt → **if** *expr* **then** *stmt*
  | **if** *expr* **then** *stmt* **else** *stmt*
  | ε
expr → *term* **relop** *term*
  | *term*
term → **id**
  | **number**

A Grammar for branching statements (similar to Pascal)

digit	→	[0-9]
digits	→	*digit*$^+$
number	→	*digits (. digits)?* (E[+-]? *digits*)?
letter	→	[A-Za-z]
id	→	*letter* (*letter* | *digit*)*
if	→	if
then	→	then
else	→	else
relop	→	< | > | <= | >= | = | <>

Patterns for tokens

The lexical analyzer recognizes the keywords (i.e., reserved words) **if**, **then**, and **else**, as well as lexemes that match the patterns for *relop*, *id*, and *number*.



Whitespace

- The lexical analyzer also strips out whitespace by recognizing the “token” *ws* defined by:

$$ws \rightarrow (\ \mathbf{blank} \mid \mathbf{tab} \mid \mathbf{newline})^{+}$$

Abstract symbols to express ASCII characters of the same names.

- Token *ws* is different from the other tokens in that the lexical analyzer does not return it to the parser, but rather restarts the lexical analysis from the character following the whitespace.



Patterns and Attribute Values of Tokens

Lexemes	Token Name	Attribute Value
Any *ws*	-	-
if	**if**	-
Then	**then**	-
Else	**else**	-
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
$<$	**relop**	LT
$<=$	**relop**	LE
$=$	**relop**	EQ
$<>$	**relop**	NE
$>$	**relop**	GT
$>=$	**relop**	GE



Transition Diagrams

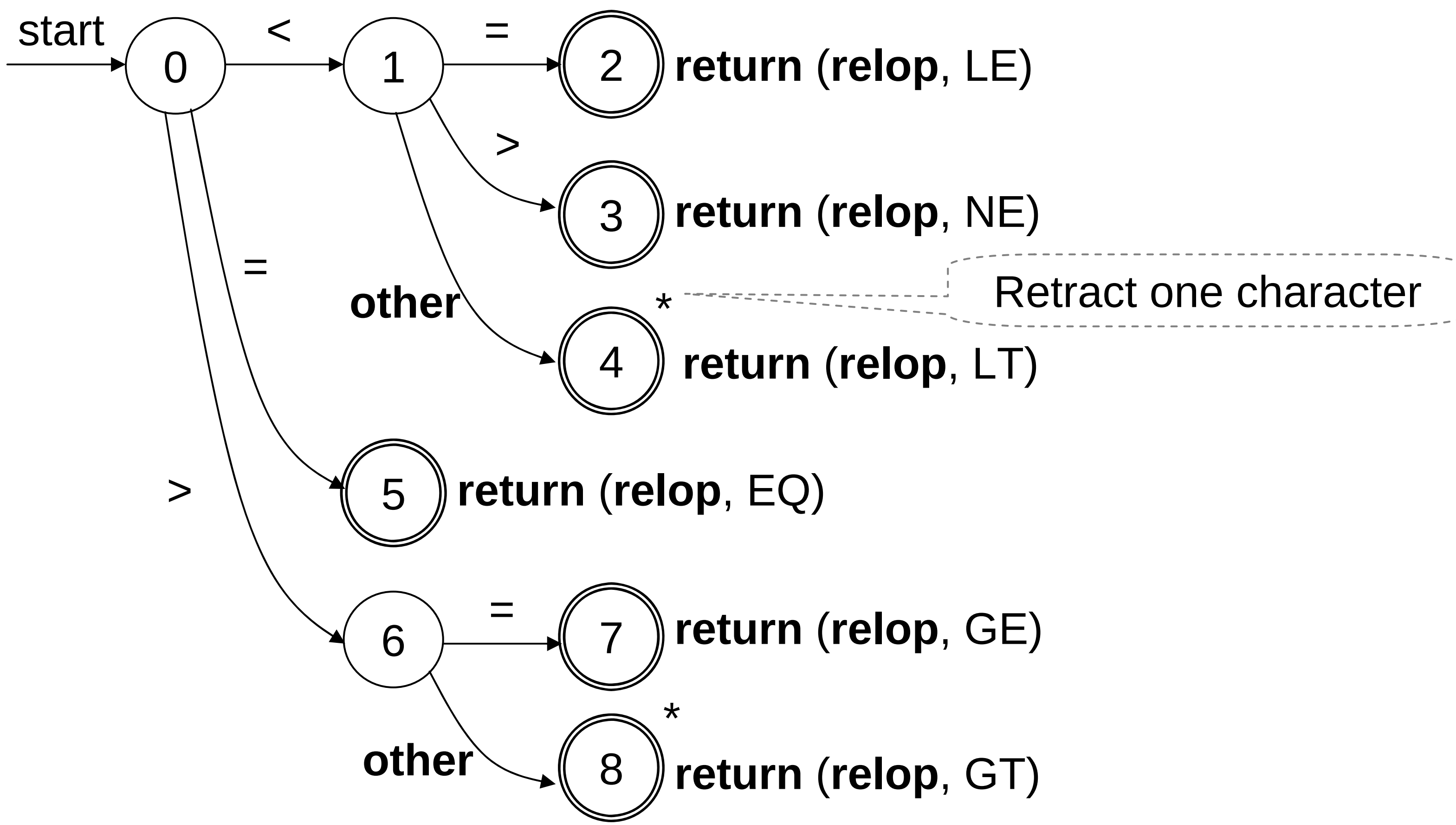
- An intermediate step of a lexical analyzer is to convert patterns into stylized flowcharts called transition diagrams.
  - Transition diagrams have a collection of nodes or circles called states.
    - Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.
    - Each edge is directed from one state to another and is labeled by a symbol or a set of symbols.
  - We assume that each transition diagram is deterministic (at this stage).
    - That is, there is never more than one edge out of a given state with a given symbol among its labels.



Transition Diagrams (Cont.)

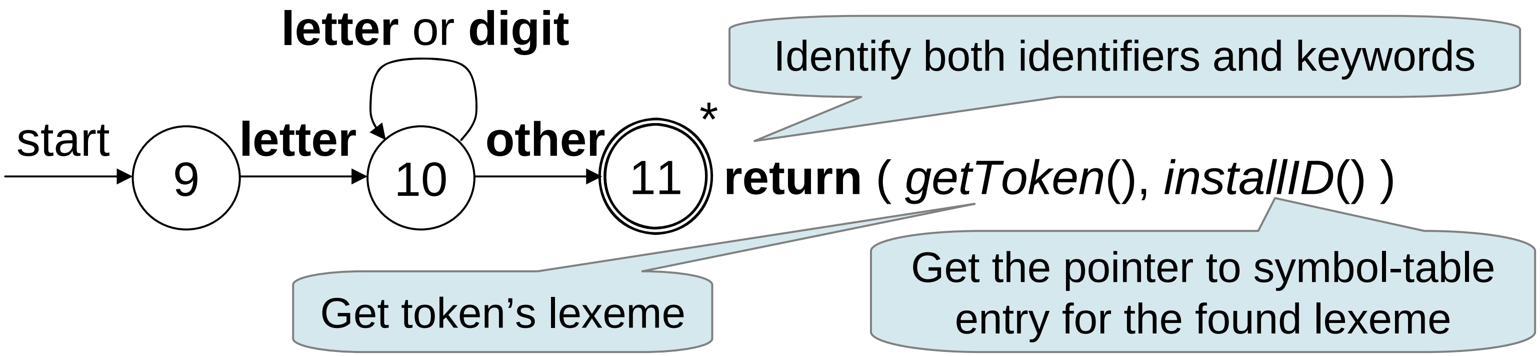
- Conventions of transition diagrams
  - Accepting states indicate that a lexeme has been found. An accepting state is indicated by a double circle.
    - Once the accepting state is reached, a token and attribute value are typically returned to the parser.
  - If it is necessary to retract the *forward* pointer one position, we shall place a * near that accepting state.
  - The start state is indicated by an edge labeled “start”, entering from nowhere.

Transition Diagram for relop

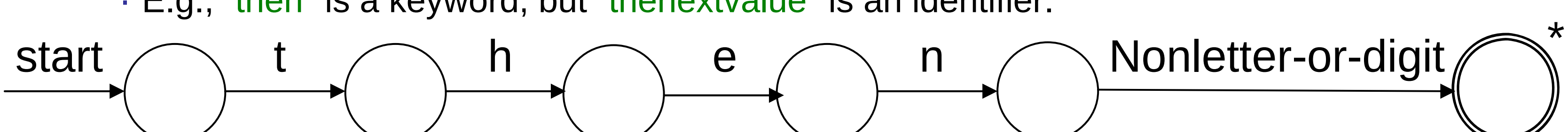


Recognition of Keywords and Identifiers

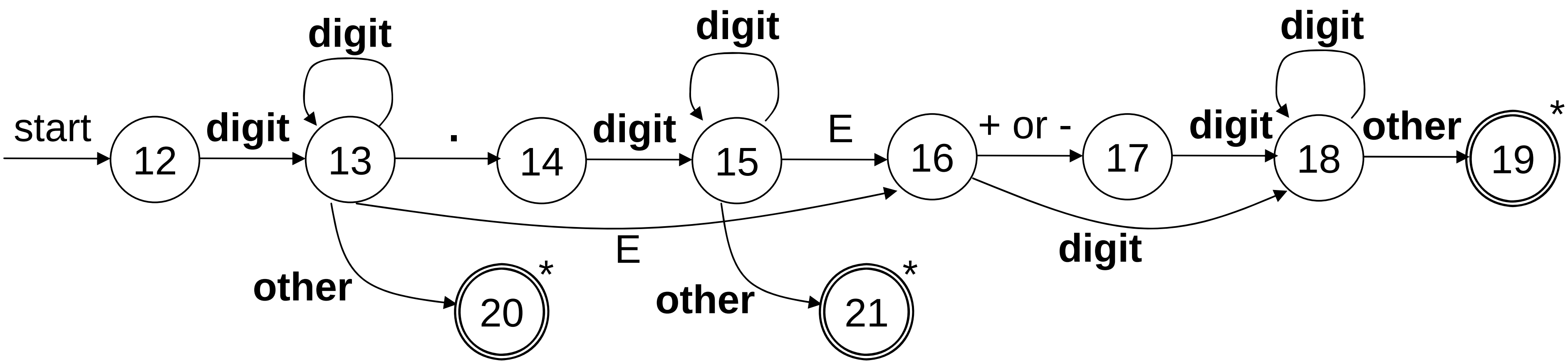
- Keywords are not identifiers, but look like identifiers.



- There are two ways to recognize keywords and identifiers:
  - Install the reserved words in the symbol table initially.
    - A field of the symbol-table entry indicates that these strings are never ordinary identifiers.
  - Create separate transition diagrams for each keyword.
    - No character can be the continuation of an keyword.
      - E.g., "then" is a keyword, but "thenextvalue" is an identifier.

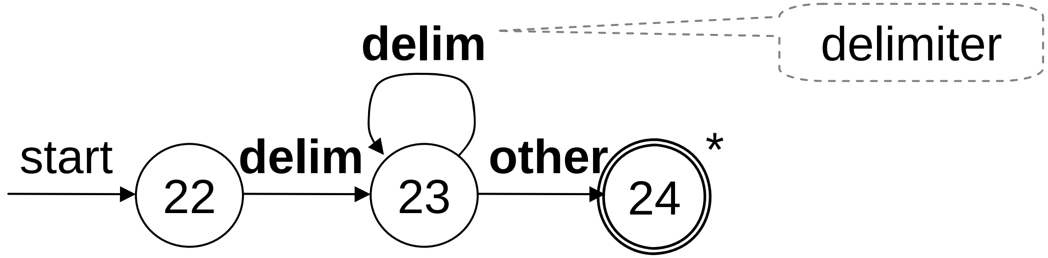


Transition Diagrams for Unsigned Numbers and Whitespace



A Transition diagram for unsigned numbers

digit → [0-9]
digits → *digit*$^+$
number → *digits* (. *digits*)? (E [+-]? *digits*)?

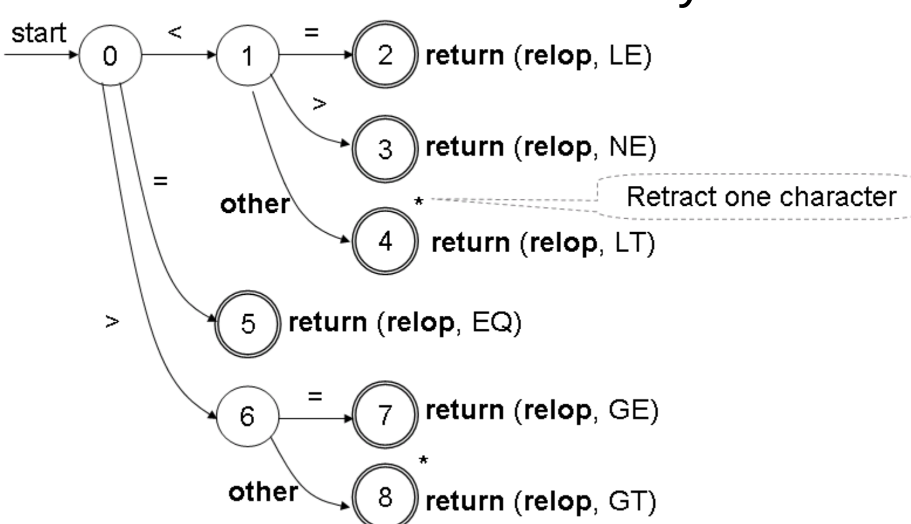


A Transition diagram for whitespace



Transition-Diagram-Based Lexical Analyzer

- A lexical analyzer can be built by a collection of transition diagrams.
  - A switch based on the current state value.
  - The code for a state is a switch statement or multi-way branch.




```
TOKEN* getRelop() {
   TOKEN* retToken = new RELOP; //token object
   while (1) { // repeat until a return or failure occurs
      switch (state) {
         case 0: c = nextChar();
                  if (c == '<') state = 1;
                  else if (c == '=') state = 5;
                  else if (c == '>') state = 6;
                  else fail(); // lexeme is not a relop
                  break;
         case 1: …
         …
         case 8: retract ();
                  retToken.attribute = GT; // attribute
                  return (retToken);
      }
   }
}
```

Reset *forward* pointer to *lexemeBegin*

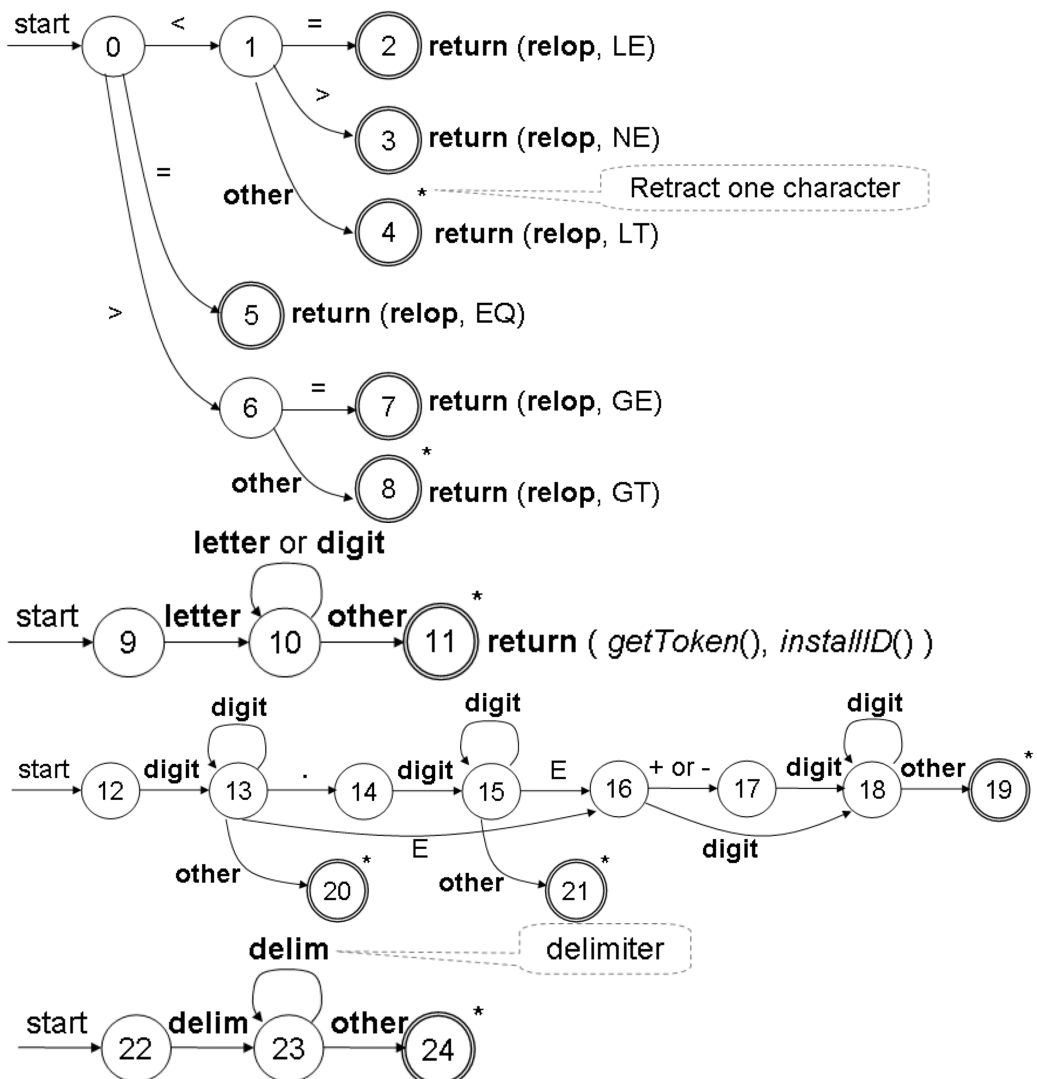
relop transition diagram (with C++)



Transition Diagrams in Lexical Analysis

- There are several ways to recognize tokens through transition diagrams:
  1. Arrange the transition diagrams for each token to be tried sequentially. Then, the function *fail()* resets the pointer *forward* to start the next transition diagram.
     – We should use transition diagrams for keywords before using the transition diagram for identifiers.
  2. Run various transition diagrams in parallel, and take the longest prefix of the input that matches any pattern.
     – This rule allows us to prefer
       – Identifier “thenext” to keyword “then”, or
       – The operator -> to -.
  3. Combine all the transition diagrams into one.
     – The combination is easy if no two tokens start with the same character.
     – In general, the problem of combining transition diagram for several tokens is complex.

The Combined Transition Diagram



Combine states 0, 9, 12, and 22 together

The Lexical-Analyzer Generator Lex

Lexical-Analyzer Generator *Lex*

- Lex (or Flex, fast Lex) allows users to specify regular expressions to describe patterns for tokens.
  - Input notation for the Lex tool is the *Lex language*, and the tool is the *Lex compiler*.
  - The Lex compiler
    - Transforms the input patterns into a transition diagram and
    - Generates code (lex.yy.c) to simulate this transition diagram.

lex.l	→	Lex Compiler (e.g., flex)	→	lex.yy.c
lex.yy.c	→	C compiler (e.g., gcc)	→	a.out (lexical analyzer)
Input stream	→	a.out	→	Sequence of tokens



Structure of Lex Programs

- Declarations section
  - Include declarations of variables, manifest constants (常數清單), e.g., the names of tokens, and regular definitions.
- Translation rules
  - Each rule has the form:

    Pattern { Action }

    - Each pattern is a regular expression that uses the regular definitions in the declaration section.
    - The actions are fragments of code, typically written in C.
- Additional functions
  - Hold whatever additional functions used in the actions.
  - These functions can be compiled separately and loaded with the lexical analyzer.

```
declarations section
%%
translation rules
%%
auxiliary functions
```

Structure of a Lex program

The symbol to separate two sections



Use of Lex

- The lexical analyzer created by ***Lex*** behaves in concert with the parser:
  - When the lexical analyzer is called by the parser, it begins reading its remaining input until it finds the longest prefix matching the pattern P_i.
  - Then the lexical analyzer executes the associated action A_i.
  - Typically, A_i will return to the parser if P_i is not a whitespace or comments.
  - The lexical analyzer returns a single value (i.e., the token name) to the parser, but use the shared integer variable (i.e., yylval) to pass additional information about the lexeme found.



Lex Program for the Tokens

Pass value to the parser

```
%{
   /* definitions of manifest constants
   LT, LE, EQ, NE, GT, GE,
   IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim     [ \t\n]
ws        {delim}+
letter     [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return{THEN);}
else      {return{ELSE);}
```

space

dot

Regular expression

```
{id}        {yylval = (int) installID(); return(ID);}
{number} {yylval = (int) installNum();
           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 installID()  { /* function to install the lexeme,
                    whose first character is pointed
                    by yytext, and whose length is
                    yyleng, into symbol table and
                    return a pointer */ }
int installNum() {/* similar to installID, but puts
                    numerical constants into a
                    separate table */ }
```



Lex Program for the Tokens (Cont.)

- Declarations section
  - Anything between %{ and %} is copied directly to the file lex.yy.c directly.
    - The manifest constants are usually defined by C #define to associate unique integer code.
  - A sequence of regular definitions
    - Curly braces { } are to surround the used regular definitions.
    - Parentheses ( ) are grouping metasymbols and don't stand for themselves.
    - \. Represents the dot, since . is a metasymbol representing any character.
- Auxiliary function
  - Everything in the auxiliary section is copied directly to file lex.yy.c, but may be used in the actions.



Lex Program for the Tokens (Cont.)

- Translation rules
  - The action taken when *id* is matched is threefold:
    - Function installID() is called to place the lexeme found in the symbol table.
    - This function returns a pointer to the symbol table, placed in global variable yylval, which is used by the parser.
      - This function has two variables that are set automatically:
        - » yytext is a pointer to the beginning of the lexeme.
        - » yyleng is the length of the lexeme found.
    - The token name ID is return to the parser.



Conflict Resolution in Lex

- When several prefixes of the input match one or more patterns, Lex
  - Always prefers a longer prefix.
  - Always prefers the pattern listed first in the program if the longest possible prefix matches two or more patterns.
- E.g.,
  - <= is a single lexeme instead of two lexemes.
  - The lexeme then is determined as the keyword **then**.



The Lookahead Operator

- The lookahead operator in Lex
  - Automatically reads one character ahead of the last character that forms the selected lexeme, and
  - Retracts the input when the lexeme is consumed from the input.
- Sometimes we want a certain pattern to be matched to the input when it is followed by a certain other characters.
  - We use the *slash* in a pattern to indicate the end of the pattern that matches the lexeme.
  - What follows / is additional pattern that must be matched before we can decide.
  - E.g., a Fortran statement: IF(I,J) = 3 (IF is the name of an array)
    a Fortran statement: IF(A<(B+C)*D)THEN… (IF is a keyword)
    - » The keyword IF always followed by a left parenthesis, a right parenthesis, and a letter. We can write a Lex rule for the keyword IF like

```
IF / \( .* \) {letter}
```

.* means any string without a newline



An Flex Program to Print the File Content

Declarations section

```
/*** Definition section ***/
%{
/* C code to be copied verbatim */
#include <stdio.h>
%}

/* This tells flex to read only one input file */
%option noyywrap

```

Translation rules

```
%%
    /*** Rules section ***/
.|\n       {   printf("%s",yytext); }

```

Auxiliary functions

```
%%
/*** C Code section ***/
int main(void) {
    /* Call the lexer, then quit. */
    yylex();
    return 0;
}
```



Finite Automata



Finite Automata

- Finite automata formulation is the heart of the transition from the input program into a lexical analyzer.
  - Finite automata are recognizers that say "yes" or "no" about each possible input string.
  - Finite automata consist of two forms:
    - Nondeterministic finite automata (NFA):
      - No restrictions on the labels of their edges
        - » A symbol can label several edges out of a state.
        - » ε is a possible label.
    - Deterministic finite automata (DFA):
      - Exactly one edge with that symbol leaving that state
  - NFA and DFA are usually represented by a transition graph.

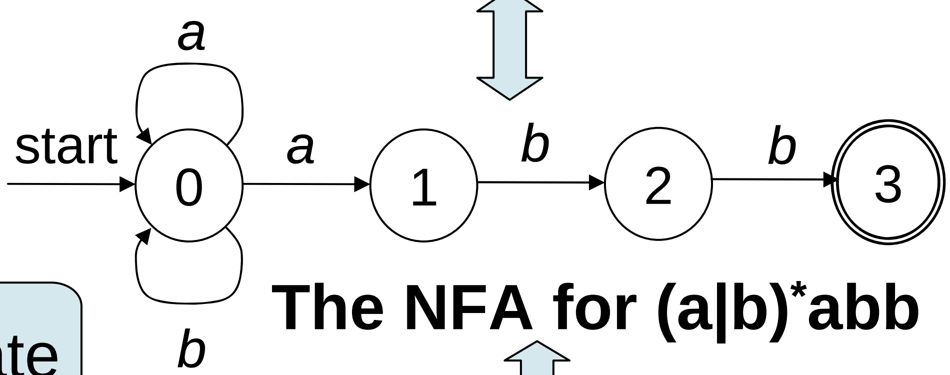


Nondeterministic Finite Automata (NFD)

- An NFA consists of
  - A finite set of states S.
  - A set of input symbols $\sum$ (the input alphabet), excluding ε.
  - A transition function that gives a set of next states for each state among the symbols in $\sum \cup \{\varepsilon\}$.
  - A state s_0 from S as the start state (or initial state).
  - A subest F of S is distinguished as the accepting states (or final states).
- The transition graph to represent NFA
  - Nodes are states
  - Labeled edges represent the transition functions.
- The transition graph for NFA is similar to a transition diagram, except:
  - A symbol can label edges of one state to several states.
  - An edge could be labeled by ε.

Nondeterministic Finite Automata (NFD) (Cont.)

Regular expression: $(a|b)^*abb$



The NFA for $(a|b)^*abb$

The advantage of transition table is easy to find the transitions on a given state and input.

The disadvantage of transition table takes a lot of space when the input alphabet is large.

state

STATE	a	b	ε
0	{0, 1}	{0}	ø
1	ø	{2}	ø
2	ø	{3}	ø
3	ø	ø	ø

Input symbol

Value of the transition function with the given state and input symbol

Transition table for the NFA

Acceptance of Input Strings by Automata

- An NFA *accepts* input string x if and only if there is some path in the transition graph from the start state to one of the accepting states, such that the symbols along the path spell out x.

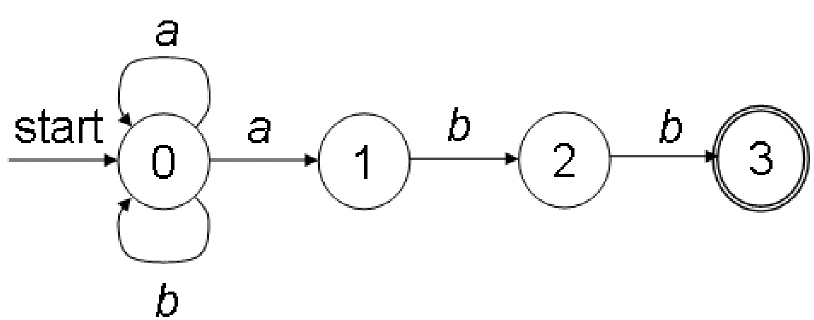
aabb

$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

Accepting path

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

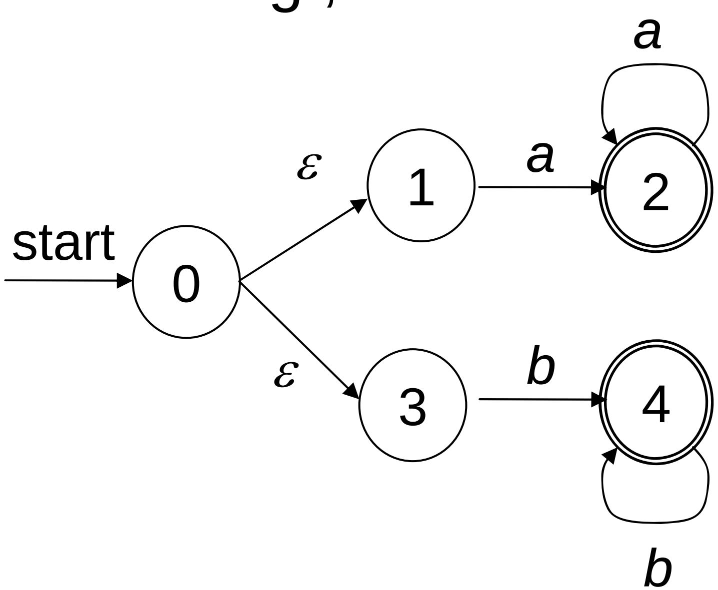
Not accepting path



The NFA for $(a|b)^*abb$

Acceptance of Input Strings by Automata (Cont.)

- We use *L(A)* to stand for the language accepted by automation *A*.
  - E.g.,



The NFA accepting *L(aa*|bb*)*

String *aaa* is accepted

$$0 \xrightarrow{\varepsilon} 1 \xrightarrow{a} 2 \xrightarrow{a} 2 \xrightarrow{a} 2$$

ε is disappeared in a concatenation



Deterministic Finite Automata (DFA)

- DFA is a special case of an NFA where
  - There are no moves on input ε, and
  - For each state *s* and input symbol *a*, there is exactly one edge labeled *a* out of *s*. (No curly brace is needed in the entries of the transition table.)
- If we use a transition table to represent a DFA, each entry is a single state. (In practice, we usually adopt DFA instead of NFA for simplicity).
- DFA is a simple, concrete algorithm for recognizing strings, while NFA is an abstract algorithm for recognizing strings.
- Each NFA can be converted to a DFA accepting the same language.



Simulating a DFA

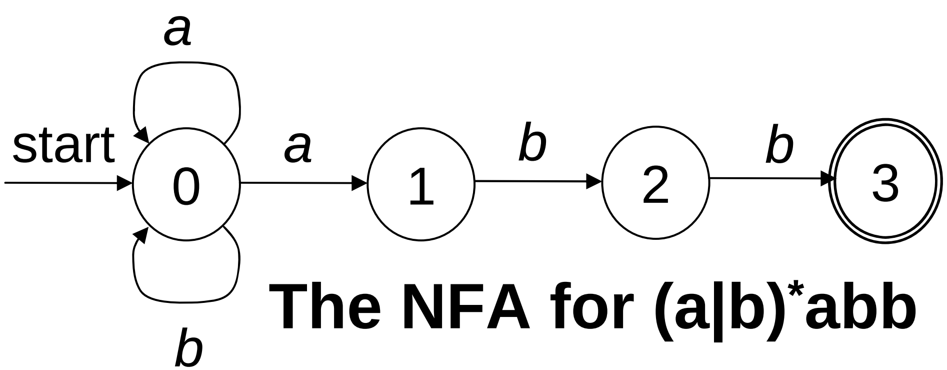
- **Algorithm:**
  - **Simulating a DFA**
- **INPUT**:
  - An input string *x* terminated by an end-of-file character **eof**. A DFA *D* with start state s_0, accepting states *F*, and transition function *move*.
- **OUTPUT**:
  - Answer "yes" if D accepts x; "no" otherwise.
- **METHOD**:

```
s = s0;
c = nextChar();
while (c != eof) {
    s = move (s,c);
    c = nextChar ();
}
if (s is in F) return "yes";
else return "no";
```

Return the next character of the input string *x*.

Move to the next state from state *s* through edge *c.*

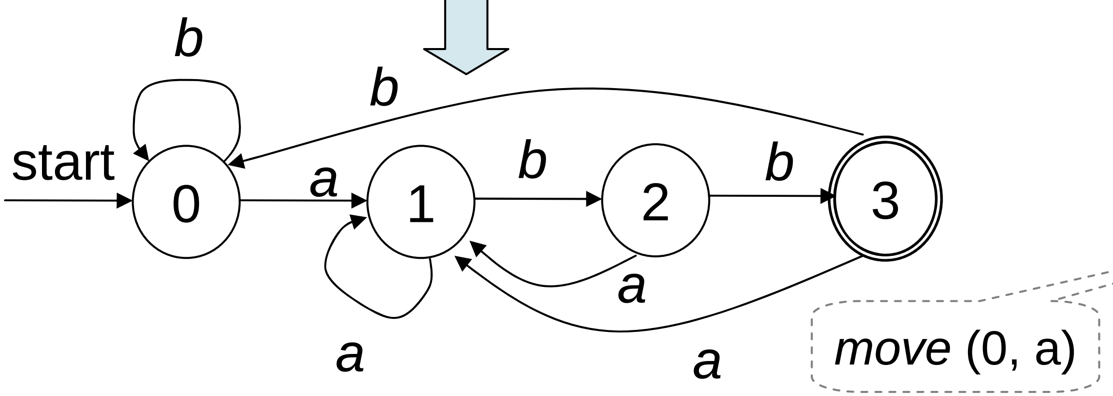
From NFA to DFA that Accepts (a|b)*abb



The NFA for (a|b)*abb

STATE	a	b	ε
0	{0, 1}	{0}	ø
1	ø	{2}	ø
2	ø	{3}	ø
3	ø	ø	ø

Transition table for the NFA



The DFA for (a|b)*abb

STATE	a	b	ε
0	1	0	ø
1	1	2	ø
2	1	3	ø
3	1	0	ø

Transition table for the DFA



From Regular Expressions to Automata



Subset Construction

- Subset construction is the technique to convert an NFA to a DFA.
  - Each state of the constructed DFA corresponds to a set of NFA states.
  - After reading input $a_1a_2...a_n$, the DFA is in that state which corresponds to the set of states that the NFA can reach (from its start state, following paths labeled $a_1a_2...a_n$).
- The number of DFA states could be exponential in the number of NFA states.
  - The maximal number of DFA states is 2^n where n is the number of states in NFA.
  - In pratice, the NFA and DFA have approximately the same number of states.



Subset Construction (Cont.)

- **Algorithm:**
  - **The *subset construction* of a DFA from an NFA**
- **INPUT**:
  - An NFA *N.*
- **OUTPUT**:
  - A DFA *D* accepting the same language as *N.*
- **METHOD**:
  - Construct a transition table ***Dtran*** for *D* so that *D* will simulate "in parallel" all possible moves that *N* can make on a given input string. (Note that the ε-transition problem of *N* should be solved.)



Subset Construction (Cont.)

- Operations on NFA states:
  - *s* is a single state of *N*, while *T* is a set of states of *N*.

Operation	Description
ε-*closure (s)*	Set of NFA states reachable from NFA state *s* on ε-transition alone.
ε-*closure (T)*	Set of NFA states reachable from some NFA state *s* in set *T* on ε-transition alone. $\varepsilon\text{-}closure\ (T) = U_{s\ in\ T}\ \varepsilon\text{-}closure\ (s)$
move (T, a)	Set of NFA states to which there is a transition on input symbol *a* from some state *s* in *T*.



Subset Construction (Cont.)

- The process of the subset construction:
  - Before reading the first input symbol, *N* can be in any of the states of ε*-closure* (s_0), where s_0 is the start state.
    - Suppose that *N* can be in set of states ***T*** after reading input string *x*. If it next reads input ***a***, then N can immediately go to any of the states in ***move (T, a)***.
    - After reading *a*, it may make several ε-transitions; thus, *N* could be in any state of ***ε-closure (move (T, a))***.
  - The start state of *D* is ε*-closure* (s_0)*.*
  - The accepting states of *D* are all those sets of *N*'s states that include at least one accepting state of *N*.



Algorithm of Subset Construction

- Structure definition:
  - *Dstates* is the set of D's states.
  - *Dtran* is the transition table.
  - *move(T, a)* is the transition function

The complexity to process a symbol is O(n+m).
n: the number of states in NFA
m: the number of transitions in NFA

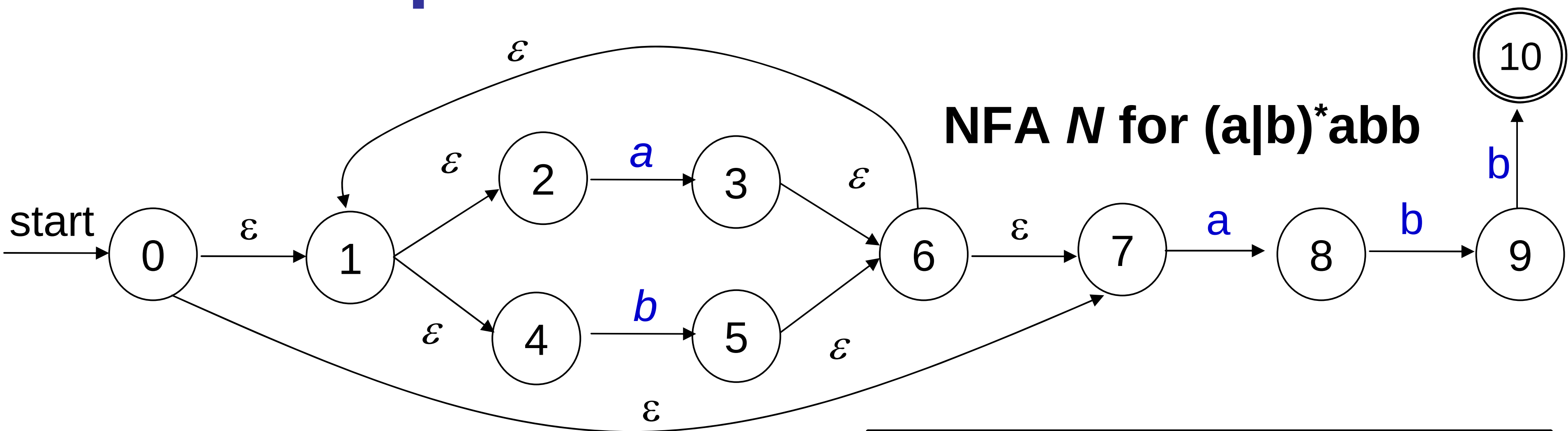
Initially, ε-*closure* (s_0) is the only state in *Dstates*, and it is unmarked.
while (there is an unmarked state *T* in *Dstates*) {
  mark *T*;
  for (each input symbol *a*) {
    U = ε-*closure (move (T, a))*
    if (U is not in Dstates)
      add U as an unmarked state to *Dstates*;
    Dtran[T, a] = U;
  }
}

Subset construction

Computing ε-*closure(T)*

push all states of *T* onto *stack*; // ***T = move(T, a)***
initialize ε-*closure(T)* to *T*;
while (*stack* is not empty) {
  pop the top element ***t*** off *stack*;
  for (each state *u* with an edge from *t* to *u* labeled ε)
    if (*u* is not in ε-*closure(T)*) {
      add *u* to ε-*closure(T)*;
      push *u* onto *stack*;
    }
}

An Example of Subset Construction



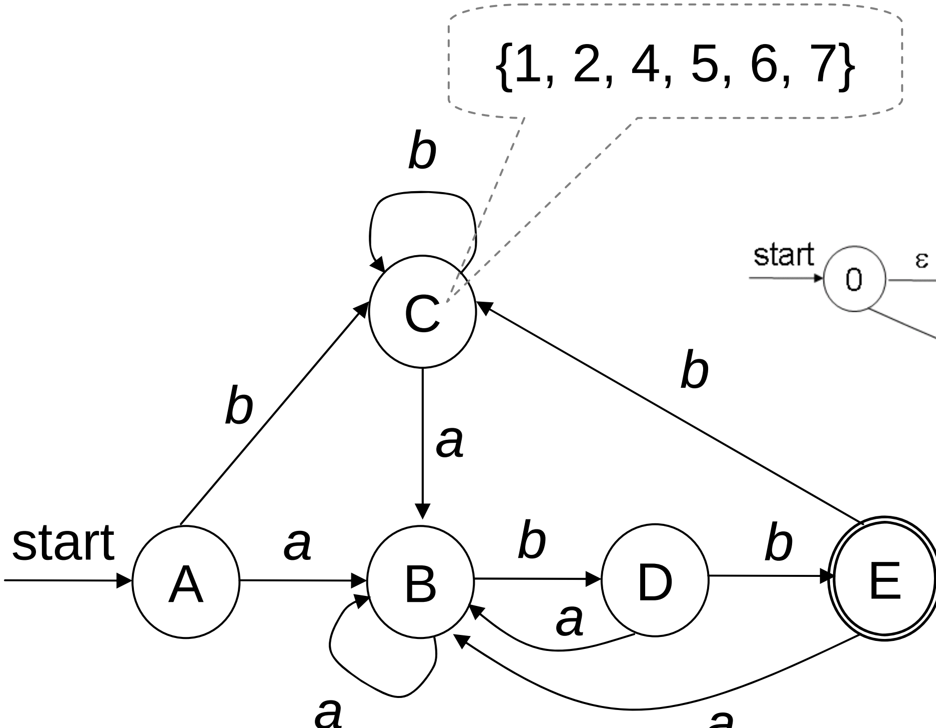
- Alphabet is {a, b}
- The start state *A* = *ε-closure(0)* = {0, 1, 2, 4, 7}
- *Dtran[A, a]* = *ε-closure(move(A, a))*
  = *ε-closure*({3, 8})
  = {1, 2, 3, 4, 6, 7, 8} = B
- *Dtran[A, b]* = *ε-closure(move(A, b))*
  = *ε-closure*({5})
  = {1, 2, 4, 5, 6, 7} = C

NFA State	DFA State	a	b
{0, 1, 2, 4, 7}	A	B	C
{1, 2, 3, 4, 6, 7, 8}	B	B	D
{1, 2, 4, 5, 6, 7}	C	B	C
{1, 2, 4, 5, 6, 7, 9}	D	B	E
{1, 2, 4, 5, 6, 7, 10}	E	B	C

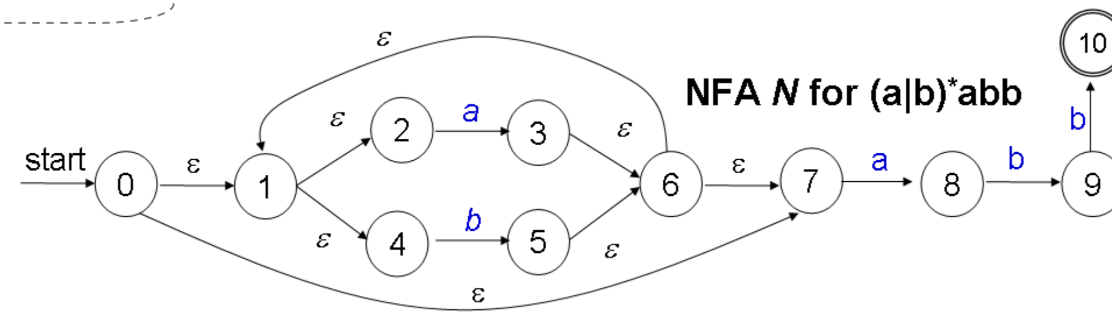
***DTran* for DFA D**



An Example of Subset Construction (Cont.)



Result of subset construction by converting the NFA *N* for (a|b)*abb



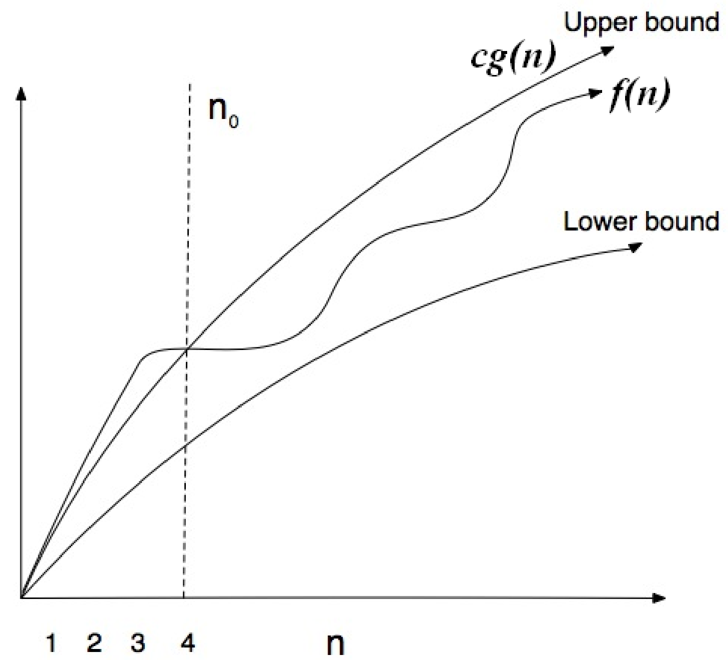
NFA State	DFA State	a	b
{0, 1, 2, 4, 7}	A	B	C
{1, 2, 3, 4, 6, 7, 8}	B	B	D
{1, 2, 4, 5, 6, 7}	C	B	C
{1, 2, 4, 5, 6, 7, 9}	D	B	E
{1, 2, 4, 5, 6, 7, 10}	E	B	C

***DTran* for DFA D**



Big-Oh: O()

- Definition
  - Given a $f(n)$, the running time is $O(g(n))$ if there are some constants c and n_0, such that $f(n) \leq cg(n)$ whenever $n \geq n_0$.
- Example:
  - O(n): at most some constant times n
  - O(1): some constant





Simulation of an NFA

- **Algorithm:**
  - **Simulating an NFA**
- **INPUT**:
  - An input string *x* terminated by an end-of-file character **eof**.
  - An NFA *N* with start state s_0, accepting states *F*, and transition function *move*.
- **OUTPUT**:
  - Answer “yes” if *N* accepts *x*; “no” otherwise.
- **METHOD**:
  - Keep a set of current states ***S*** that are reached from s_0 following the path labeled by the inputs read so far.

$S = \varepsilon\text{-}closure(s_0)$;
$c = nextChar()$;
while (c != **eof**) {
    $S = \varepsilon\text{-}closure(move(S,c))$;
    $c = nextChar()$;
}
if ($S \cap F \neq \emptyset$) **return** “yes”;
else return “no”;



Efficiency of NFA Simulation

```
1) S = ε-closure(s0);
2) c = nextChar();
3) while (c != eof) {
4)     S = ε-closure(move(S,c));
5)     c = nextChar();
6) }
7) if (S∩F != ø) return "yes";
8) else return "no";
```

- The data structures we need are:
  - Two stacks:
    - *oldStates* holds the current set of states (*S* on the right side of line (4))
    - *newStates* holds the next set of states (*S* on the left side of line(4))
  - A boolean array
    - *alreadyOn* indexed by the NFA states is to indicate which states are in *newStates*.
    - Array more efficient to search for a given state.
  - A two-dimensional array
    - *move[s, a]* holds the transition table of the NFA. Each entry of this table points to a set of states and is represented by a linked list.



Efficiency of NFA Simulation (Cont.)

- Transition graph: *n* states with *m* edges (or transitions)
- Initialization:
  - Set each entry of *alreayOn* to FALSE.
  - Put each state *s* in *ε-closure(s_0)* to the *oldStage*.

The complexity to process a character is O(n+m)

```
1) S = ε-closure(s0);
2) c = nextChar();
3) while (c != eof) {
4)     S = ε-closure(move(S,c));
5)     c = nextChar();
6) }
7) if (S ∩ F != ø) return "yes";
8) else return "no";
```

```
 9) addState(s) {
10)     push s onto newStates;
11)     alreadyOn[s] = TRUE;
12)     for ( t on move[s, ε] )
13)         if ( !alreadyOn(t) )
14)             addState(t);
15) }
```

Compute ε-closure(s)

At most *m* times over *n* calls

```
16) for ( s on oldStates) {
17)     for (t on move[s, c] )
18)         if ( !alreadyOn[t])
19)             addState(t);
20)     pop s from oldStates;
21) }
22) for ( s on newStates) {
23)     pop s from newStates;
24)     push s onto oldStates;
25)     alreadyOn[s] = FALSE;
21) }
```

At most *n* times

At most *m* times in total

At most called for *n* times in total

Implementation of line (4)

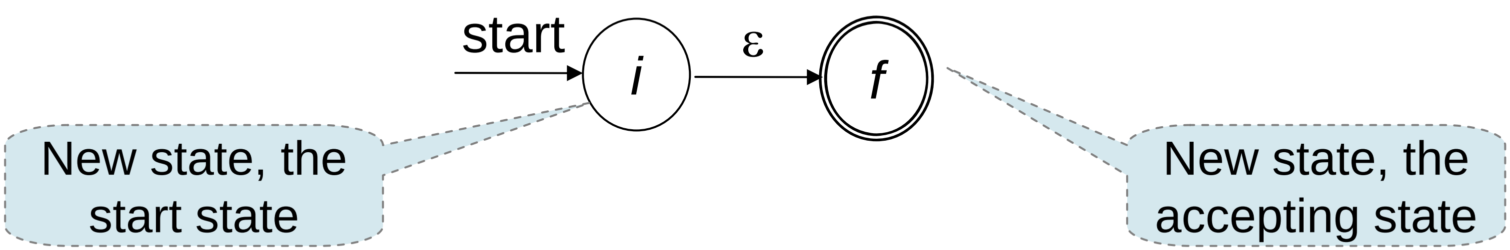


Construction of an NFA from a Regular Expression

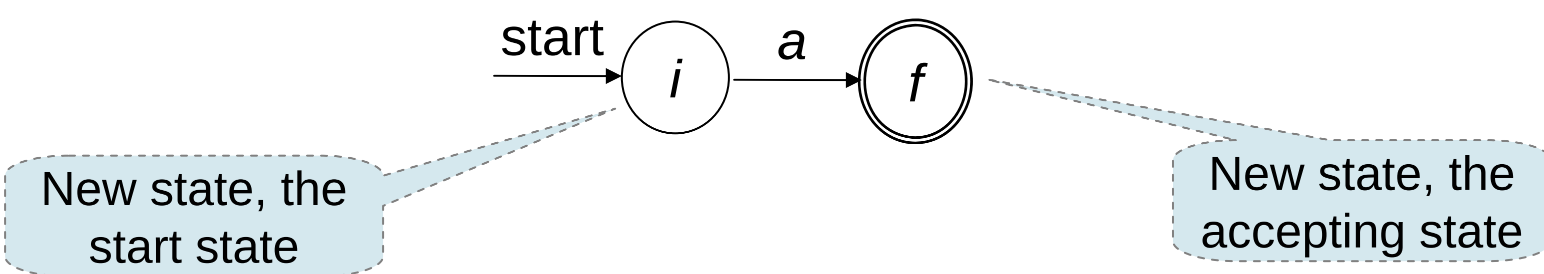
- **Algorithm**:
  - The **McNaughton-Yamada-Thompson** algorithm to convert a regular expression to an NFA
- **INPUT**:
  - A regular expression *r* over alphabet ∑.
- **OUTPUT**:
  - An NFA *N* accepting *L(r)*.
- **METHOD**:
  - Begin by parsing *r* into its constituent subxpressions with **basis rules** and **inductive rules**.
    - **2 basis rules**: handle subexpressions with no operators
    - **4 inductive rules**: construct larger NFAs from the NFAs for the subexpression of a given expression.

Basis Rules

- For expression ε, construct the NFA

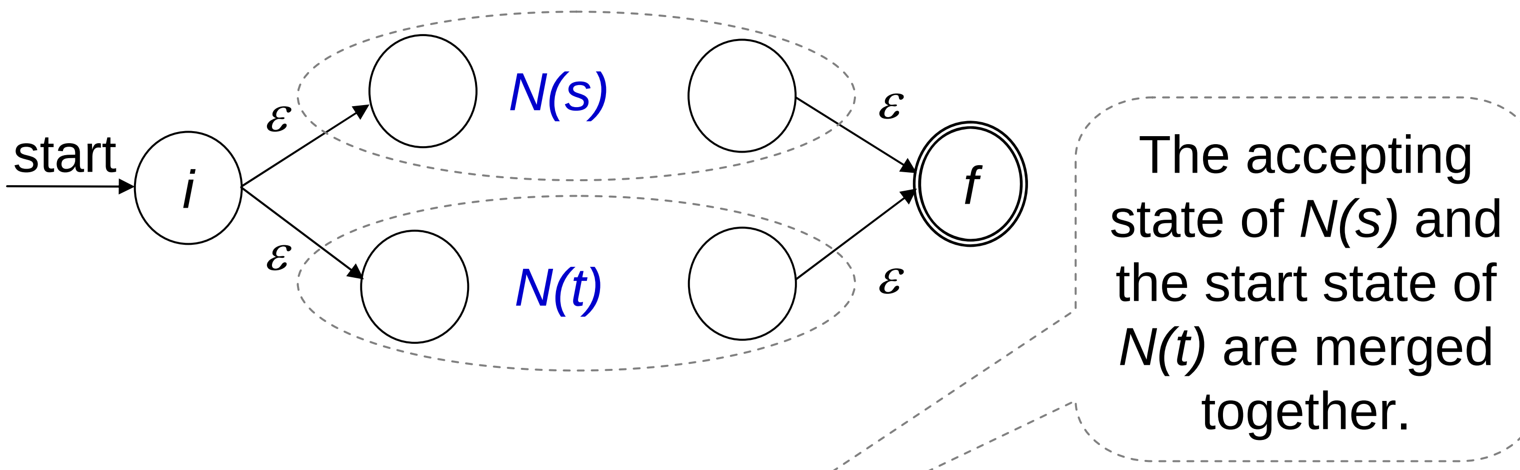


- For any subexpression *a* in ∑, constuct the NFA

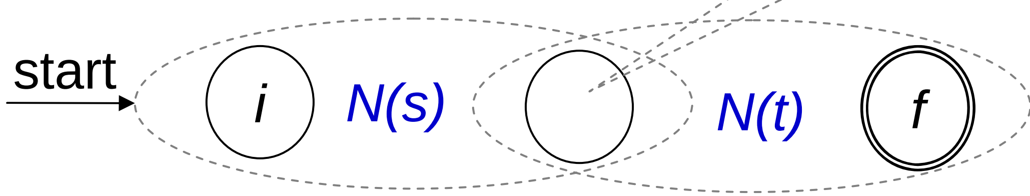


Induction Rules

- Suppose *N(s)* and *N(t)* are NFAs for regular expressions *s* and *t* that denote languages *L(s)* and *L(t)*, respectively.
  - Suppose *r* = *s* | *t*. *N(r)* accepts *L(s)* U *L(t)*, and is an NFA for *r*=*s* | *t*.

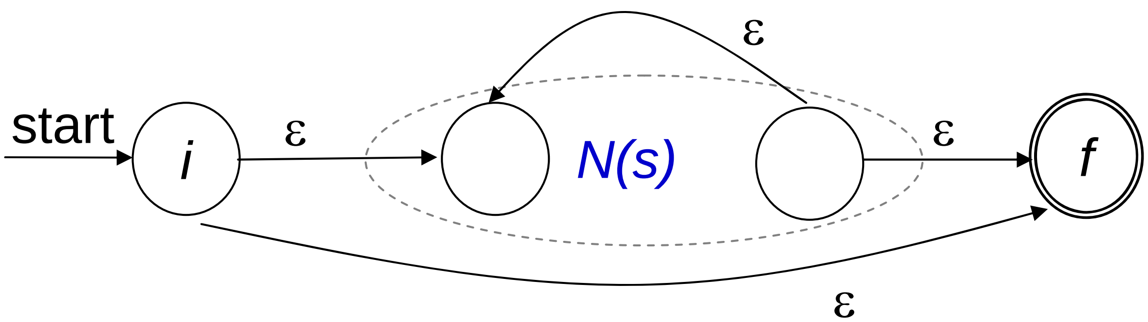


  - Suppose *r* = *st*. *N(r)* accepts *L(s)L(t)*, and is an NFA for *r*=*st*.



Induction Rules (Cont.)

– Suppose $r = s^*$. $N(r)$ accepts $L(s^*)$, and is an NFA for $r=s^*$.



– Suppose $r = (s)$. Then $L(r) = L(s)$, and therefore $N(s) = N(r)$.

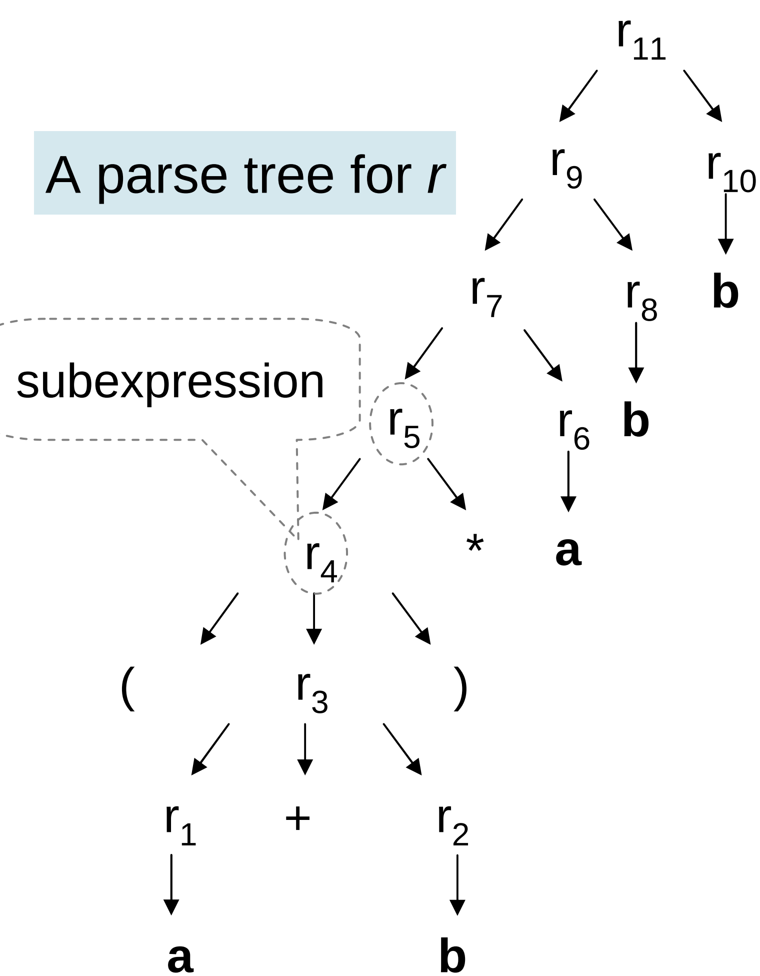


Properties of the **McNaughton-Yamada-Thompson** Algorithm

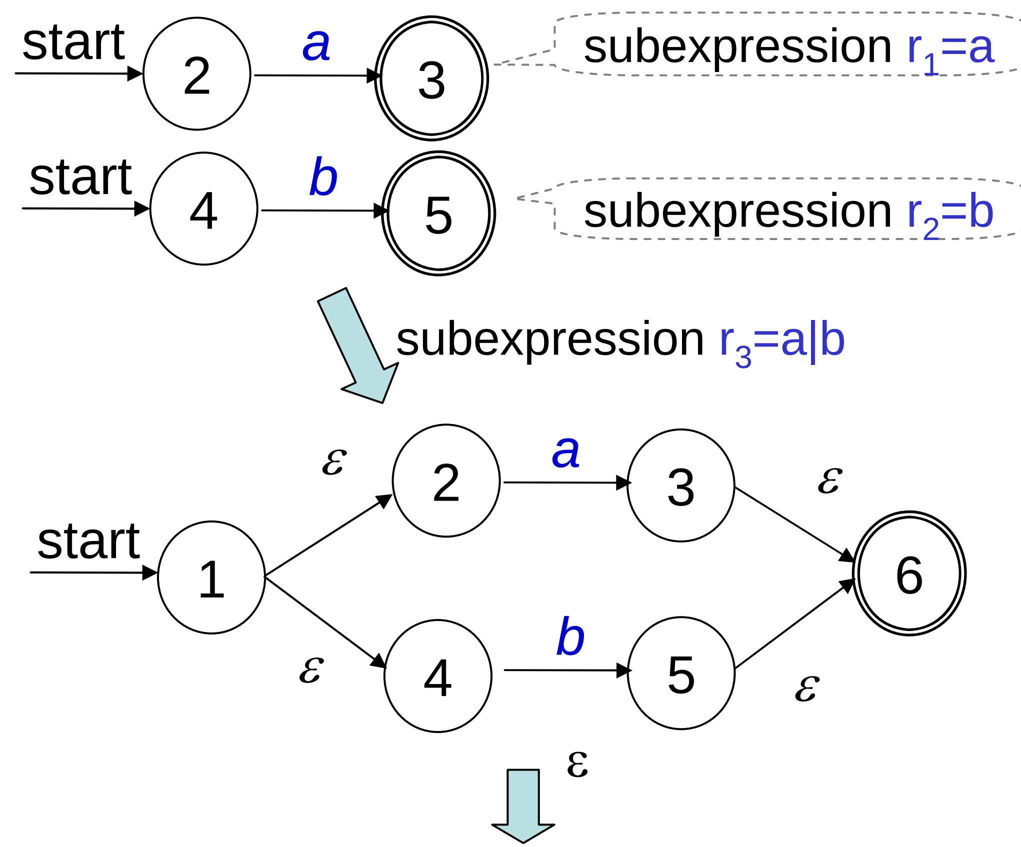
- *N(r)* has at most twice as many states as there are operators and operands in *r*.
  - Each step of the algorithm creates at most two new states.
- *N(r)* has one start state and one accepting state.
  - The accepting state has no outgoing transitions.
  - The start state has no incoming transitions.
- Each state of *N(r)* other than the accepting state has
  - Either one outing transition on a symbol in $\sum$
  - Or two outgoing ε-transitions.

NFA Construction with McNaughton-Yamada-Thompson (MYT) Algorithm

A parse tree for *r*

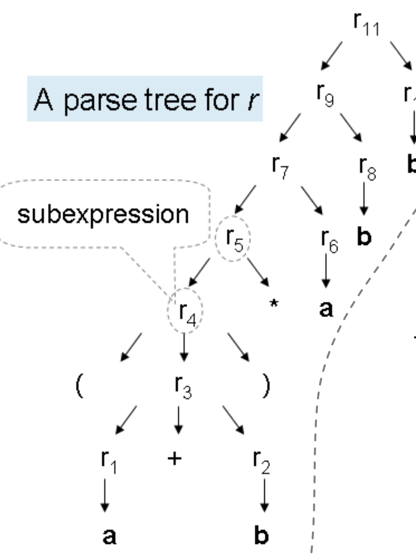


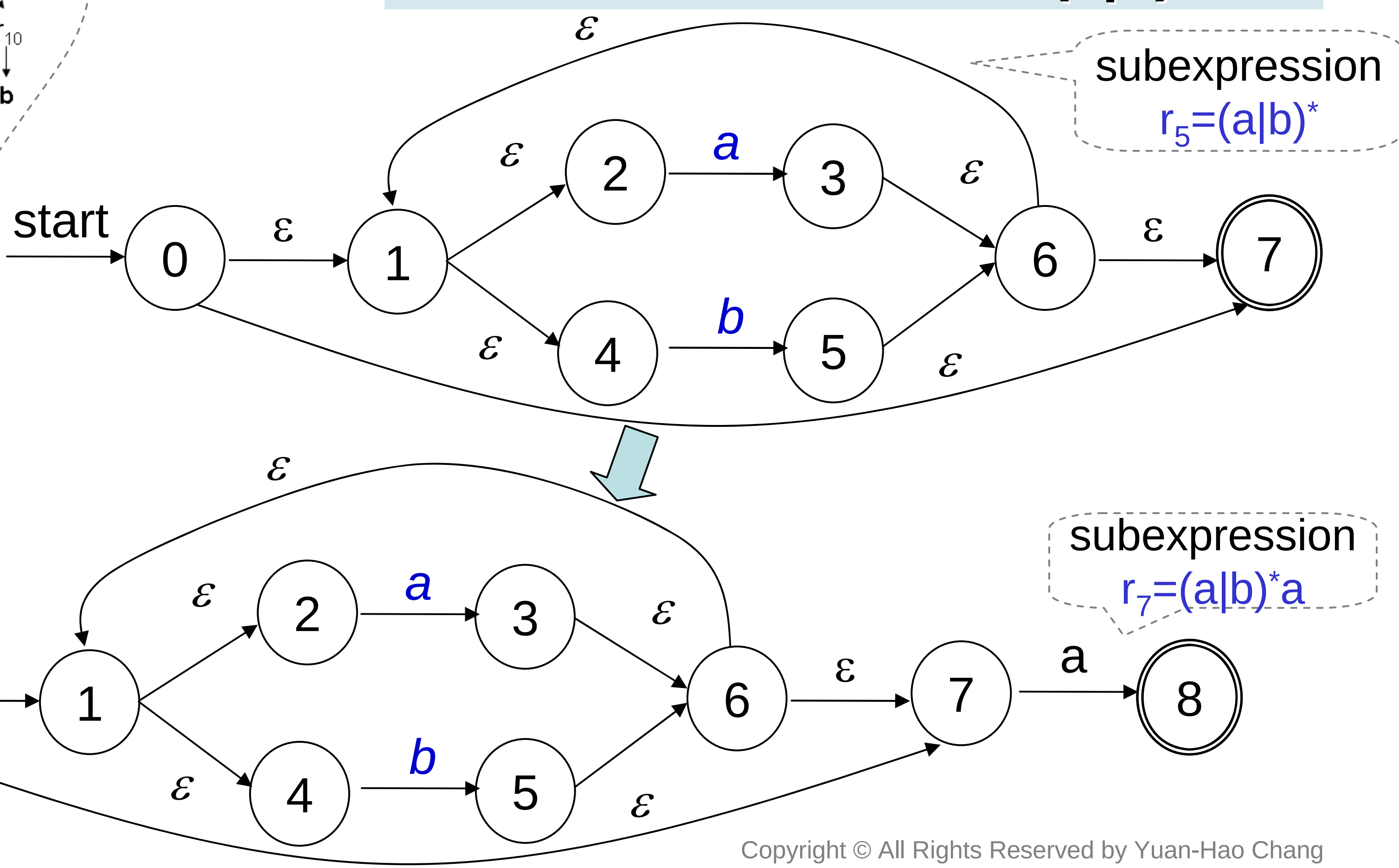
Construct an NFA for **r = (a|b)*abb**



NFA Construction with McNaughton-Yamada-Thompson (MYT) Algorithm (Cont.)

Construct an NFA for $\mathbf{r = (a|b)^*abb}$

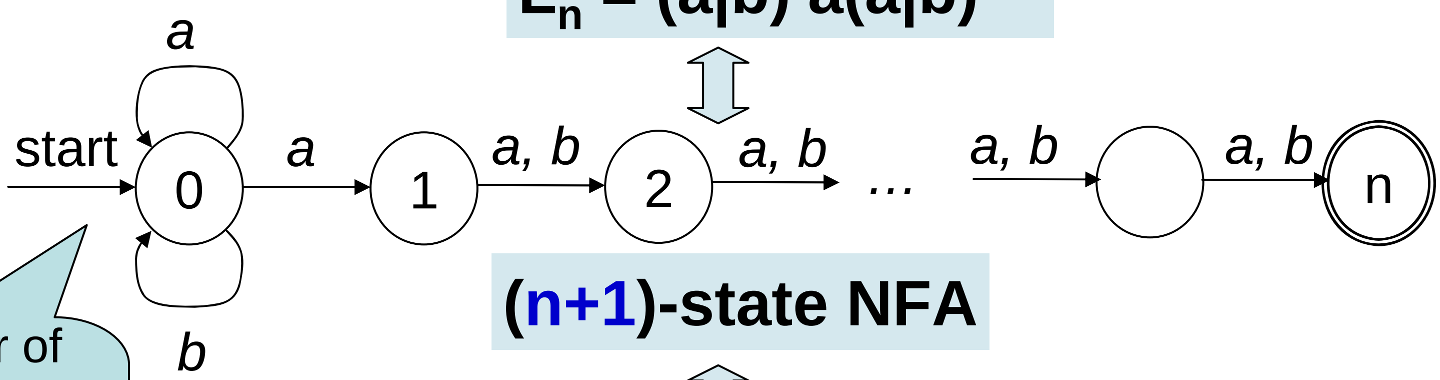






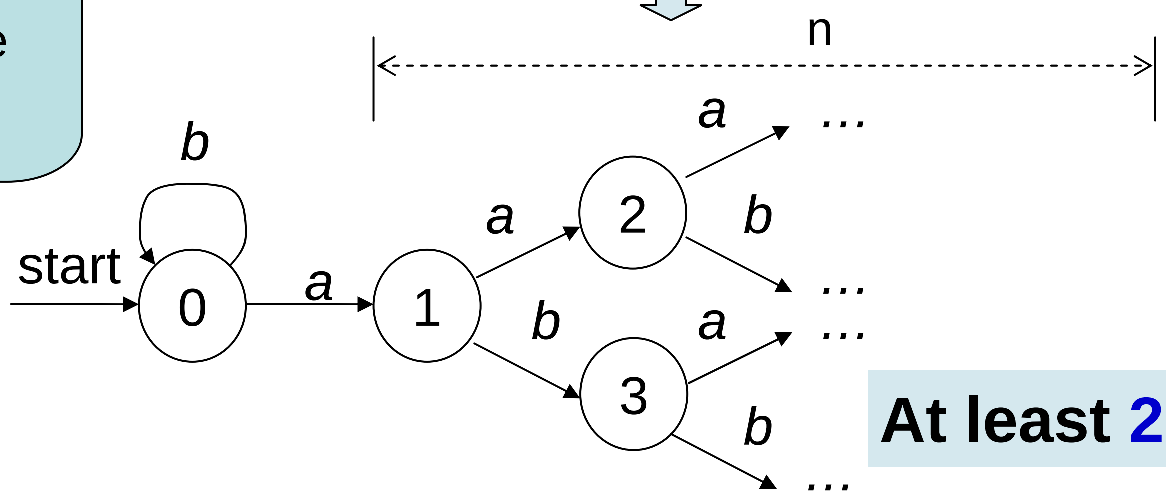
Extreme Case of Regular Expression

$L_n = (a|b)^*a(a|b)^{n-1}$



(n+1)-state NFA

The number of states is reduced by some state minimization algorithm



At least 2^n-state DFA

Complexity of NFA and DFA

- The NFA for a regular expression *r* consists of at most 2|r| states and 4|r| transitions.
  - $n \le 2|r|$
  - $m \le 4|r|$
- The time to recognize string *x* with NFA is |*x*| times the size of the NFA's transition graph, i.e, O((n+m)|*x*|) = O(|r|×|x|).
- Initial time for a DFA = Initial time for an NFA + time for subset construction
  - The key step U=ε-closure(move(T,a)) in the subset construction takes O(n+m) = O(|r|) to construct a set of states U from a set of states T.
  - There are at most *r* symbols in the regular expression *r*.
  - There are *s* states in the DFA

|*x*| = the size of input string *x* = the length of *x*

|*r*| = the size of *r*
= # of operators in r + # of operands in r

The time to construct the parse tree is O(|r|)

Avg. case: $s \approx r$
Worst case: $s \approx 2^{|r|}$

Automation	Initial time	Time to recognize string *x*
NFA	O(|*r*|)	O(|*r*|×|*x*|)
DFA	O($\|r\|^2 s$)	O(|*x*|)
DFA typical case	O($\|r\|^3$)	O(|*x*|)
DFA worst case	O($\|r\|^2 2^{\|r\|}$)	O(|*x*|)

Time for subset construction = O(|*r*|×|*r*|×*s*)

Time for one symbol r symbols s states



NFA or DFA

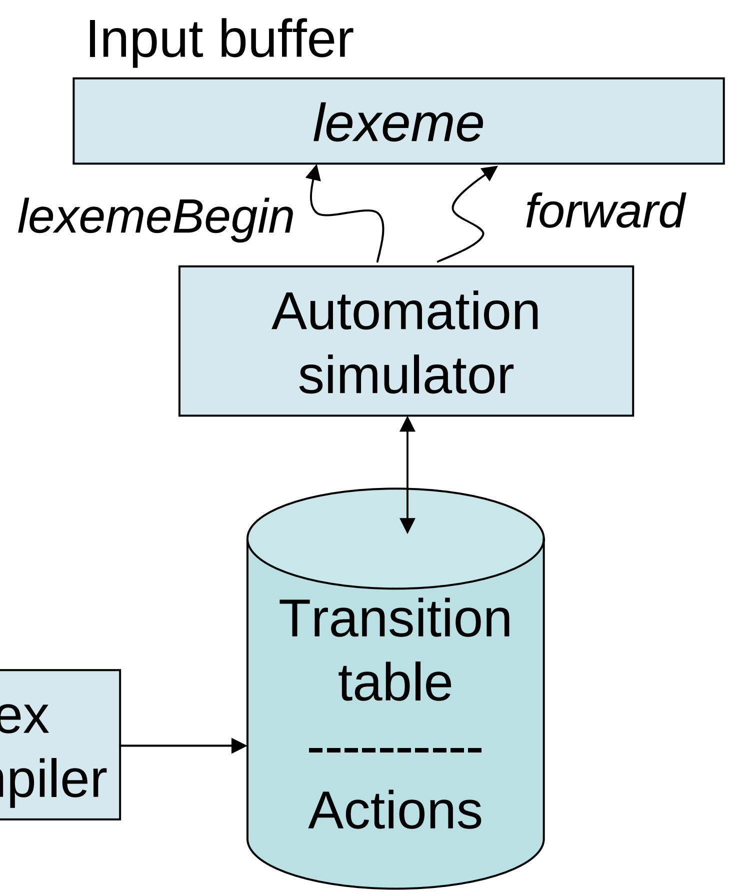
- Choose to convert a regular expression to an NFA or DFA.
  - Convert to an NFA when the regular expression is used for several times.
    - E.g., the **grep** command: Users specify one regular expression to search one or several files for one pattern.
  - Convert to an NFA when the transition table of DFA is too large to fit in main memory.
  - Convert to a DFA when the regular expression is used frequently.
    - E.g., a lexical analyzer that uses each specified regular expression for many times to search the patterns of tokens.

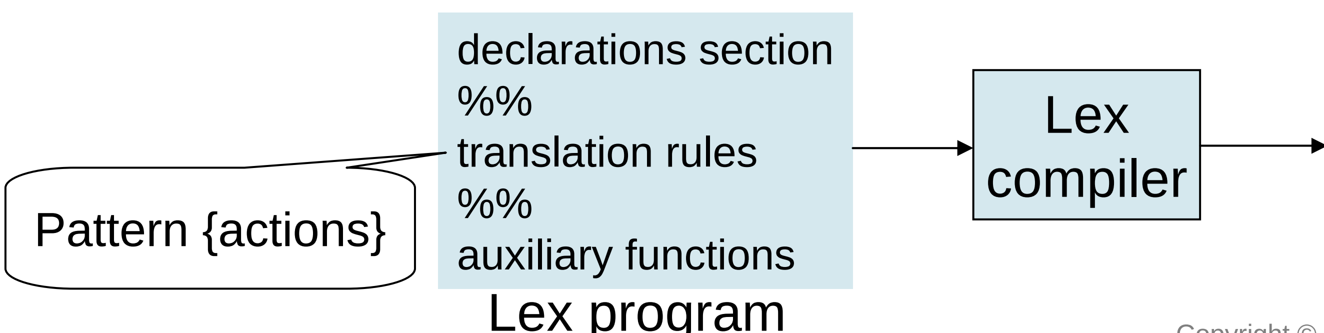
Design of a Lexical-Analyzer Generator



The Architecture of a Lexical Analyzer

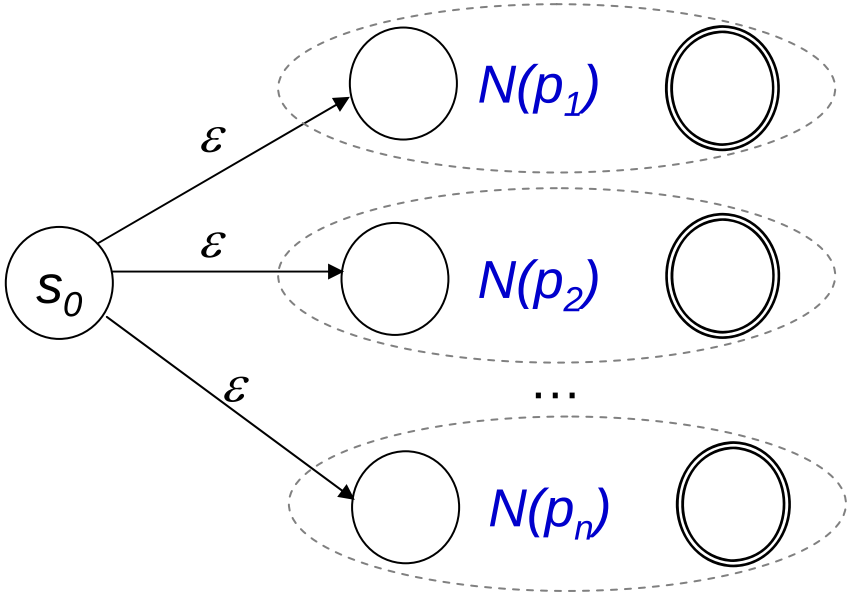
- A Lex program is turned into a transition table and actions, which are used by a finite-automation simulator.
- Components in the lexical analyzer:
  - A transition table for the automation
  - The functions that are directly passed to the output from the Lex program
  - The actions from the input Lex program, which appears as fragments of code to be invoked by the automation simulator.





Automation Construction in *Lex*

- Steps to construct the automation:
  - 1. Take each regular-expression pattern in the ***Lex*** program and convert it to an NFA by using the **McNaughton-Yamada-Thompson algorithm**.
  - 2. Combine all the NFAs into one by introducing a new start state with ε-transitions to the start state of each NFA N_i for pattern p_i.





An Example of an NFA Construction

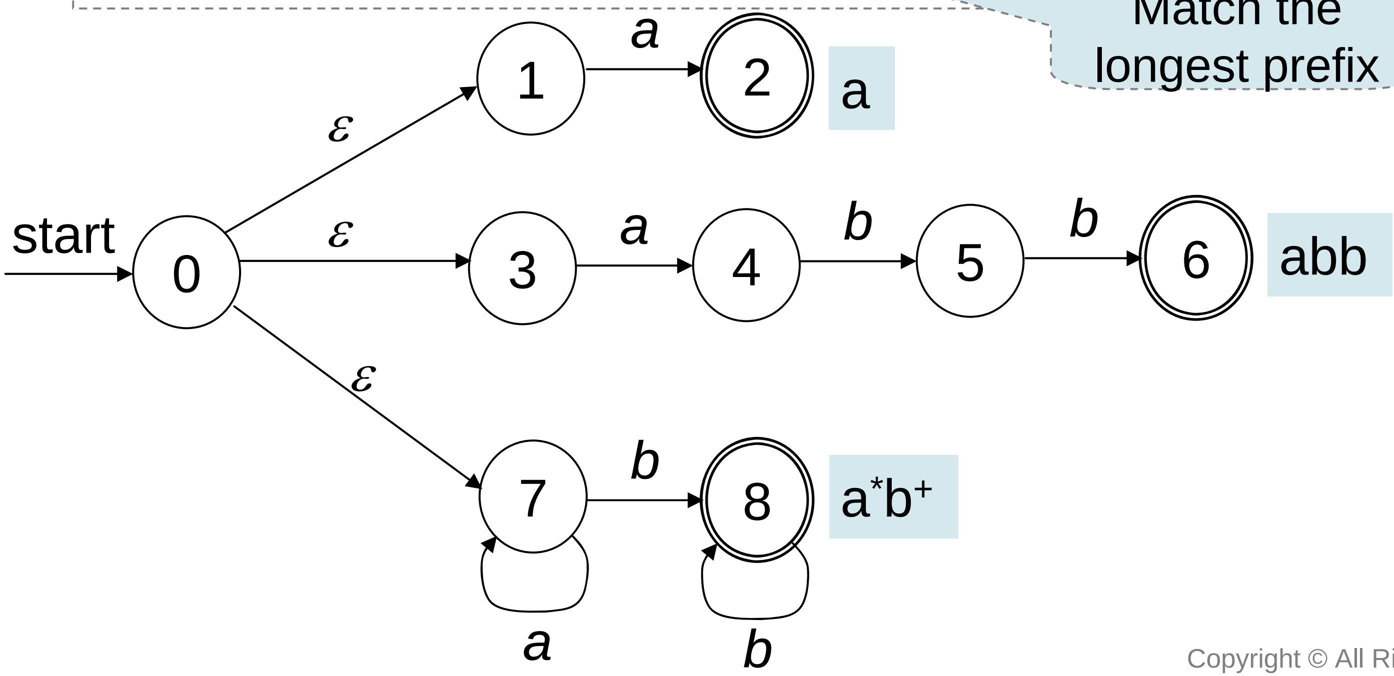
- String *abb* matches both the p_2 and p_3, but we shall consider it a lexeme for p_2.
- String *aabbb* matches p_3.

Match first matched rule

Match the longest prefix

Regular expression in Lex

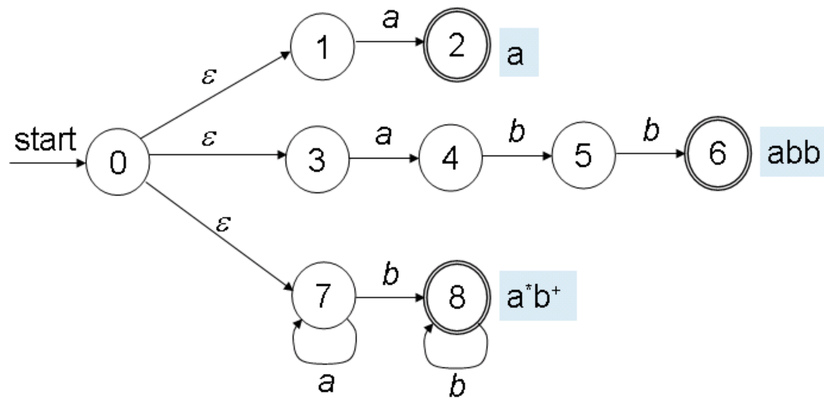
a	{ action A_1 for pattern p_1 }
abb	{ action A_2 for pattern p_2 }
a^*b^+	{ action A_3 for pattern p_3 }

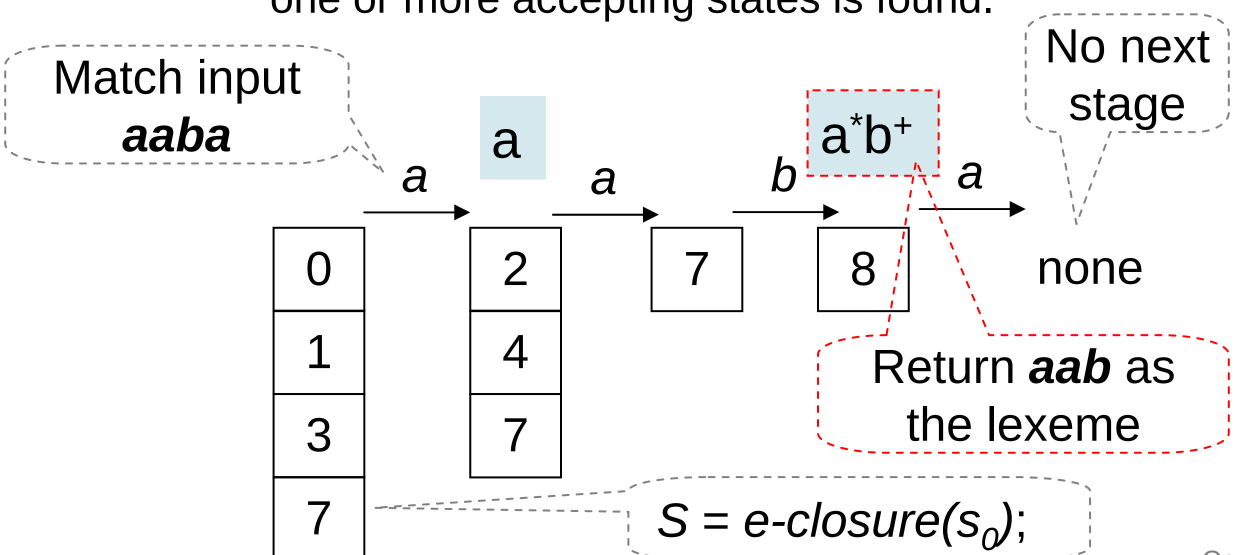




Pattern Matching Based on NFAs

- Pattern matching in NFAs:
  - 1. Adopt the “Simulating an NFA” algorithm to analyze the input string until there are no next states. (The algorithm should be adjusted.)
  - 2. Then decide the longest prefix:
    - Look backwards in the sequence of sets of states, until a set that includes one or more accepting states is found.



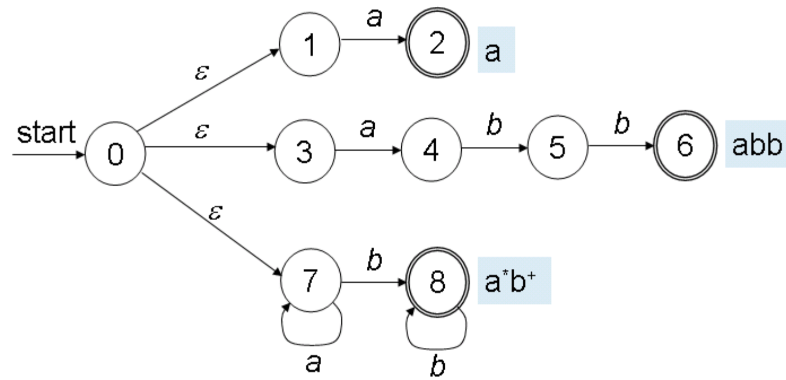


1) *S* = *ε-closure*(s_0);
2) *c* = *nextChar*();
3) **while** (*c* != **eof**) {
4) *S* = *ε-closure(move(S,c))*;
5) *c* = *nextChar*();
6) }
7) **if** ($S \cap F$!= ø) **return** “yes”;
8) **else return** “no”;

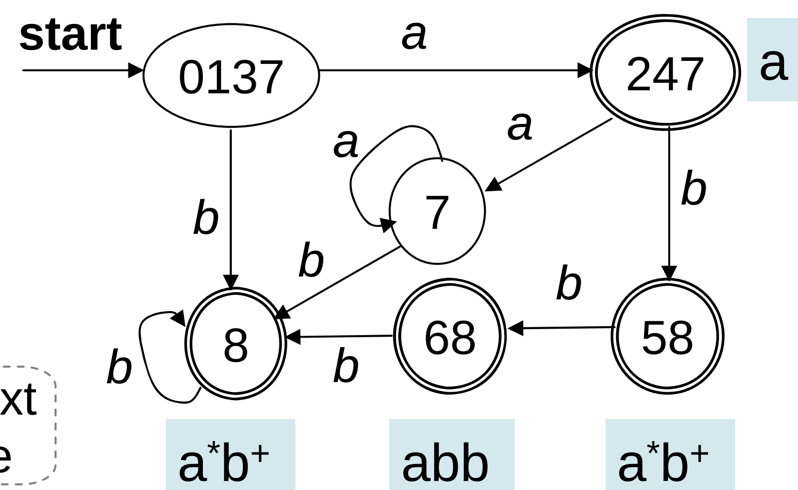
“Simulating an NFA” algorithm

DFAs for Lexical Analyzers

- Pattern matching in DFAs:
  - 1. Conver NFAs to DFAs by the subset construction.
  - 2. Adopt the "Simulating a DFA" algorithm to analyze the input string until there are no next states, i.e., ø or dead state.
  - 3. Look backwards in the sequence of sets of states, until a set that includes one or more accepting states is found.



Subset construction



- If there are more than one accepting states in a DFA state, determine the first pattern whose accepting state is represented in the Lex program, and return the matched pattern.
  - E.g., The state {6, 8} has two accepting states **abb** and **a^*b^+**, but only the former is matched.

Match input ***abba***

0137 $\xrightarrow{a}$ 247 $\xrightarrow{b}$ 58 $\xrightarrow{b}$ 68 $\xrightarrow{a}$

No next stage

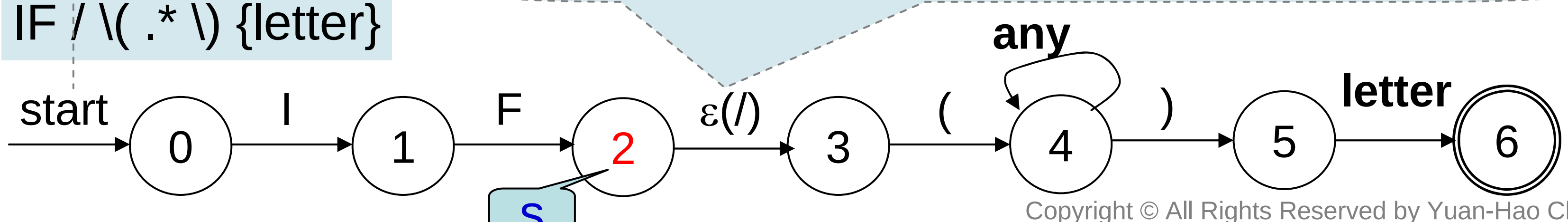
Implementing the Lookahead Operator

- Lookahead operator / is sometimes necessary.
  - When converting the pattern r_1/r_2 to an NFA, we treat the / as if it were ε, so we do not actually look for a / on the input.
  - If the NFA recognizes a prefix *xy* of the input buffer as matching this regular expression, the end occurs when the NFA enters a state *s* such that:
    - 1. *s* has an ε-transition on the (imaginary) /,
    - 2. There is a path from the start state of the NFA to state *s* that spells out *x*.
    - 3. There is a path from state *s* to the accepting state that spells out *y*.
    - 4. *x* is as long as possible for any *xy* satisfying conditions 1-3.

x y

IF / \(.* \) {letter}

The lexeme **IF** is found by scanning backwards to the last occurrence of state 2, whenever state 6 is entered.





Optimization of DFA-Based Pattern Matchers

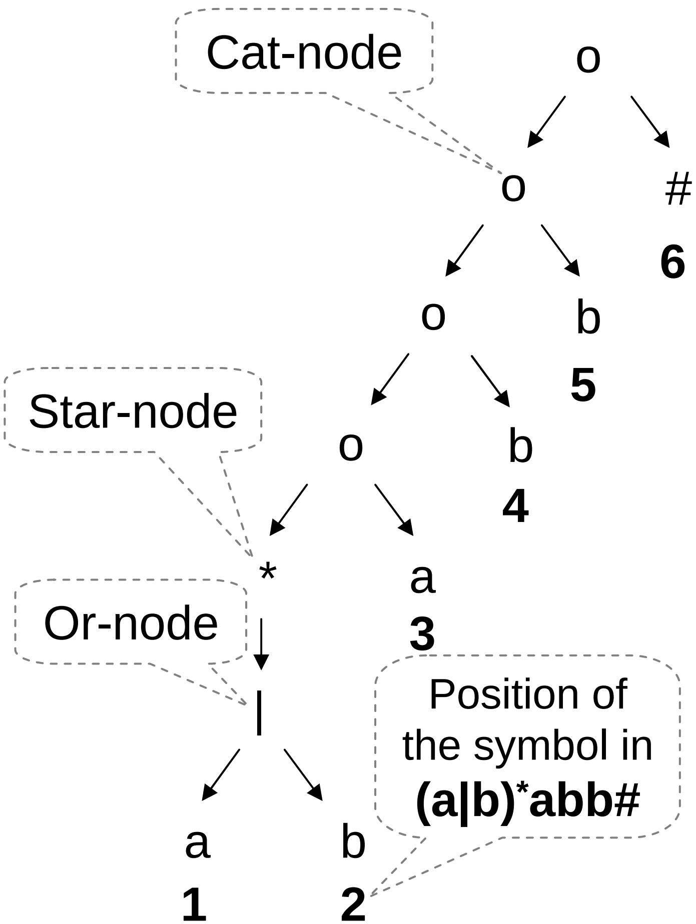
Optimize DFA-Based Pattern Matchers

- Three algorithms are widely adopted to optimize pattern matchers constructed from regular expressions.
  - 1. Converting a regular expression directly to a DFA
    - Construct DFA directly from a regular expression. This is useful in a **Lex** compiler.
  - 2. Minimizing the number of states of a DFA
    - Combine states that have the same future behavior.
    - The state minimization is with the time complexity $O(n \log n)$ where n is the number of states in the DFA.
  - 3. Trading time for space in DFA simulation
    - Compact the representations of translation tables

Important States of an NFA

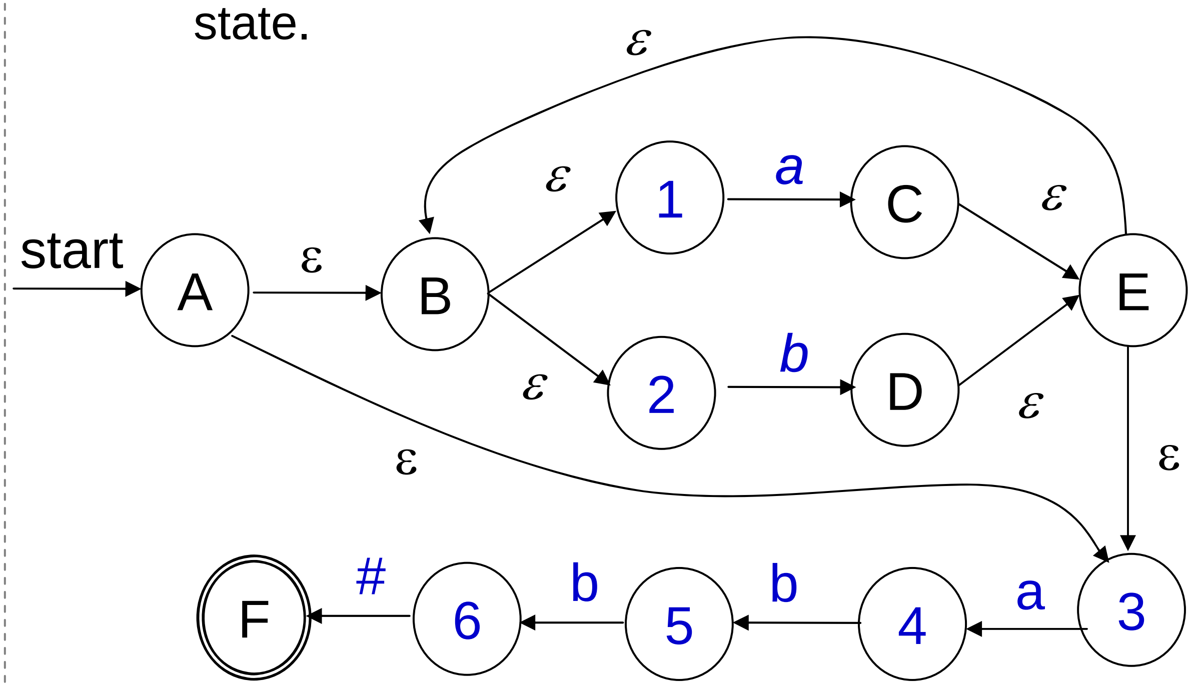
- Important state
  - A state of an NFA is an important state if it has a non-ε out-transition.
- When the NFA is constructed from **McNaughton-Yamada-Thompson** algorithm,
  - Each important state corresponds to a particular operand in the regular expression.
    - Each important state of the NFA corresponds directly to the position in the regular expression that holds symbols of the alphabet.
  - Only the important states in a set *T* are used when it computes *ε-closure(move(T, a))*.
    - Because the set of states *move(s, a)* is nonempty only if state *s* is important.
  - Two sets of NFA states can be treated as if they were the same set if they
    - 1. Have the same important states, and
    - 2. Either both have accepting states or neither does.

Augmented Regular Expression



Syntax tree of **(a|b)*abb#**

- The augmented regular expression ***(r)*#**
  - Give the accepting state for a transition on # to make the accepting state an important state.



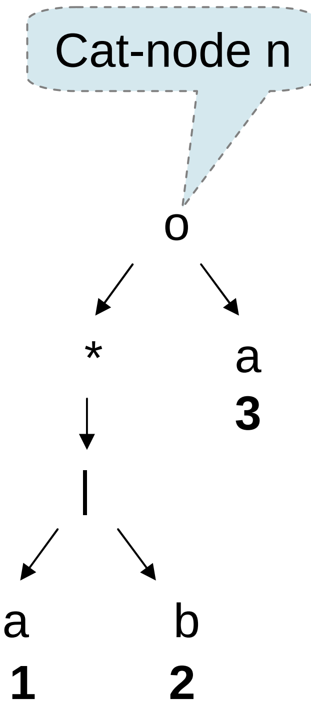


Functions Computed from the Syntax Tree

- To construct a DFA from a regular expression, we construct its syntax tree with four functions:
  - *nullable(n)*
    - Is true for a syntex-tree node *n* iff the subexpression represented by *n* has ε in its language. That is, the subexpression can be the null string.
  - *firstpos(n)*
    - Is the set of positions (in the subtree rooted at *n*) that can be the first symbol of at least one string in the subexpression rooted at *n*.
  - *lastpos(n)*
    - Is the set of positions (in the subtree rooted at *n*) that can be the last symbol of at least one string in the subexpression rooted at *n*.
  - *followpos(p)*
    - Is the set of positions *q* (in the entire syntax tree) that could follows *p*.

An Example of the Four Functions

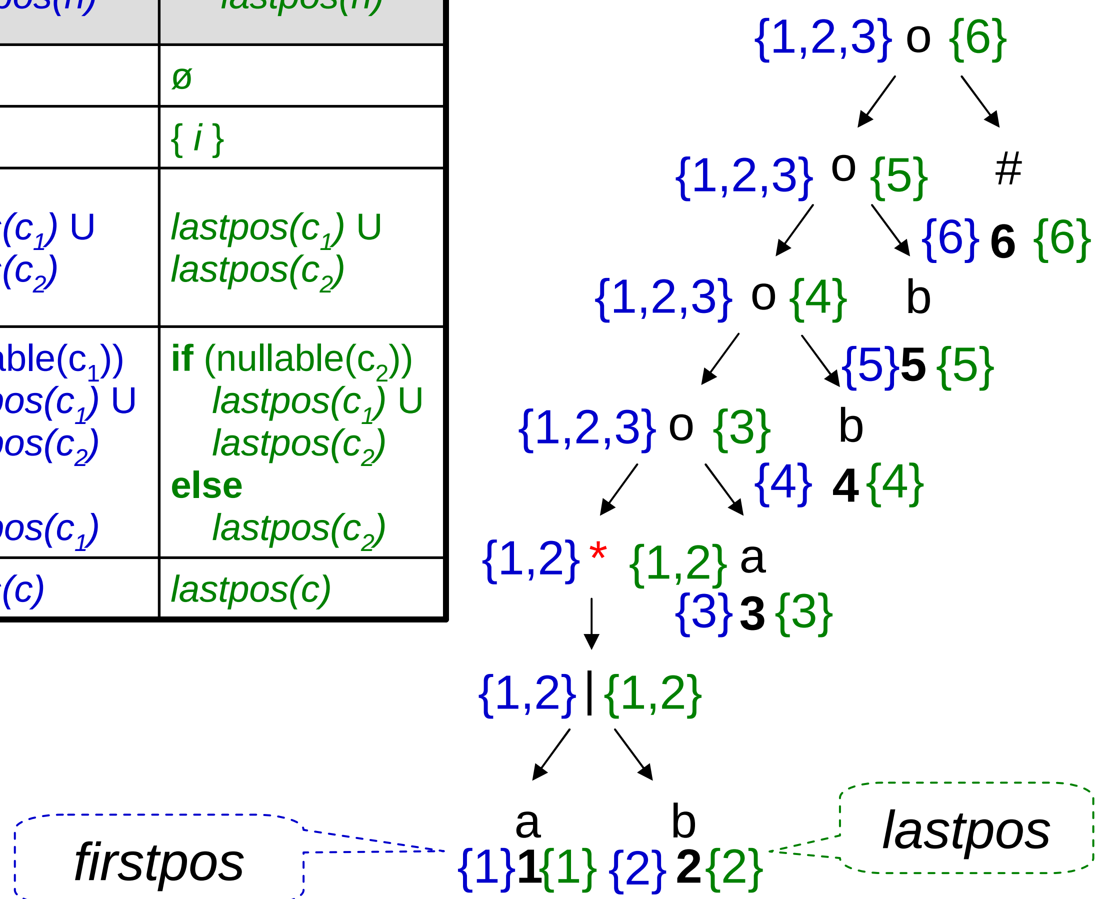
- Consider the cat-node n that corresponds to the expression **(a|b)*a**
  – *nullable(n) = false* since it ends in an *a*.
    - *nullable((a|b)*)=true*: only star-node or ε is nullable
  – *firstpos(n)* = {1, 2, 3}
    - E.g.,
      - The string *aa* could start from position 1.
      - The string *ba* could start from position 2.
      - The string *a* could start from position 3.
  – *lastpos(n)* = {3}
    - E.g., any string match this expression ends at position 3.
  – *followpos(1)* = {1, 2, 3}
    - Consider a string *ac*.
      - *c* is either a (position 1) or b (position 2) according to (a|b)*.
      - *c* comes from position 3 if *a* is the last in the string generated by (a|b)*.



Rules for Computing the Four Functions

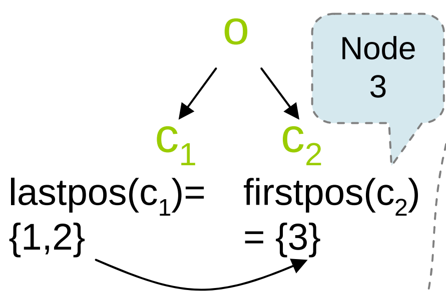
Syntax tree of **(a|b)*abb#**
Position: **1 2 3456**

NODE *n*	*nullable(n)*	*firstpos(n)*	*lastpos(n)*
A leaf labeld ε	**true**	ø	ø
A leaf with position *i*	**false**	{ *i* }	{ *i* }
An or-node n = c_1 | c_2	*nullable(c_1)* **or** *nullable(c_2)*	*firstpos(c_1)* U *firstpos(c_2)*	*lastpos(c_1)* U *lastpos(c_2)*
A cat-node n = c_1 c_2	*nullable(c_1)* **and** *nullable(c_2)*	**if** (nullable(c_1)) *firstpos(c_1)* U *firstpos(c_2)* **else** *firstpos(c_1)*	**if** (nullable(c_2)) *lastpos(c_1)* U *lastpos(c_2)* **else** *lastpos(c_2)*
A star-node n = c^*	**true**	*firstpos(c)*	*lastpos(c)*



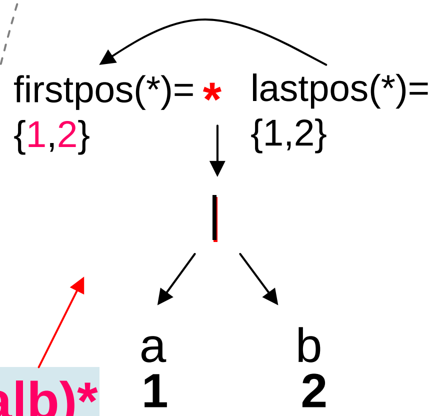
Rules for Computing the Four Functions (Cont.)

- Only two ways that a position of a regular expression can be made to follow another:
  - 1. If *n* is a **cat-node** with left child c_1 and right child c_2, for every position *i* in *lastpos*(c_1), all positions in *firstpos*(c_2) are in *followpos(i)*.
  - 2. If *n* is a **start-node**, and *i* is a position in *lastpos(n)*, then all positions in *firstpos(n)* are in *followpos(i)*.



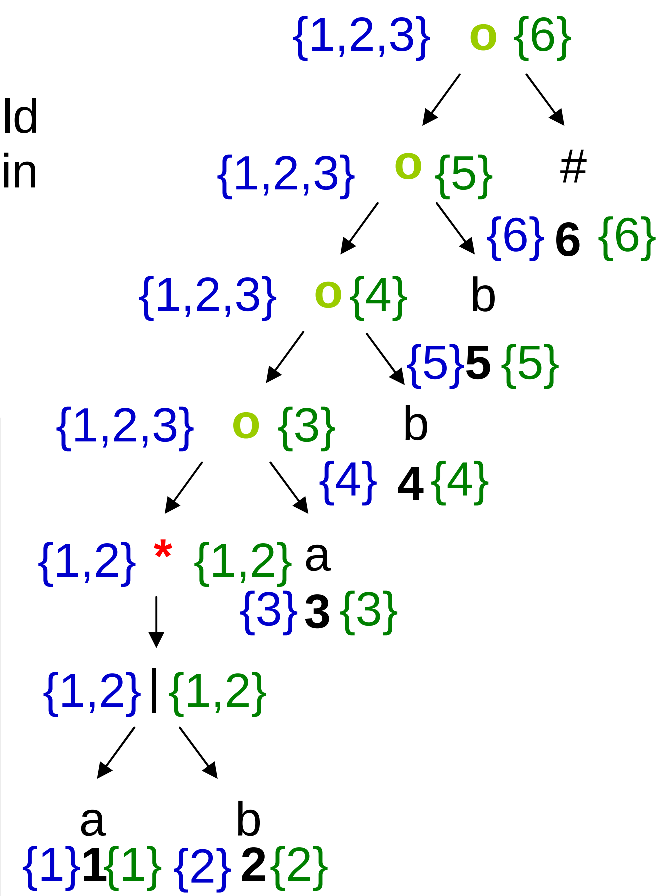
followpos(1) = {3}
followpos(2) = {3}

followpos(1) = {1,2}
followpos(2) = {1,2}



NODE *n*	*followpos(n)*
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	ø

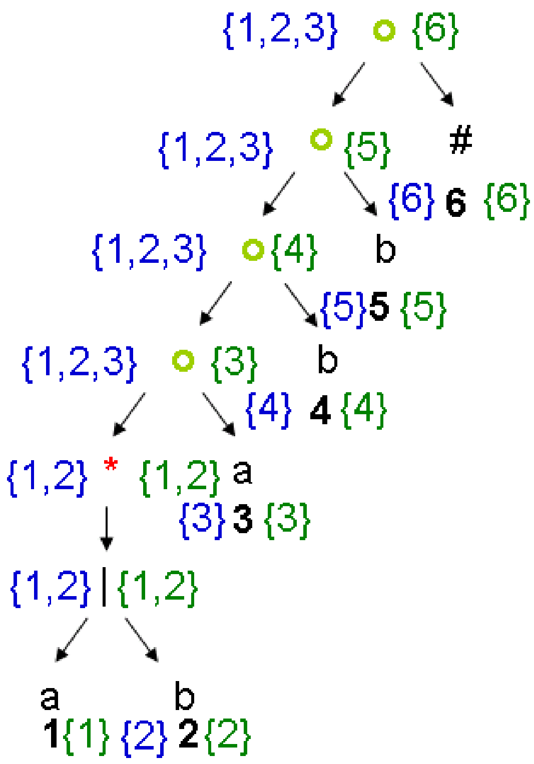
Syntax tree of **(a|b)*abb#**

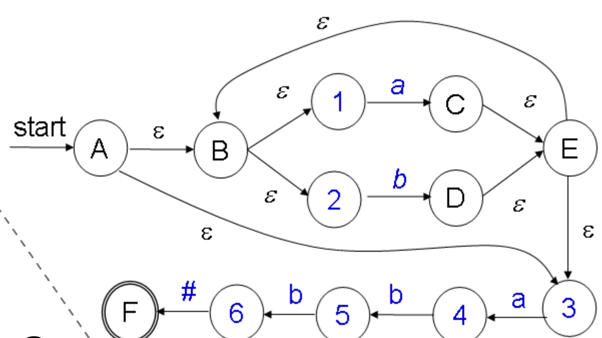


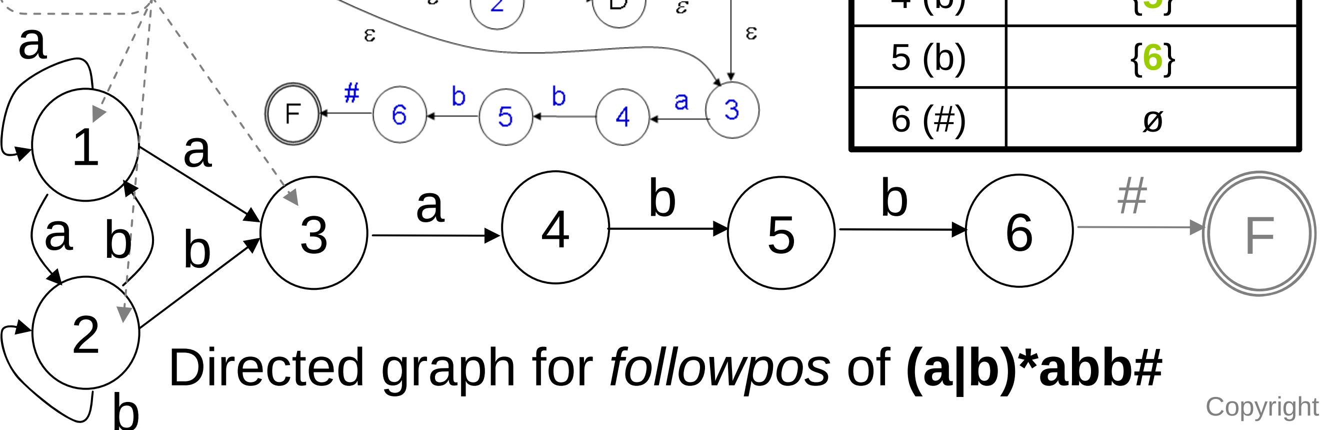
Directed Graph for the Function *followpos*

- The directed graph for *followpos* is almost an NFA without ε-transitions. We can convert it to an NFA by
  - Making all positions in *firstpos* of the root be initial states.
  - Labeling each arc from i to j by the symbol at position i.
  - Making the position associated with endmarker # be the only accepting state.

NODE *n*	*followpos(n)*
1 (a)	{1,2,3}
2 (b)	{1,2,3}
3 (a)	{4}
4 (b)	{5}
5 (b)	{6}
6 (#)	ø







Directed graph for *followpos* of **(a|b)*abb#**



Converting a Regular Expression to a DFA Directly

- **Algorithm: Construction of a DFA from a regular expression *r*.**
- INPUT: A regular expression *r.*
- OUTPUT: A DFA *D* that recognizes *L(r)*.
- METHOD:
  - 1. Construct a syntax tree *T* from the augmented *r(#)*.
  - 2. Compute *nullable*, *firstpos*, *lastpos*, and *followpos* for *T*.
  - 3. Construct *Dstates* (the set of states of DFA *D*) and *Dtran* (the transition function for *D*).
    - The states of *D* are sets of positions in *T*.
    - The start state of *D* is *firstpos*(n_0), where n_0 is the root node of *T*.
    - The accepting states are those containing the endmarker #.
    - Initially, each state is *unmarked* and a state becomes marked when evaluated.

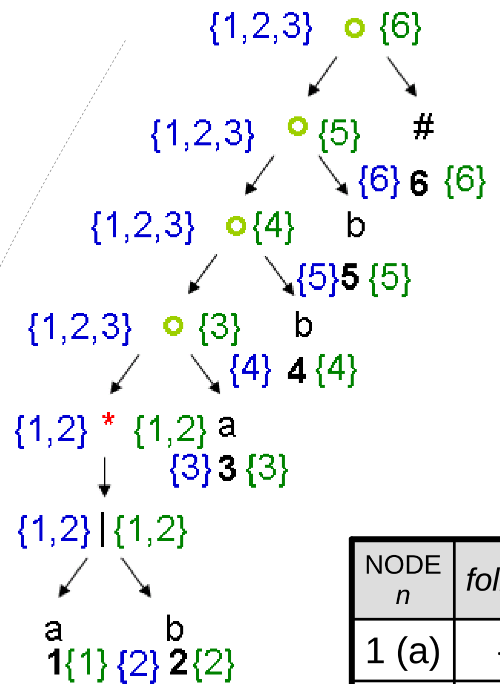
```
Initialize Dstates to contain only the unmarked state firstpos(n0), where n0 is the root of syntax tree T for (r)# ;
while (there is an unmarked state S in Dstates) {
   mark S;
   for (each input symbol a) {
      let U be the union of followpos(p) for all p in S that correspond to a;
      if ( U is not in Dstates ) add U as an unmarked state to Dstates;
      Dtran[S, a] = U;
   }
}
```

Converting a Regular Expression to a DFA Directly (Cont.)

Correspond to a

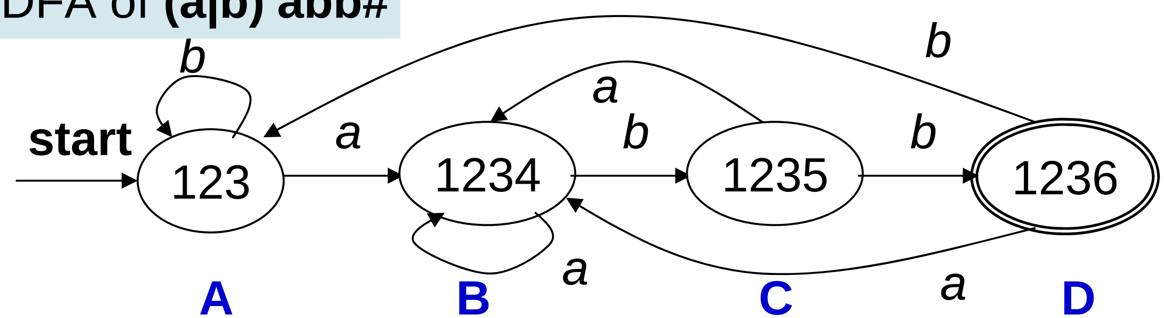
Correspond to a

- *firstpos(n_0)* = {1, 2, 3} = A
- Transition of A {1, 2, 3}
  - *Dtran[A, a]* = followpos(1) U followpos(3) = {1, 2, 3, 4} = B
  - *Dtran[A, b]* = followpos(2) = {1, 2, 3} = A
- Transition of B {1, 2, 3, 4}
  - *Dtran[B, a]* = followpos(1) U followpos(3) = {1, 2, 3, 4} = B
  - *Dtran[B, b]* = followpos(2) U followpos(4) = {1, 2, 3, 5} = C
- Transition of C {1, 2, 3, 5}
  - *Dtran[C, a]* = followpos(1) U followpos(3) = {1, 2, 3, 4} = B
  - *Dtran[C, b]* = followpos(2) U followpos(5) = {1, 2, 3, 6} = D
- Transition of D {1, 2, 3, 6}
  - *Dtran[D, a]* = followpos(1) U followpos(3) = {1, 2, 3, 4} = B
  - *Dtran[D, b]* = followpos(2) = {1, 2, 3} = A



NODE *n*	*followpos(n)*
1 (a)	{1,2,3}
2 (b)	{1,2,3}
3 (a)	{4}
4 (b)	{5}
5 (b)	{6}
6 (#)	ø

DFA of **(a|b)*abb#**

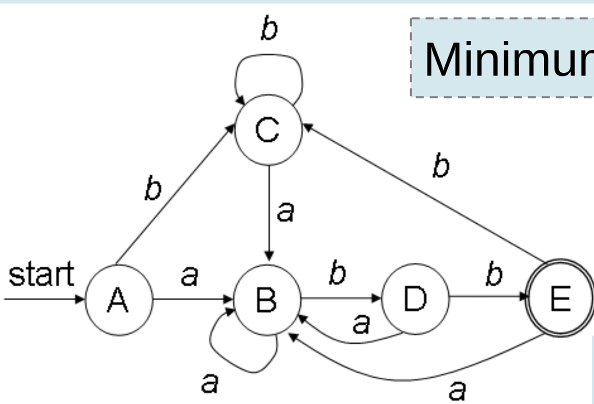


DFA State	*a*	*b*
A	B	A
B	B	C
C	B	D
D	B	A

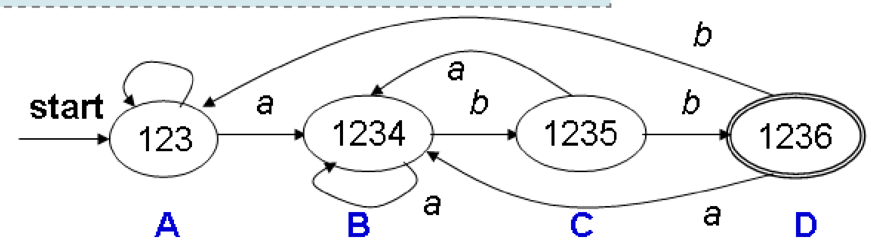
Dtran

Minimizing the Number of States of a DFA

- There can be many DFAs that recognize the same language.
- Two automata are the same if one can be transformed into the other by doing nothing more than changing the names of states.
- There is always a unique minimum state DFA for any regular language.
  - State A and C are equivalent because they transfer to the same state on any input. → Both A and C behave like state 123.
  - State B behaves like state 1234.
  - State D behaves like state 1235.
  - State E behaves like state 1236.



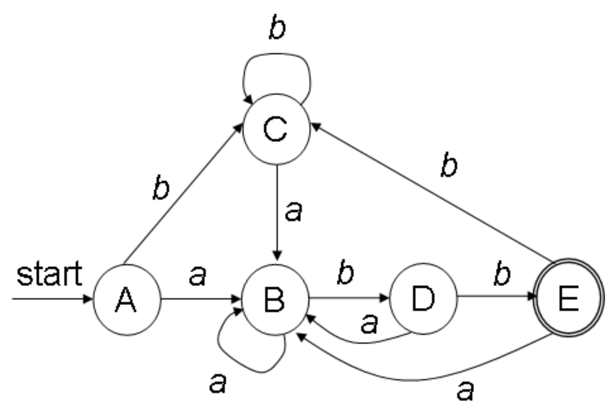
Minimum-state DFA = {A, C} {B} {D} {E}



DFAs of **(a|b)*abb#**

Distinguishing States

- State *s* is distinguishable from state *t* if there is some string that distinguishes them.
  - String *x* distinguishes state *s* from state *t* if exactly one of the states reached from *s* and *t* by following the path with label *x* is an accepting state.
  - E.g., string *bb* distinguishes state A from state B.
    - String bb takes
      - A to the non-accepting state C.
      - B to the accepting state E.





State Minimization for DFA

- State minimization
  - Partition the states of a DFA into groups of states that can't be distinguished.
  - Then merge states of each group into a single state of the minimum-state DFA.
- State-minimization algorithm
  - Maintain a partition, whose groups are sets of states that have not yet been distinguished.
    - Note that any two states from different groups are distinguishable.
  - When the partition can't be refined by breaking any group into smaller groups, the minimum-state DFA is derived.
    - In practice, the initial partition usually consists of two groups: the nonaccepting states $A=\{s_1, s_2, \ldots, s_k\}$ and accepting states F.
    - Then take some input symbol to see whether the input symbol can distinguish between any states in group A, and split A into groups.



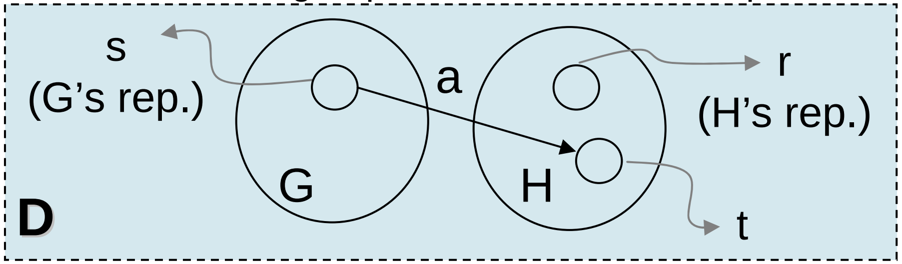
State-Minimization Algorithm for DFA

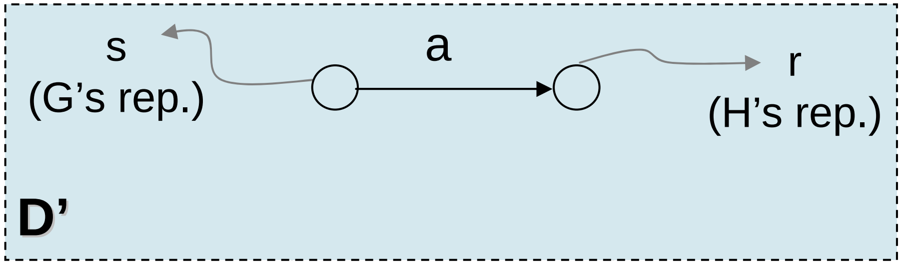
- **Algorithm: Minimizing the number of states of a DFA.**
- INPUT: A DFA *D* with set of states *S*, input alphabet $\sum$, start state s_0, and set of accepting states *F*.
- OUTPUT: A DFA *D'* accepting the same language as D and having as few states as possible.
- METHOD:
  - **1.** Start with an initial partition Π with two groups, F (the accepting states) and S-F (the non-accepting states).
  - **2.** Construct a new partition Π_{new}.

Let $\Pi_{new} = \Pi$;
for (each group G of Π) {
    partition G into subgroups such that two stats *s* and *t* are in the same subgroup iff
    for all input symbols *a*, states *s* and *t* have transitions on *a* to states in the same group of Π;
    /* at worst, a state will be in a subgroup by it self */
    replace G in Π_{new} by the set of all subgroups formed;
}

State-Minimization Algorithm for DFA (Cont.)

- **3.** If $\Pi_{new} = \Pi$, let $\Pi_{final} = \Pi$. Otherwise repeat step (2) with Π_{new} in place of Π.
- **4.** Choose one state in each group of Π_{final} as the *representative* for that group. The representatives will be the stats of the minimum-state DFA *D'*.
  - (a) The start state of D' is the representative of the group containing the start state of D.
  - (b) the accepting states of D' are the representatives of those groups that contain an accepting state of D.
    - Note that each group contains either only accepting states or only nonaccepting states because the initial partition separates those into two groups.
  - (c) Let *s* be the representative of some group *G* of Π_{final}, and let the transition of D from *s* on input *a* be to state *t*. Let *r* be the representative of t's group *H*. Then in D', there is a transition from *s* to *r* on input *a*.
    - Note that in *D*, every state in group *G* must go to some state of group *H* on input *a*, or else, group G would have been split.



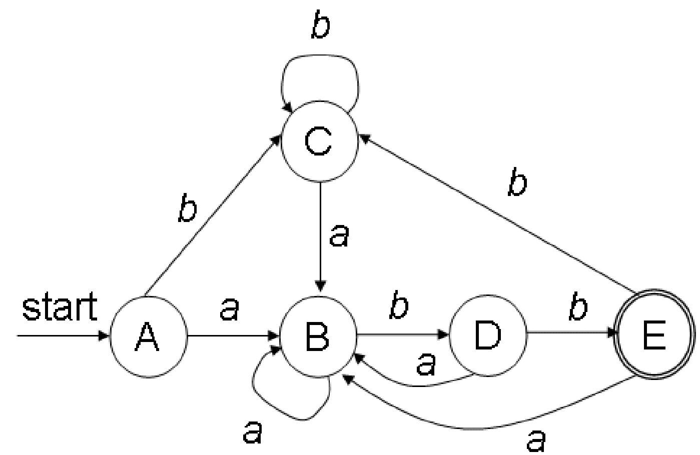




An Example of the DFA State Minimization

- Step 1: Initial partition: {A, B, C, D} {E}
- Step 2 – first iteration with partition {A, B, C, D} {E}
  - Group {E} can't be split because it has only one state.
  - Group {A, B, C, D}
    - On input *a*, A, B, C, and D go to the same group {A, B, C, D}.
    - On input *b*, A, B, and C go to the same group {A, B, C, D}, but D goes to the other group {E}.
      → Split {A, B, C, D} into {A, B, C} {D}
- Step 2 – second iteration with partition {A, B, C} {D} {E}
  - Groups {D} and {E} can't be split.
  - Group {A, B, C}
    - On input *a*, A, B, and C go to the same group {A, B, C}
    - On input *b*, A and C go to the same group {A, B, C}, but B goes to the other group {D}.
      → Split {A, B, C} into {A, C} {B}
- Step 2 – third iteration with partition {A, C} {B} {D} {E}
  - Groups {B}, {D}, and {E} can't be split.
  - Group {A, C}
    - On input *a*, A and C go to the same group {B}
    - On input *b*, A and C go to the same group {C}
      → No further split
- Step 3: $\Pi_{new} = \Pi$, let $\Pi_{final} = \Pi$ = {A, C} {B} {D} {E}
- Step 4: Choose representatives to construct D'

Let's pick A here.



DFA State	a	b
A	B	A
B	B	D
D	B	E
E	B	A

***DTran* for DFA D'**

DFA State	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

***DTran* for DFA D**

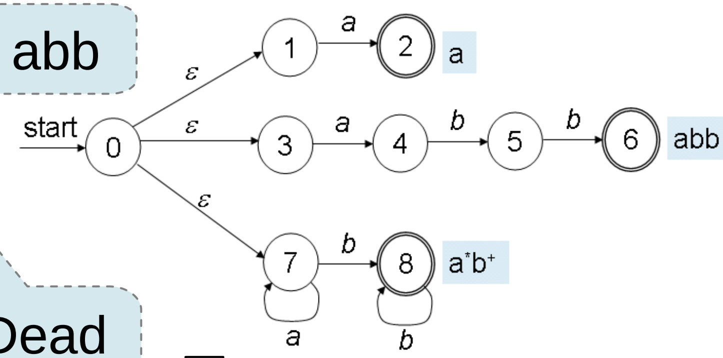


State Minimization in Lexical Analyzer

Indicate no token | a | a*b+ | abb

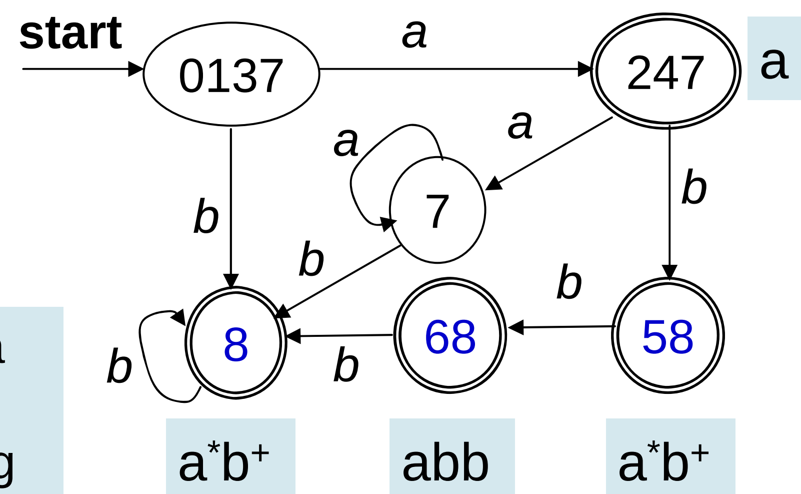
- Step 1:
  Initial partition: {0137, 7} {247} {8, 58} {68} {ϕ}
  - Group all states that recognize a particular token, and also
  - Group those states that do not indicate any token

Dead state



Subset construction

- Step 2: Split step
  - Split {0137, 7} because they go to different groups on input *a*.
  - Split {8, 58} because they go to different groups on input *b*.



- Dead state is to target the missing transitions on a from states 8, 58, and 68.
- The dead state is dropped so that we treat missing transitions as a signal to end token recognition.



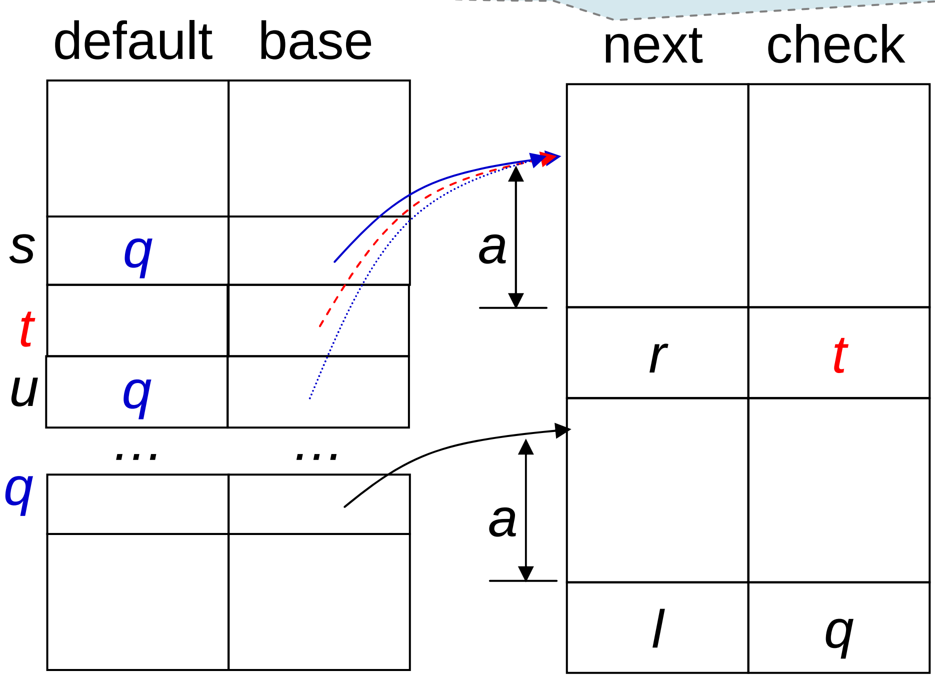
Trading Time for Space in DFA Simulation

- The transition table is the main memory overhead to construct a DFA.
  - The simplest and fastest way is to use a two-dimensional table indexed by states and characters.
    - E.g., *Dtran[state, character]*
  - A typical lexical analyzer has several hundred states and involves the ASCII alphabet of 128 input characters.
    - The array size consumes less than 1 megabyte.
- In small devices, the transition table should be compacted.
  - E.g., Each state represented by a list of transitions ended by a default state for any input characters not on the list.

Trading Time for Space in DFA Simulation (Cont.)

- Another table compaction method:
  - Combine the speed of array access with the compression of lists with defaults.

Make the next-check arrays short by taking advantage of the similarities among states.




```
int nextState(s,a) {
    if (check[base[s] + a] = s)
        return next[ base[s] + a ];
    else
        return nextState(default[s], a);
}
```

Process as if *q* were the current state.