

Compilers Construction

An Overview

Dr. Mohammed Salem Atoum

M.Atoum@inu.edu.jo

Course Goals

- Provide a practical introduction to compiler concepts and their construction
- Focus on both **theory** and **practice**
 - Theory demonstrates strong mathematical basis of compilers
 - Practice provides experimental framework for illustrating theory

Course Learning Requirements

○ Evaluation / Earning Credit

- First Exam 25%
- Second Exam 25%
- Final Exam 50%

Textbooks and Class Materials

- Textbook:
 - *Compiler Construction: Principles and Practice*, K. Louden, PWS, 1997
 - *Class Notes (Slides)*

Course Outline (may changed):

○ Introduction	1
○ Scanning	2
○ CFG & Parsing	3
○ Top-Down Parsing	4
○ Bottom-Up Parsing	5
○ Semantic Analysis	6
○ Run-Time Environment	7
○ Code Generation	8

Cheating Policy

- Cheating is an illegal and unethical activity
- Standard INU policy will be applied
- Cooperative problem solving and sharing code is NOT permitted and will likely lead to ZERO!

Any Question?

Compilers Construction

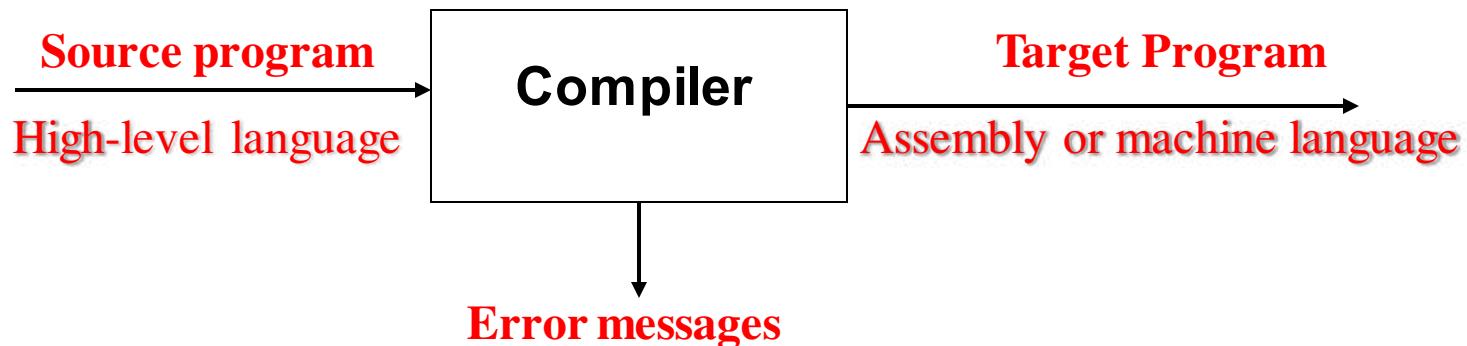
Introduction

Dr. Mohammed Salem Atoum

M.Atoum@inu.edu.jo

What is a Compiler? *Read Chapter 1*

- Is a program that *translates* one language to another
 - Takes as input a source program typically in a high-level language
 - Produces an equivalent target program in assembly or machine language
 - Reports error messages as part of the translation process



Brief History

- First computers of late 1940s were programmed in **machine language**
 - C7 06 0000 0002
is an instruction to move the number 2 to the location 0000 (in hexadecimal) on the Intel 8x86 processor
- Machine language was soon replaced by **assembly language**
 - Instructions and memory locations are given symbolic names
 - An *assembler* translates the symbolic assembly code into equivalent machine code
 - Assembly language improved programming, but is still machine dependent
 - **MOV x, 2**
is equivalent to above machine instruction

Brief History

- First compiler was for the **high- level language** FORTRAN
 - Developed between 1954 and 1957 at IBM by a group led by John Backus
 - Proved the viability of high-level and thus less machine dependent languages
- The study of **scanning** and **parsing** were pursued in the 1960s and 1970s
 - Led fairly to a complete solution
 - Became standard part of compiler theory
 - Resulted in scanner and parser generators that automate part of compiler development
- Methods of generating efficient **target code** is still an ongoing research
 - Known as optimization or code improvement techniques
- Compiler technology was also applied in rather unexpected areas:
 - Text-formatting languages
 - Hardware description languages for the automatic creation of VLSI circuits

The operation of a compiler involves several different kinds of files

- A **source code** text file (**.c**, **.cpp**, **.java**, etc. file extensions)
- **Intermediate code** files: transformations of source code during compilation, usually kept in temporary files rarely seen by the user
- An **assembly code** text file containing symbolic machine code, often produced as the output of a compiler (**.asm**, **.s** file extensions)
- One or more **binary object code** files: machine instructions, not yet linked or executable (**.obj**, **.o** file extensions)
- A **binary executable file**: linked, independently executable (well, not always...) code (**.exe**, **.out** extensions, or no extension)

Why study compiler construction?

- To learn about program design
- It brings together ideas from many different areas of computer science:
 - programming languages, algorithms, architecture, etc
- It has many applications:
 - debugging aids, interpreters, text formatters, improving program performance, etc

What do we want from a compiler?

- Correct code
- Consistent optimization
- Target code runs fast
- Compiler runs fast
- Good cooperation with debugger and good syntax error handling
- Compile time proportional to program size
- Ability to compile modules separately

The Translation (Compilation) Process

- A compiler performs two major tasks:

Analysis

: تحليل

Decompose Source into an intermediate representation

Synthesis

: تركيب

Target program generation from intermediate representation

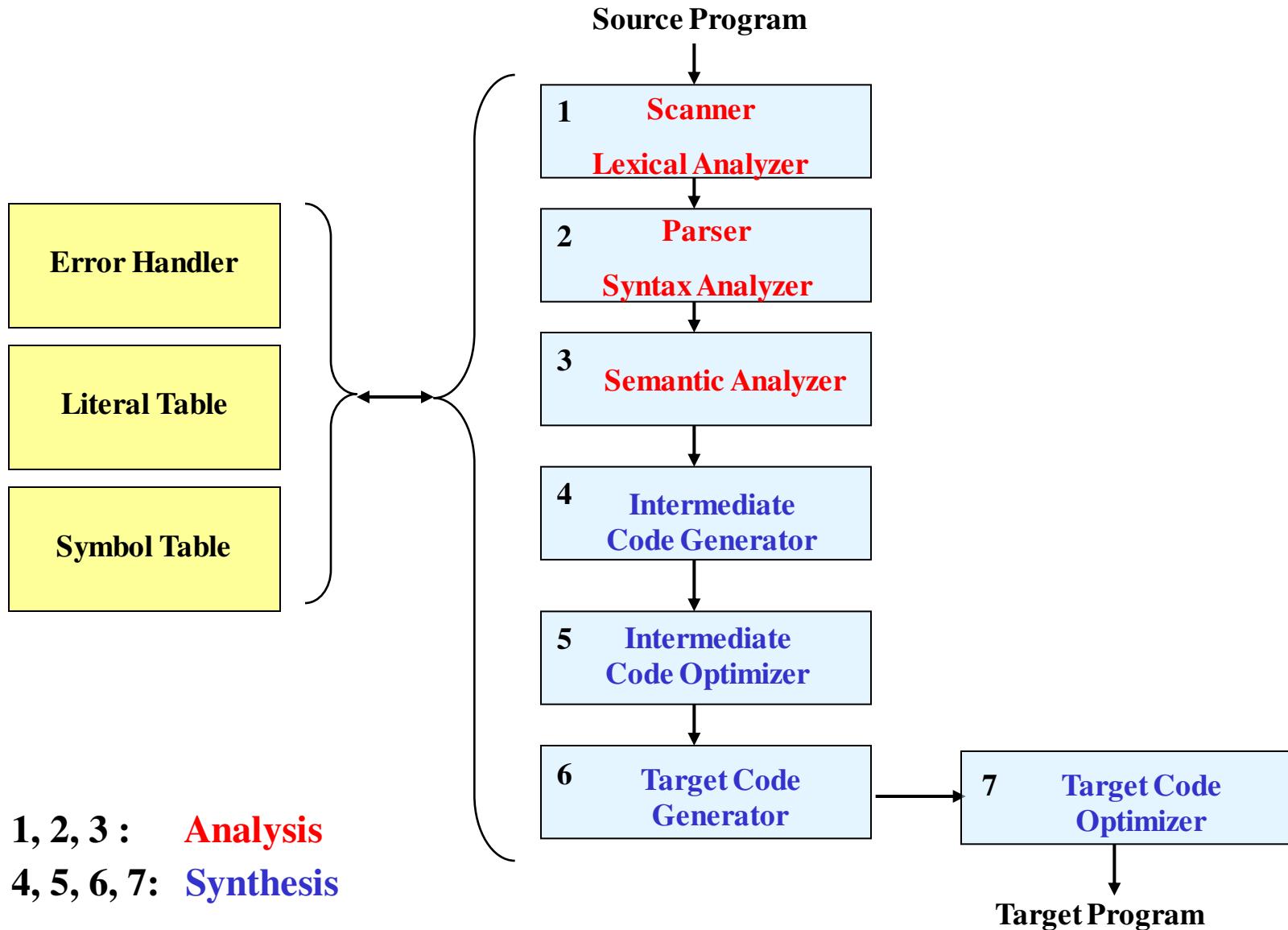
The Translation Process – Cont'd

- Phases of a compiler:

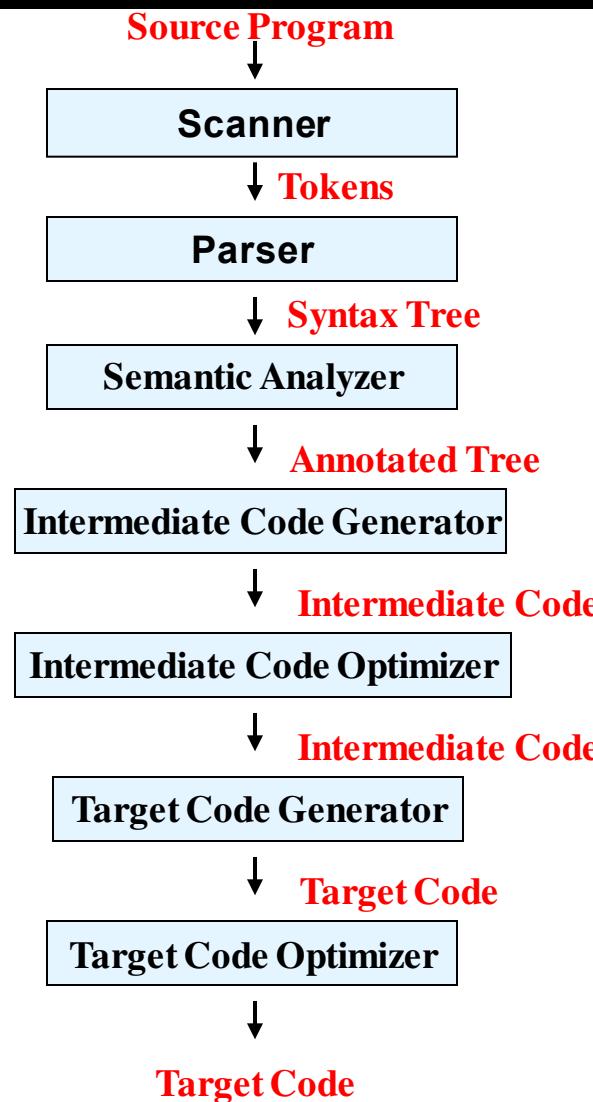
- Scanning
 - Parsing
 - Semantic Analysis
- } **Mainly Analysis**
- Intermediate Code Generation
 - Intermediate Code Optimization
 - Target Code Generator
 - Target Code Optimization
- } **Mainly Synthesis**

- Three auxiliary components interact with some or all phases:
 - Literal Table
 - Symbol Table
 - Error Handler

The Phases of a Compiler



The Phases of a Compiler



The Analysis Task For Compilation

- Three Phases:
 - Scanner (Lexical Analysis):
 - identify words: group characters into tokens
 - Parser (Syntax Analysis):
 - identify structure: group tokens into syntactic units (parse tree)
 - Semantic Analysis:
 - identify meaning: annotate the parse tree with information such as types and perform other tests (type checks, flow of control, scoping and identifier uniqueness, etc.)

Scanner (Lexical Analysis)

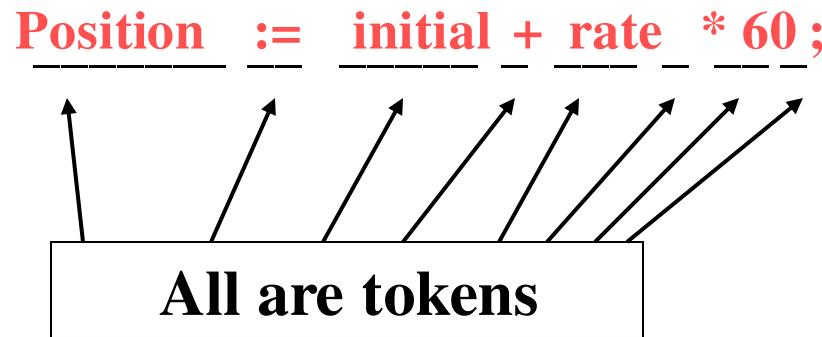
- The scanner begins the analysis of the source program by:
 - Reading file character by character
 - Grouping characters into tokens
 - Eliminating unneeded information (comments and white space)
 - Entering preliminary information into literal or symbol tables
 - Processing compiler directives by setting flags
- **Tokens** represent basic program entities such as:
 - Identifiers, Literals, Reserved Words, Operators, Delimiters, etc.
- Example: **a := x + y * 2.5 ;** is scanned as

a	identifier	y	identifier
:=	assignment operator	*	multiplication operator
x	identifier	2.5	real literal
+	plus operator	;	semicolon

Scanner (Lexical Analysis)

Easiest Analysis - Identify tokens which are building blocks

For Example:



Blanks, Line breaks, etc. are scanned out

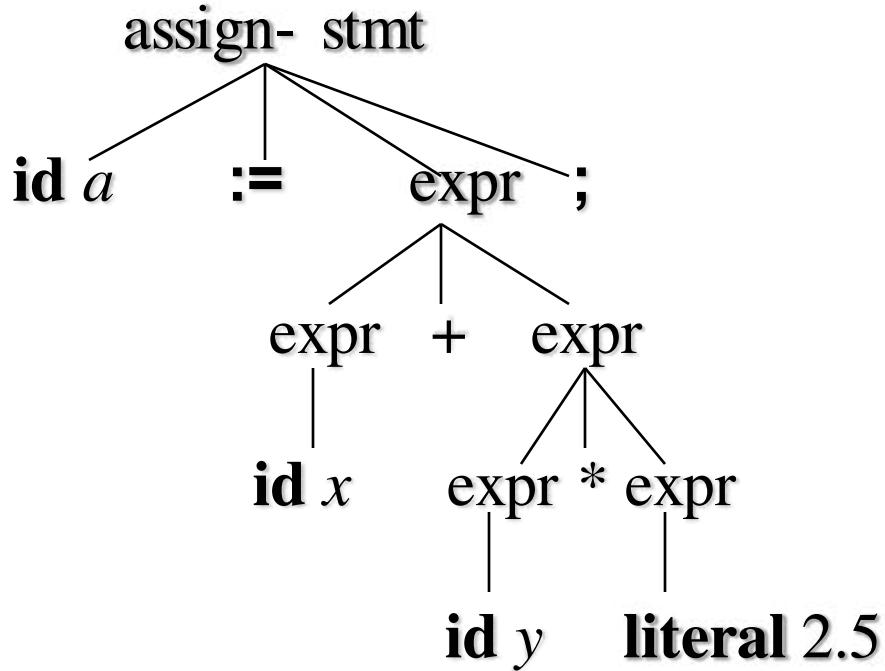
Parser (Syntax Analysis)

- Receives tokens from the scanner
- Recognizes the structure of the program as a **parse tree**
 - Parse tree is recognized according to a context-free grammar
 - Syntax errors are reported if the program is syntactically incorrect
- A parse tree is inefficient to represent the structure of a program
- A **syntax tree** is a more condensed version of the parse tree
- A syntax tree is usually generated as output of the parser

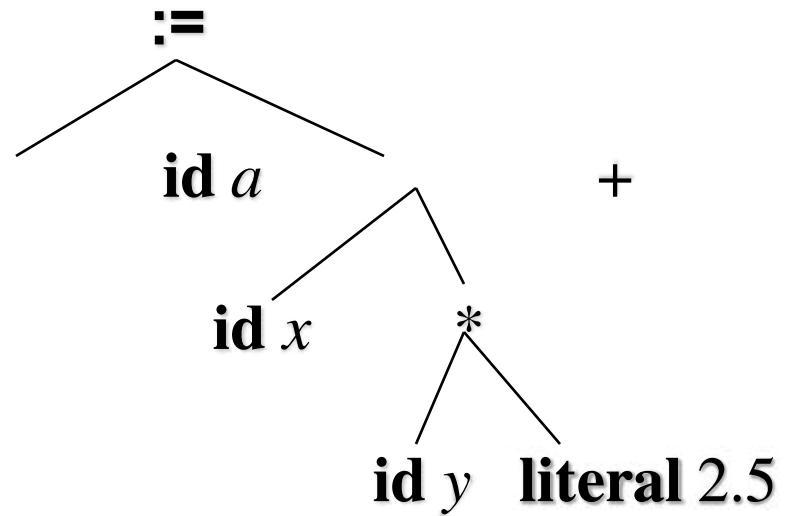
Parser (Syntax Analysis)

a := x + y * 2.5 ;

Parse
Tree



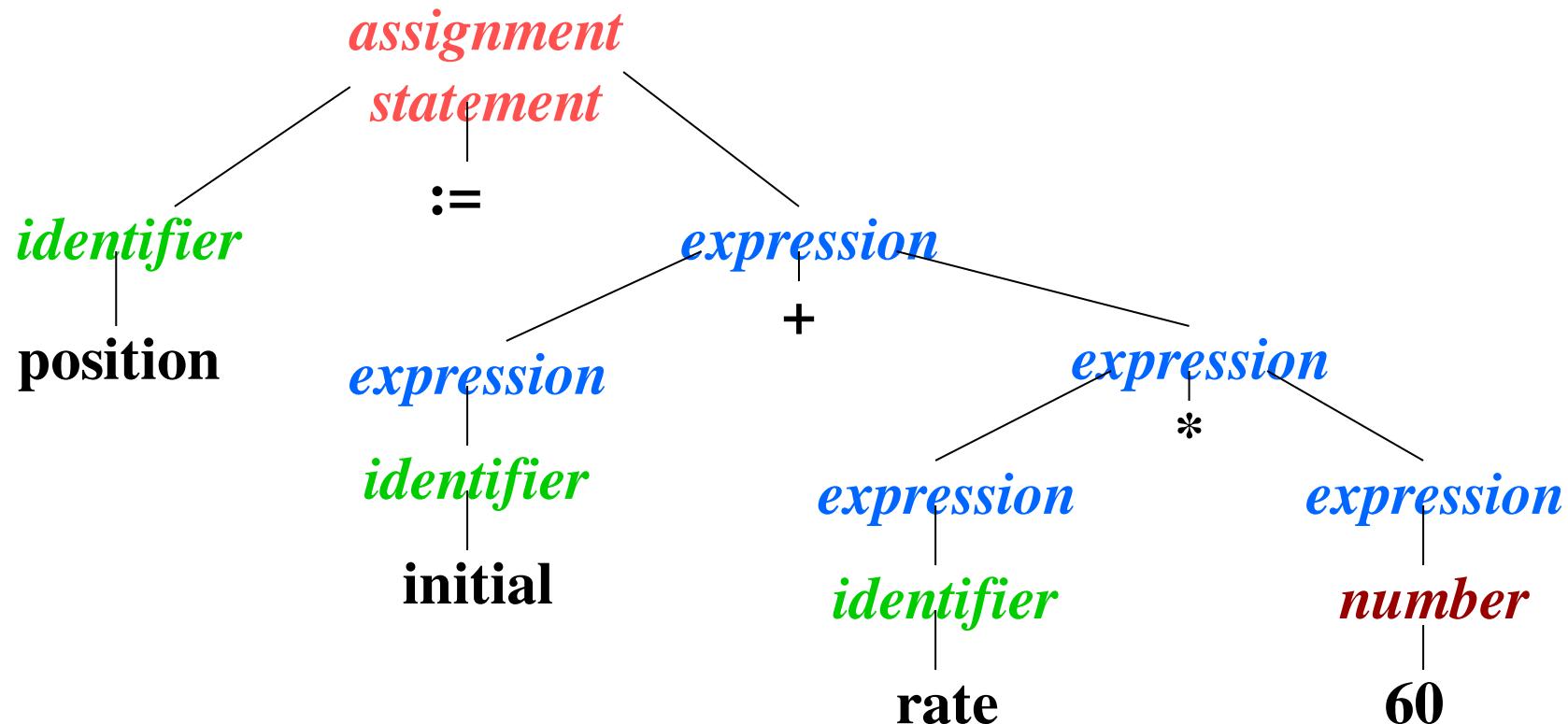
Syntax
Tree



Parser (Syntax Analysis)

Parse Tree for

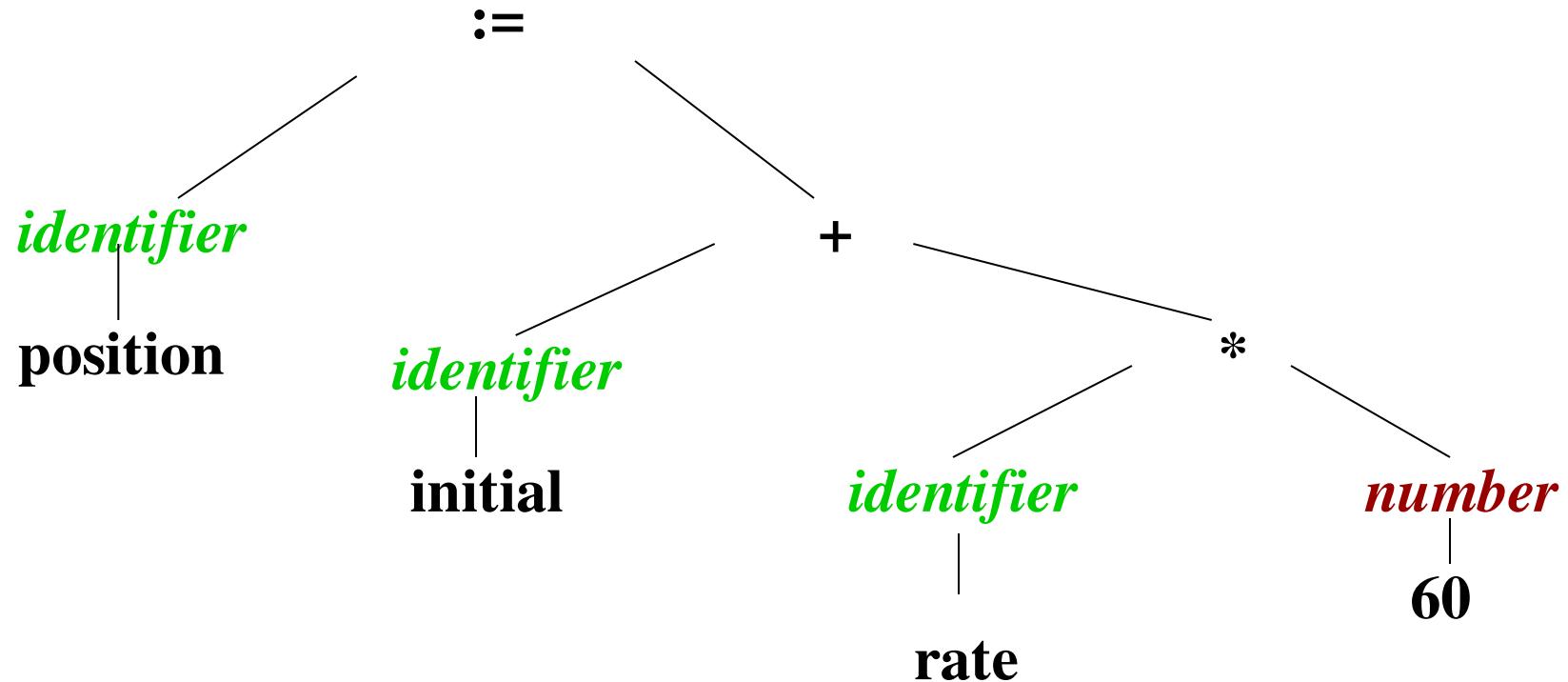
position := initial + rate * 60



Nodes of tree are constructed using a grammar for the language

Parser (Syntax Analysis)

Syntax Tree for **position := initial + rate * 60**



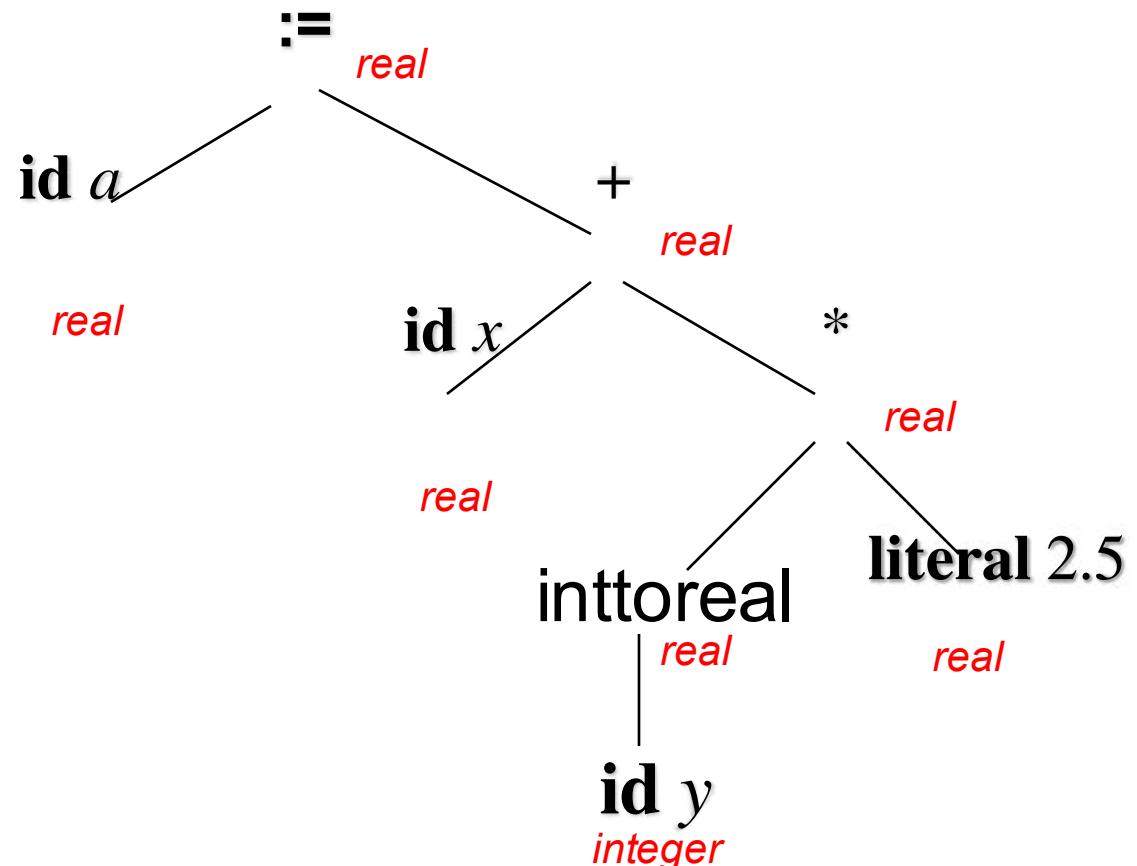
Semantic Analysis

- The semantic of a program is its **meaning** as opposed to structure
- Semantics consist of:
 - Runtime semantics – behavior of program at runtime
 - Static semantics – checked by the compiler
- Static semantics include:
 - Declarations of variables before use
 - Calling functions that exist
 - Passing parameters properly
 - Type checking
- The semantic analyzer does the following:
 - Checks the static semantics of the language
 - Annotates (decorates) the syntax tree with **type** information

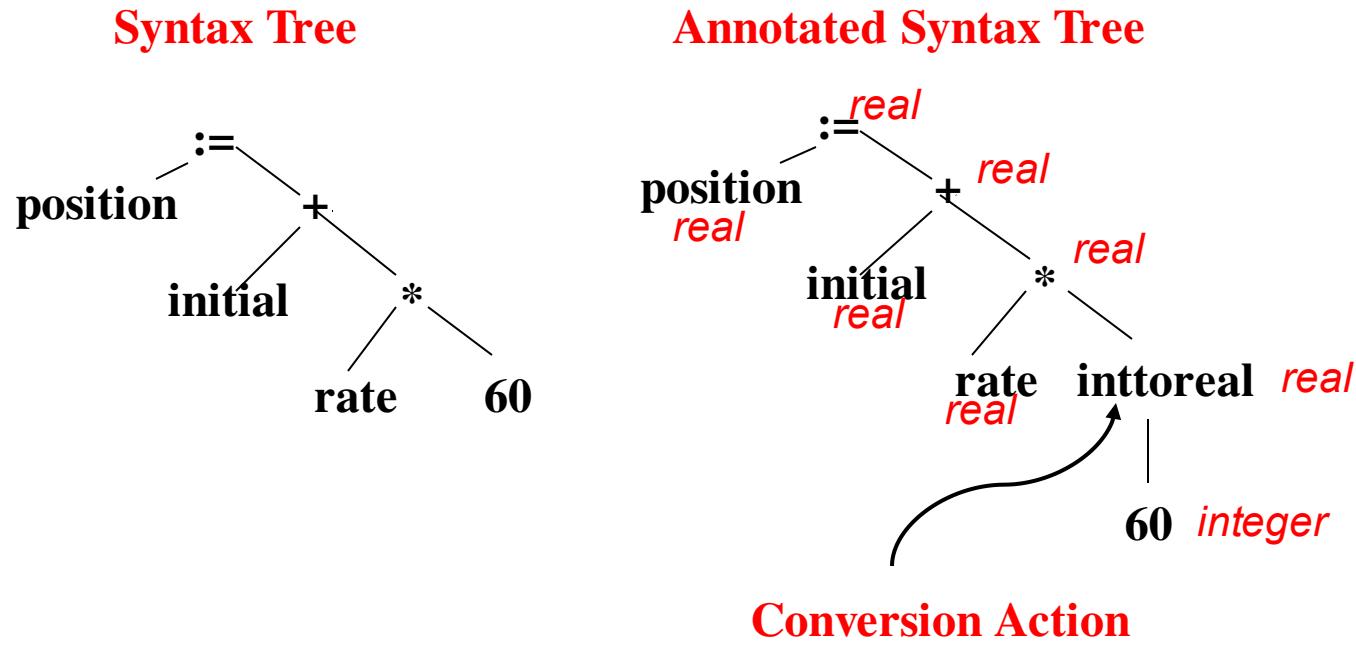
Semantic Analysis

a := x + y * 2.5 ;

Annotated Syntax Tree



Semantic Analysis

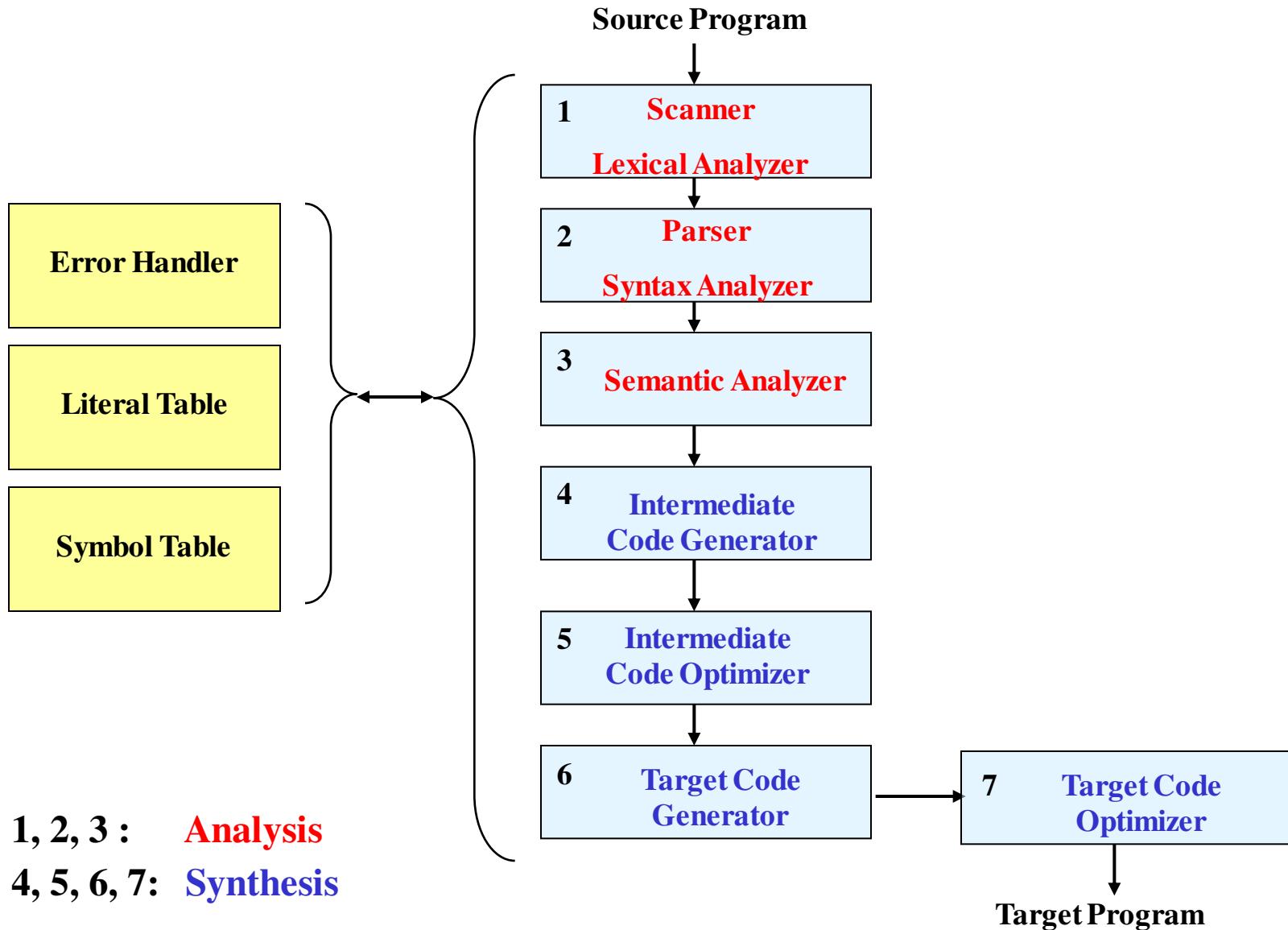


Semantic Analysis

- Most important activity in this phase:
 - Type Checking
 - Legality of Operands
- Many Different Situations:

```
Real := int + char ;  
  
A[int] := A[real] + int ;  
  
while char <> int do  
.... Etc.
```

The Phases of a Compiler



1, 2, 3 : Analysis
4, 5, 6, 7: Synthesis

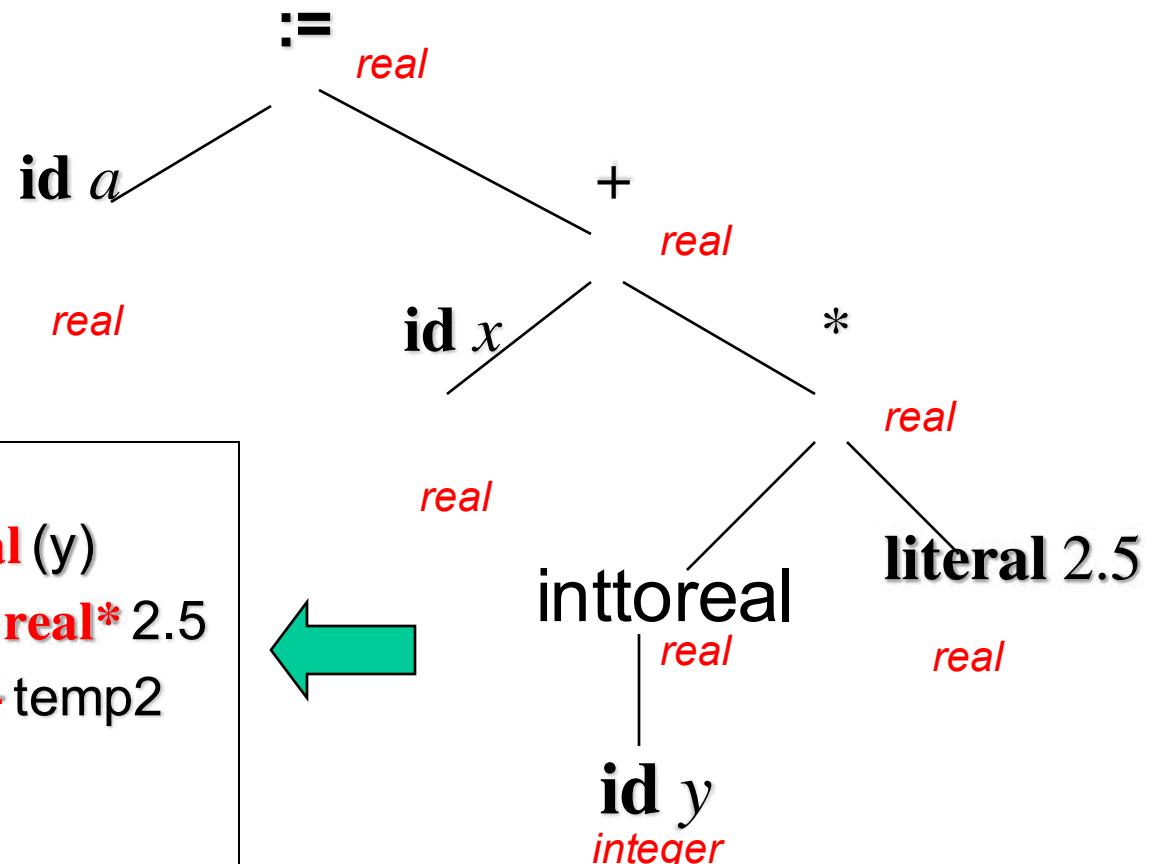
Intermediate Code Generator

- Comes after syntax and semantic analysis
- Separates the compiler **front end** from its **backend**
- Abstract machine version of code - Independent of machine Architecture
- Intermediate representation should have 2 important properties:
 - Should be easy to produce
 - Should be easy to translate into the target language
- Intermediate representation can have a variety of forms:
 - three-address code: address of (up to) 3 locations in memory
 - P-code: used in many Pascal compilers

Intermediate Code Generator

a := x + y * 2.5 ;

Annotated Syntax Tree

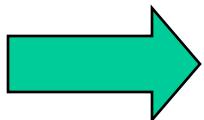


Code Generator

- Generates code for the target machine, typically:
 - Assembly code, or
 - Machine code
- Properties of the target machine become a major factor
- Code generator selects appropriate machine instructions
- Allocates memory locations for variables
- Allocates registers for intermediate computations

Code Generator

Three- address code
temp1 := **int2real** (y)
temp2 := temp1 **real*** 2.5
temp3 := x **real+** temp2
a := temp3



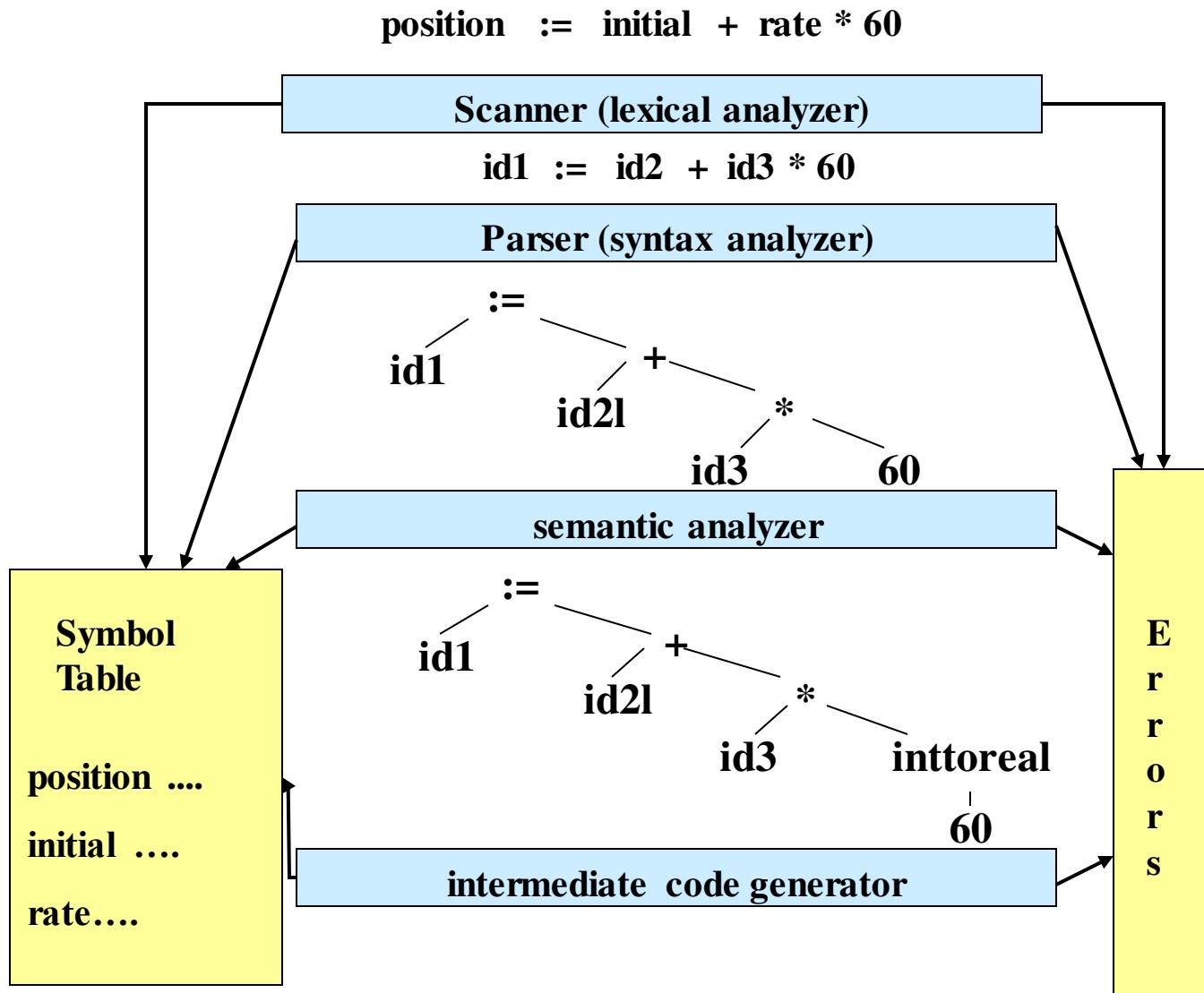
Assembly code (Hypothetical)

LOADI	R1 = [y] ;; R1 MEM[y]
CI2F	F1 = R1 ;; F1 int2real(y)
MOVF	F2 = 2.5 ;; F2 2.5
MULF	F3 = F1, F2 ;; F3 int2real(y) * 2. 5
LOADF	F4 = [x] ;; F4 MEM[x]
ADDF	F5 = F4, F3 ;; F5 x + int2real(y) * 2.5
STOREF	[a] = F5 ;; MEM[a] F5

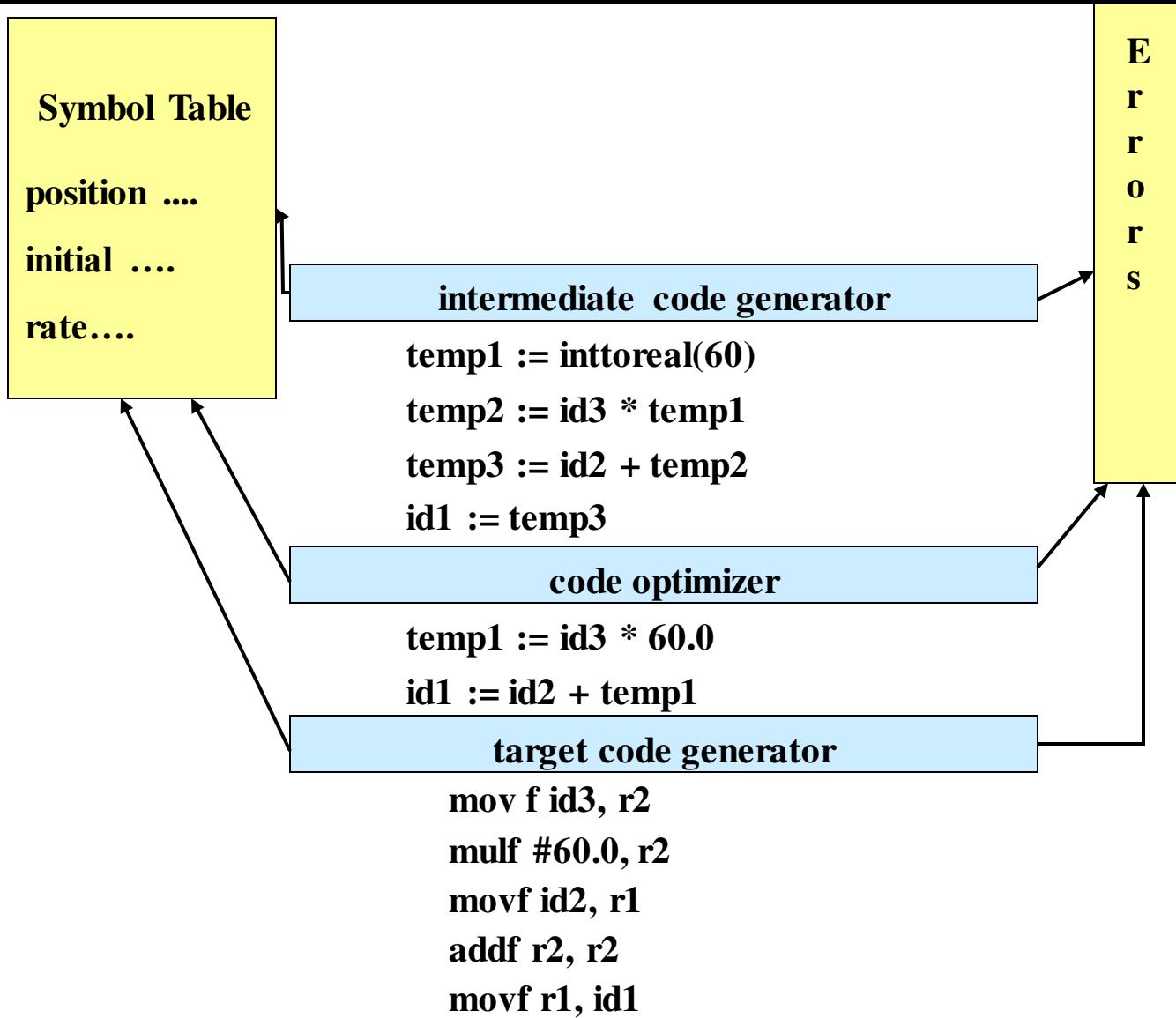
Code Optimization

- Code improvement techniques can be applied to:
 - Intermediate code – independent of the target machine
 - Target code – dependent on the target machine
 - Intermediate code improvement include:
 - Constant folding use 6 instead of $4 + 2$
 - Elimination of common sub-expressions
 - Identification and elimination of unreachable code (called dead code)
 - Improving loops
 - Improving function calls
 - Target code improvement include:
 - Allocation and use of registers
 - Selection of better (faster) instructions and addressing modes
 - Eliminating redundant or unnecessary operations

Reviewing the Entire Process



Reviewing the Entire Process



Programs Related to Compilers: Interpreter

- Is a program that reads a source program and executes it
- Works by analyzing and executing the source program commands one at a time
- Does not translate the source program into object code
- Well-known examples of interpreters
 - BASIC interpreter, LISP interpreter, UNIX shell interpreter, SQL interpreter
- In principle, any programming language can be either interpreted or compiled

Programs Related to Compilers

- **Preprocessor**

- Produces input to a compiler
 - Performs the following:
 - Macro processing (substitutions)
 - File inclusion
 - Delete comments

- **Assembler**

- Translator for the assembly language
 - Two-Pass Assembly:
 - All variables are allocated storage locations
 - Assembler code is translated into machine code
 - Output is relocatable machine code

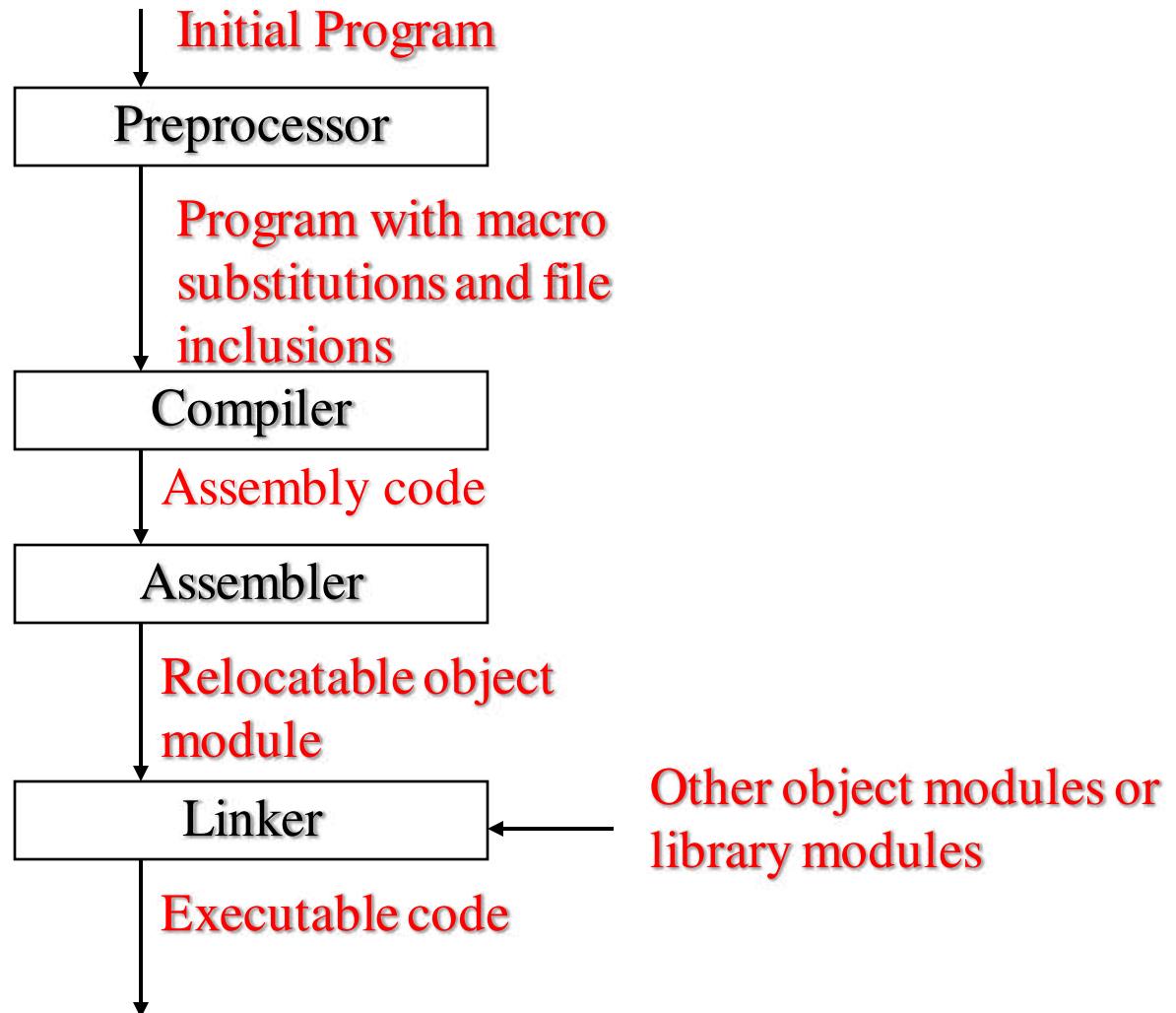
- **Linkers**

- Links object files separately compiled or assembled
 - Links object files to standard library functions
 - Generates a file that can be loaded and executed

- **Debuggers**

- **Editors**

Programs Related to Compilers



Major Data and Structures in a Compiler

- **Token**

- Represented by an integer value or an enumeration literal
 - Sometimes, it is necessary to preserve the string of characters that was scanned
 - For example, name of an identifier or value of a literal

- **Syntax Tree**

- Constructed as a pointer-based structure
 - Dynamically allocated as parsing proceeds
 - Nodes have fields containing information collected by the parser and semantic analyzer

Major Data and Structures in a Compiler

○ **Symbol Table**

- Keeps information associated with all kinds of identifiers:
- Variables, functions, parameters, types, fields, etc.
- Identifiers are entered by the scanner, parser, or semantic analyzer
- Semantic analyzer adds type information and other attributes
- Code generation and optimization phases use the information in the symbol table
- Insertion and search operations should be efficient because they are frequent
- Hash table with constant-time operations is usually the preferred choice

Major Data and Structures in a Compiler

○ Literal Table

- Stores constant values and string literals in a program
 - "Hello, world!", 3.141592653589793, etc.
- One literal table applies globally to the entire program
- Used by the code generator to:
 - Assign addresses to literals
 - Enter data definitions in the target code file
- Avoids the replication of constants and strings
- Quick insertion and lookup are essential
- Deletion is not necessary

○ Temporary Files

- Used historically by old compilers due to memory constraints
- Hold the data of various stages

Issues in Compiler Structure: Passes

- A compiler may traverse a program several times to generate code
- Each traversal is called a **pass**
- Passes may or may not correspond to phases
- A **one-pass** compiler: all phases occur during a single pass
 - Languages, such as C and Pascal, permit one-pass compilation
 - Compilation is efficient; a small chunk of the program is translated at a time
 - Resulting code is inefficient because optimizations require additional passes
- A **multi-pass** compiler holds transformed program between passes
 - Some languages require more than one pass to be translated properly
 - A first pass may construct the syntax tree from source program
 - A second pass may do semantic analysis
 - A third pass may generate code
 - Additional passes may do optimizations

Issues in Compiler Structure: Error Handling

- One of the more difficult parts of a compiler to design
- Error handler should:
 - Report the place of errors clearly and accurately
 - Must handle a wide range of errors & multiple errors
 - Recover from each error quickly enough to be able to detect subsequent errors (Must not get stuck)
 - Not significantly slow down the processing of correct programs
 - Must not get into an infinite loop (typical simple-minded strategy:count errors, stop if count gets too high)

Issues in Compiler Structure: Error Handling

- Errors can be:
 - Lexical: misspelling an identifier or keyword
 - Syntax: unbalanced parentheses
 - Semantic: operator used with an incompatible operand
 - run-time: divide-by-zero, infinitely recursive call
- Lexical:

```
Whil (x > 0) { x = x - 1; }
```
- Syntax:

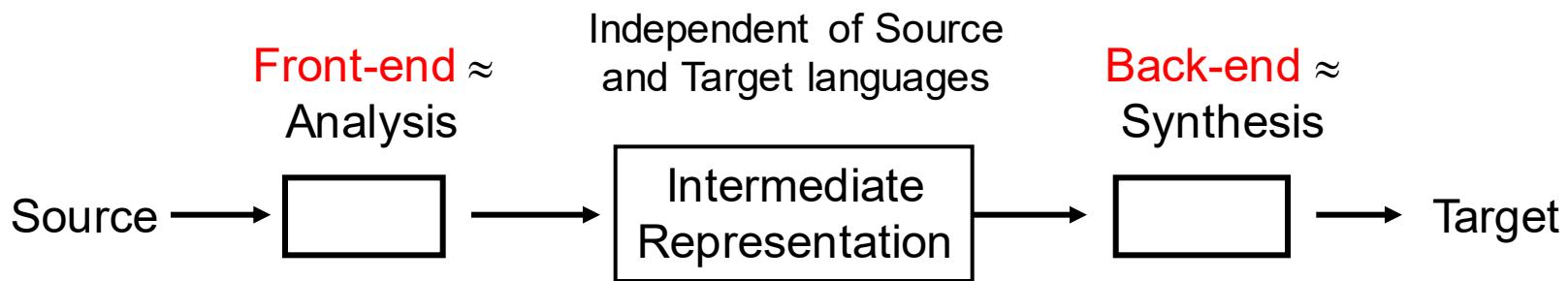
```
If (x == 0) y + = z + r; }
```
- Semantic:

```
int x = "Hello, world!";
```
- Runtime:

```
int x = 2;  
...  
double y = 3.14159 / (x - 2);
```

Issues in Compiler Structure: Front End & Back End

- Front end: Parser, scanner, semantic analyzer and source code optimizer depend primarily on source language.
- Back end: code generator and target code optimizer depend primarily on target language (machine architecture).



Ideally:

For a new Source language – we can add a new front-end to an existing back-end
For a new Target language – we can add a new back-end to an existing front-end

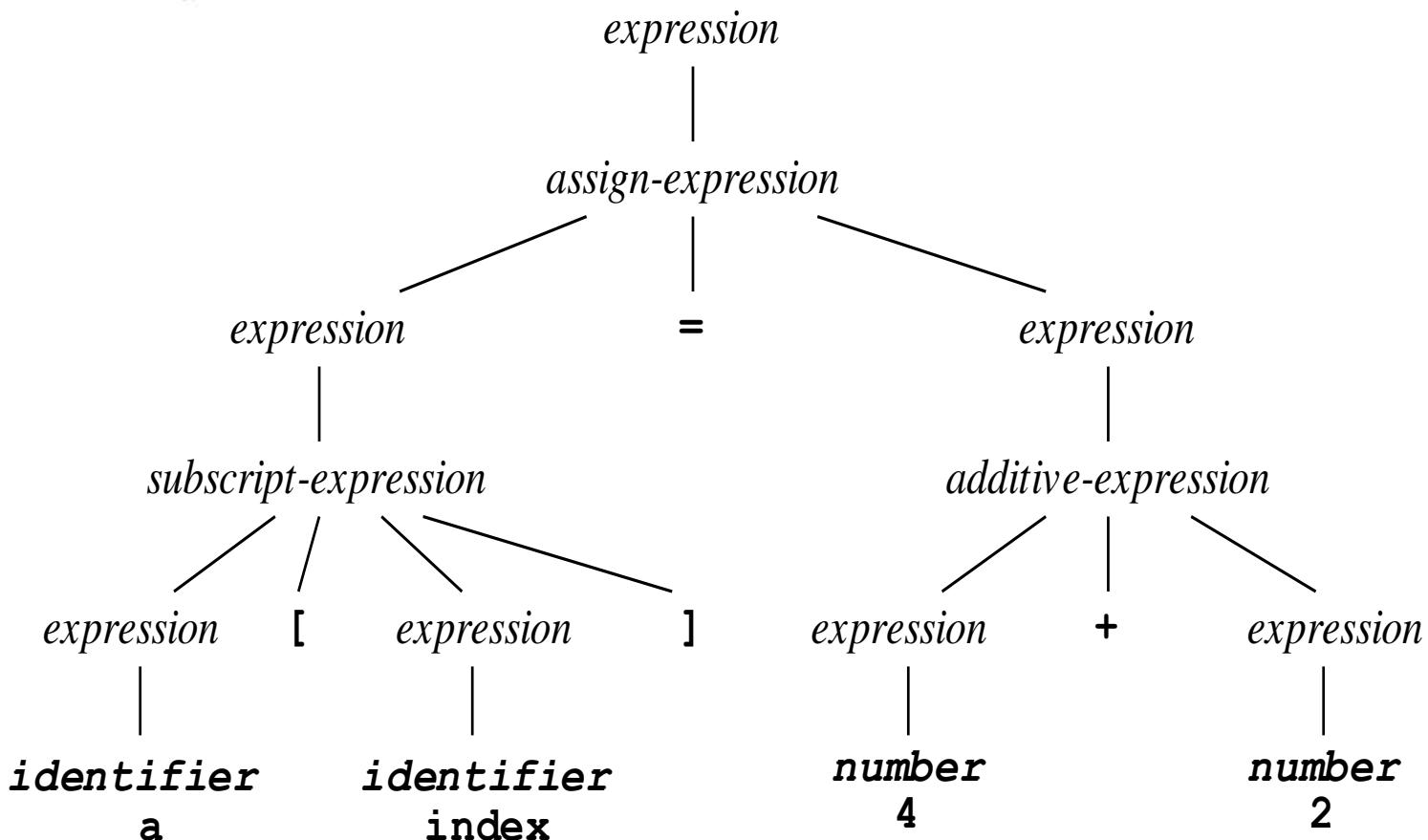
Extended Example

Source code: a [index] = 4 + 2

Tokens: ID Lbracket ID Rbracket
 AssignOp Num AddOp Num

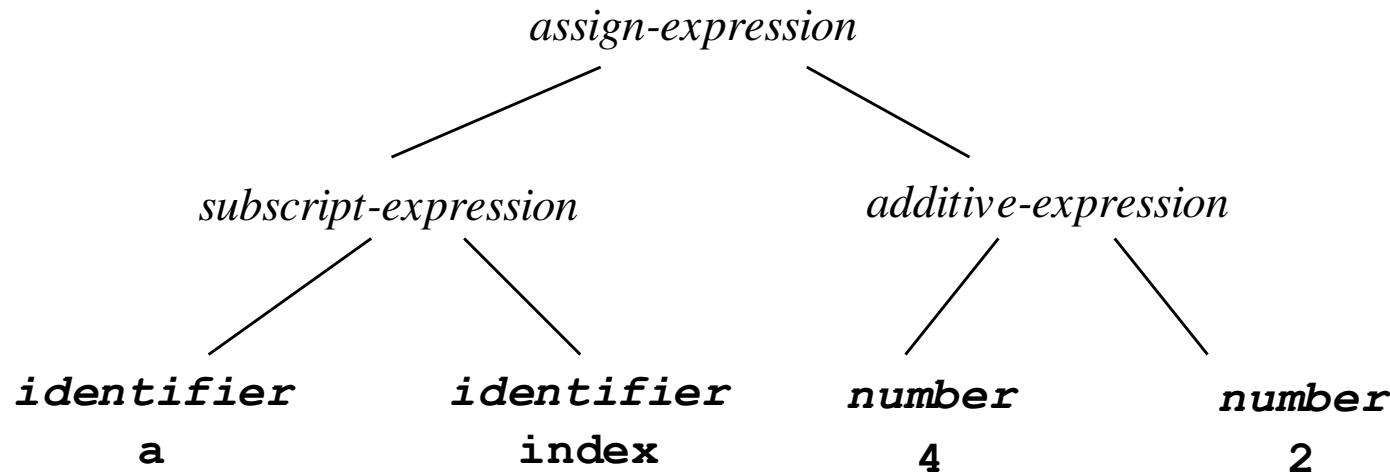
Parse tree

- Parse tree (syntax tree with all steps of the parser in gory detail):



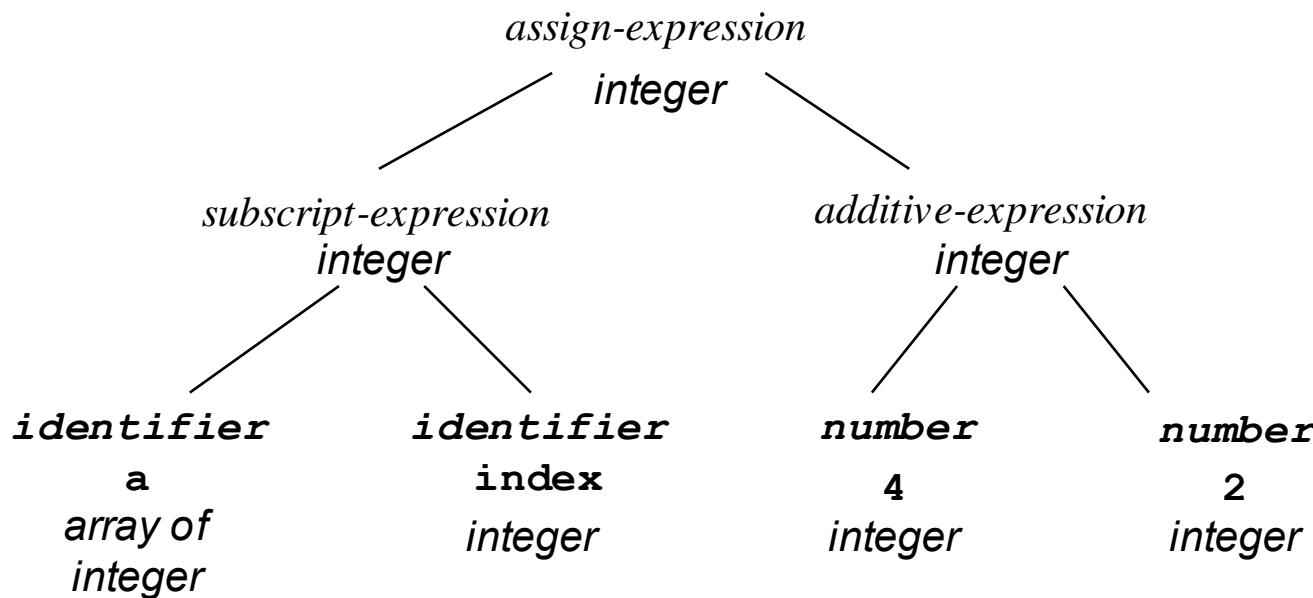
Syntax tree

- Syntax tree: a "trimmed" version of the parse tree with only essential information:



Annotated tree

- Annotated tree with attributes:



Target code

- Target code (edited & modified for this presentation):

```
mov eax, 6
mov ecx, DWORD PTR _index$[ebp]
mov DWORD PTR _a$[ebp+ecx*4], eax
```

(Note source level constant folding optimization.)

Sample compilers in this class ("Toys")

- TINY: a 4-pass compiler for the TINY language, based on Pascal
- C-Minus: your project language. Based on C

A Sample TINY Program

- Calculating the factorial of an input number x

{Sample program in TINY language – Factorial}

```
write "Enter an integer value: ";
read x;
factorial := 1;
count := x;
while count > 1 do
    factorial := factorial * count;
    count := count - 1;
end;
write "factorial of " , x , " = " , factorial;
```

C-Minus Example

```
int fact( int x )
{
    if (x > 1)
        return x * fact(x-1);
    else
        return 1;
}
```

```
void main( void )
{
    int x;
    x = read();
    if (x > 0) write( fact(x) );
}
```

Structure of the TINY Compiler

globals.h	main.c
util.h	util.c
scan.h	scan.c
parse.h	parse.c
symtab.h	symtab.c
analyze.h	analyze.c
code.h	code.c
cgen.h	cgen.c

Conditional Compilation Options

- **NO_PARSE:**
 - Builds a scanner-only compiler.
- **NO_ANALYZE:**
 - Builds a compiler that parses and scans only.
- **NO_CODE:**
 - Builds a compiler that performs semantic analysis, but generates no code.

Listing Options

- **EchoSource:**
 - Echoes the TINY source program to the listing, together with line numbers.
- **TraceScan:**
 - Displays information on each token as the scanner recognizes it.
- **TraceParse:**
 - Displays the syntax tree in a linearized format.
- **TraceAnalyze:**
 - Displays summary information on the symbol table and type checking.
- **TraceCode:**
 - Prints code generation-tracing comments to the code file.

Compilers Construction

Chapter 2

Lexical Analysis (Scanning)

Dr. Mohammed Salem Atoum

M.Atoum@inu.edu.jo

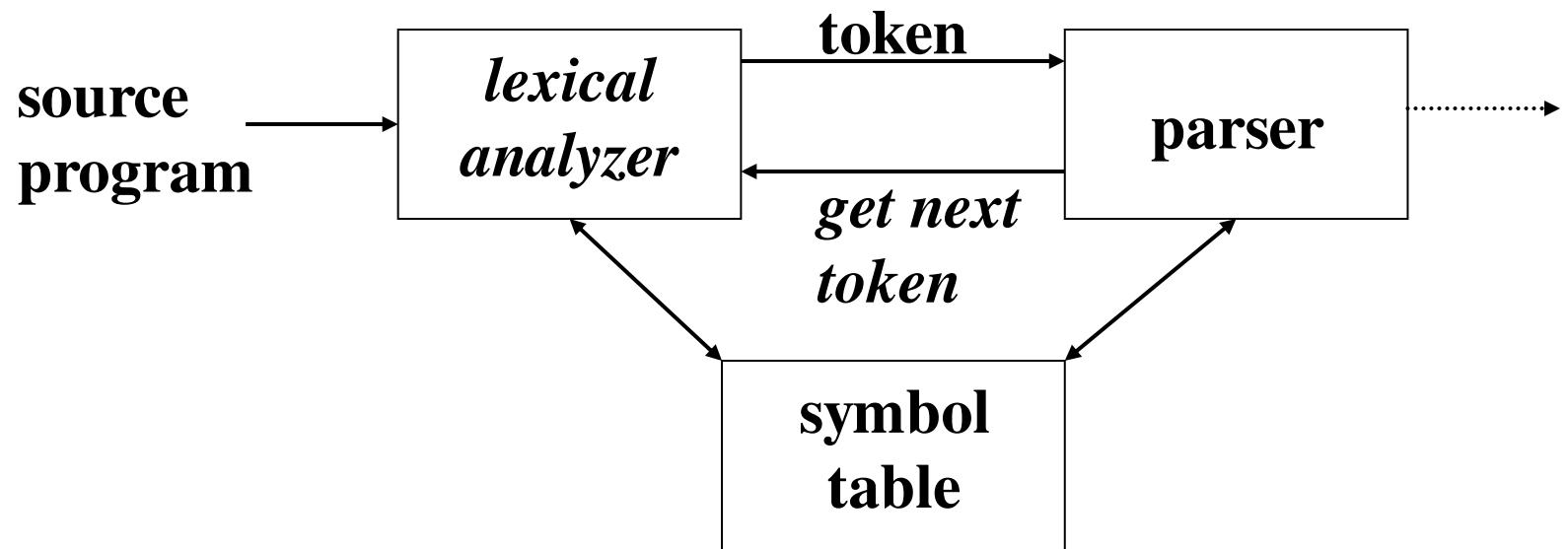
Outline

- Basic Concepts & Regular Expressions
 - What does a Lexical Analyzer do?
 - How does it Work?
 - Formalizing Token Definition & Recognition
- Reviewing Finite Automata Concepts
 - Non-Deterministic and Deterministic FA
 - Conversion Process
 - Regular Expressions to NFA
 - NFA to DFA
- Relating NFAs/DFAs /Conversion to Lexical Analysis
- Concluding Remarks /Looking Ahead

The Role of Lexical Analyzer

- The function of a scanner, called also **lexical analyzer**, is to:
 - Read characters from the source file
 - Group input characters into meaningful units, called **tokens**
- The scanner takes care of other things as well:
 - Removal of comments and white space
 - Keeping track of current line number and character position
 - Required for reporting error messages
 - Case conversions of identifiers and keywords
 - Simplifies searching if the language is not case- sensitive
 - Communication with the symbol or literal table
 - Identifiers can be entered in the symbol table
 - String literals can be entered in the literal table

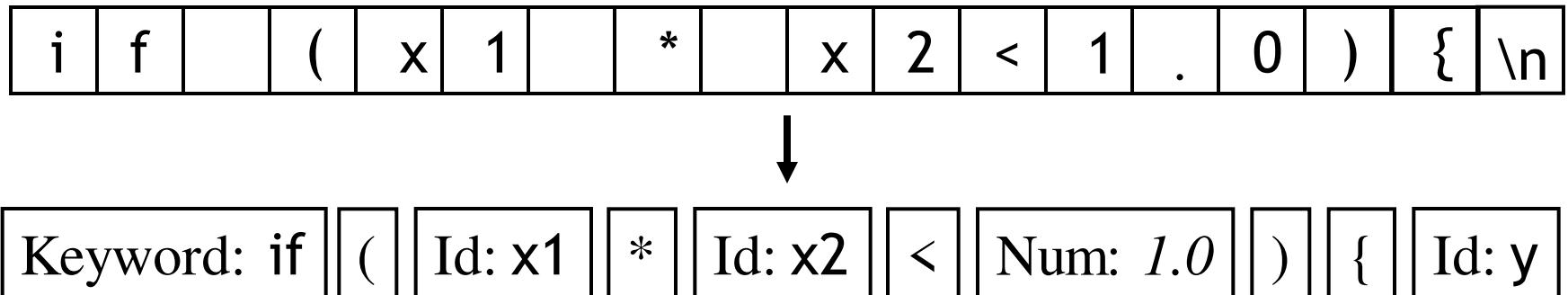
The Role of Lexical Analyzer



The Role of Lexical Analyzer

- Converts character stream to token stream

```
if (x1 * x2<1.0){  
    y = x1;  
}
```



Tokens

- Identifiers: x y11 elsex _i00
- Keywords: if else while break
- Integers: 2 1000 -500 5L
- Floating point: 2.0 0.00020 .02 1. 1e5
- Symbols: + * { } ++ < << [] >=
- Strings: “x” “He said, \“Are you?\””
- Comments: /** comment **/

Introducing Basic Terminology

- What are Major Terms for Lexical Analysis?
 - TOKEN
 - A classification for a common set of strings
 - Examples Include <Identifier>, <number>, etc.
 - LEXEME
 - Actual sequence of characters that matches pattern and is classified by a token
 - Identifiers: x, count, name, etc...
 - x, count, and name are lexemes of type Identifier
 - PATTERN
 - The rules which characterize the set of strings for a token

Introducing Basic Terminology

Token	Sample Lexemes	Informal Description of Pattern
const	const	const
if	if	if
relation	<, <=, =, < >, >, >=	< or <= or = or < > or >= or >
id	pi, <u>count</u> , <u>D2</u>	letter followed by letters and digits
num	<u>3.1416</u> , <u>0</u> , <u>6.02E23</u>	any numeric constant
literal	“core dumped”	any characters between “ and “ except ““

Example

```
function gcd(m, n : integer) : integer;  
begin  
    if n = 0 then gcd := m  
        else gcd := gcd(n, m mod n)  
end; (* of gcd *)
```

<u>Tokens</u>	<u>Lexemes</u>
keyword	function, begin, if, then, else, end
Identifier	gcd, m, n, mod, integer
constant	0
delimiter	(:) ; ,
relop	=
assignop	:=
comment	(*of gcd*)

Specification Of Tokens

- How to describe tokens
`2.e0 20.e-01 2.0000`
- How to break text up into tokens
`if (x == 0) a = x<<1;`
`iff (x == 0) a = x<1;`
- Programming language tokens can be described as **regular expressions**
- Regular expressions are an important notation for specifying patterns.

Specification Of Tokens

- An **alphabet** is a finite set of characters, denoted by the Greek symbol Σ (sigma)
 - The alphabet can be the ASCII set or a subset of ASCII
 - Examples:
 - $\Sigma = \{0, 1\}$: binary alphabet
 - $\Sigma = \{a, b, c, \dots, z\}$: the set of all lower case letters
 - The set of all ASCII characters
- A **string** over some alphabet is a sequence of symbols drawn from the alphabet Σ
 - Example: 01001 is string over the alphabet $\{0, 1\}$
 - Examples:
 - 01101 where $\Sigma = \{0, 1\}$
 - *abracadabra* where $\Sigma = \{a, b, c, \dots, z\}$

Specification Of Tokens

- The empty or **null string** ϵ is a special string of length zero
 - When ϵ is concatenated with any string s yields s . That is, $s \epsilon = \epsilon s = s$
- **Length of String:** Number of symbols in the string
 - The length of a string w is usually written $|w|$
 - $|1010| = 4$
 - $|\epsilon| = 0$
 - $|uv| = |u| + |v|$
- **Reverse** : w^R
 - If $w = abc$, $w^R = cba$

Specification Of Tokens

- **Concatenation:** if x and y are strings, then xy is the string obtained by placing a copy of y immediately after a copy of x
 - $x = a_1a_2 \dots a_i, \quad y = b_1b_2 \dots b_j$
 - $xy = a_1a_2 \dots a_i b_1b_2 \dots b_j$
 - Example: $x = 01101, \quad y = 110, \quad xy = 01101110$
 - $x\varepsilon = \varepsilon x = x$

Specification Of Tokens

- **Power of an Alphabet:** Σ^k = the set of strings of length k with symbols from Σ
 - Example: $\Sigma = \{0, 1\}$
 - $\Sigma^1 = \Sigma = \{0, 1\}$
 - $\Sigma^2 = \{00, 01, 10, 11\}$
 - $\Sigma^0 = \{\varepsilon\}$
- **Question:** How many strings are there in Σ^3 ?
- The set of all strings over Σ is denoted Σ^*
 - $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$
- Also
 - $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$
 - $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$
 - $\Sigma^+ = \Sigma^* - \{\varepsilon\}$

Specification Of Tokens

- **Substring:** any string of consecutive characters in some string w
 - If $w = abc$
 - ϵ, a, ab, abc are substrings of w
- **Prefix and suffix:**
 - if $w = vu$
 - v is a prefix of w
 - u is a suffix of w
 - Example
 - If $w = abc$
 - a, ab, abc are prefixes of w
 - c, bc, abc are suffixes of w

Strings and Languages (6)

Suppose: **S** is the string **banana**

Prefix : ban, banana

Suffix : ana, banana

Substring : nan, ban, ana, banana

Proper prefix, suffix,
or substring cannot
be all of **S**

Language Concepts

A language, L, is simply any set of strings over a fixed alphabet.

Alphabet

{0,1}

{a,b,c}

{A, ..., Z}

{A,...,Z,a,...,z,0,...9,
+,-,...,<,>,...}

Languages

{0,10,100,1000,100000...}

{0,1,00,11,000,111,...}

{abc,aabbcc,aaabbbccc,...}

{TEE,FORE,BALL,...}

{FOR,WHILE,GOTO,...}

{ All legal PASCAL progs}
{ All grammatically correct
English sentences }

Special Languages: \emptyset - EMPTY LANGUAGE
 \in - contains \in string only

Specification Of Tokens

- A **formal language** is a set of strings (possibly infinite) over some alphabet
 - Example: $L = \{00, 01, 10, 11\}$ is the set of all 2- character strings over $\Sigma = \{0, 1\}$
- A language is a subset of Σ^*
 - Example of languages:
 - The set of valid Arabic words
 - The set of legal C programs
 - The set of strings consisting of n 0's followed by n 1's
 - $\{\epsilon, 01, 0011, 000111, \dots\}$
 - The set of strings with equal number of 0's and 1's
 - $\{\epsilon, 01, 10, 0011, 0101, 1010, 1001, 1100, \dots\}$
- **Empty language:** $\emptyset = \{ \ }$
- The language $\{\epsilon\}$ consisting of the empty string
- Note: $\emptyset \neq \{\epsilon\}$

Finite Specification of Languages (1)

- The **concatenation** of two languages L_1 and L_2 (sets of strings) is:
 - Obtained by concatenating every string in L_1 with every string in L_2
 - $L = L_1 L_2 = \{ s = s_1 s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \}$
 - $L^n = L$ concatenated with itself n times
 - $L^0 = \{\varepsilon\}; L^1 = L$
 - $L_1 = \{\varepsilon, 0, 00\}, L_2 = \{\varepsilon, 1, 11\}, L = L_1 L_2 = \{\varepsilon, 1, 11, 0, 01, 011, 00, 001, 0011\}$
- The **exponentiation** of a language is defined as follows:
 - $L^0 = \{\varepsilon\}$ and $L^i = L^{i-1} L$
 - $L = \{0, 1\}, L^0 = \{\varepsilon\}, L^1 = L = \{0, 1\}$
 - $L^2 = \{00, 01, 10, 11\}$ and $L^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$
- The **Kleene closure** of a language L , denoted as L^* , is defined as:
 - $L^* = L^0 \cup L^1 \cup L^2 \cup \dots$
 - $L = \{0, 1\}, L^* = \{ \ , 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, \dots \}$

Finite Specification of Languages (2)

- The **Positive closure** of a language L , denoted as L^+ , is defined as:
 - $L^+ = L^* - L^0$
 - $L = \{0, 1\}$, $L^+ = \{0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, \dots\}$
- Languages can be finite or infinite
 - $L = \{a, aba, bba\}$
 - $L = \{a^n \mid n > 0\}$

Formal Language Operations

OPERATION	DEFINITION
<i>union of L and M</i> written $L \cup M$	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
<i>concatenation of L and M</i> written LM	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
<i>Kleene closure of L</i> written L^*	$L^* = \cup L^i, i=0, \dots, \infty$ L^* denotes “zero or more concatenations of “ L
<i>positive closure of L</i> written L^+	$L^+ = \cup L^i, i=1, \dots, \infty$ L^+ denotes “one or more concatenations of “ L $L^+ = LL^*$

Formal Language Operations Examples

$$L = \{A, B, \dots, Z, a, b, \dots z\} \quad D = \{1, 2, \dots, 9\}$$

$L \cup D$ = the set of letters and digits

LD = all strings consisting of a letter followed by a digit

L^2 = the set of all two-letter strings

$L^4 = L^2 \ L^2$ = the set of all four-letter strings

$L^* = \{ \text{All possible strings of } L \text{ plus } \in \}, \ L^+ = L^* - \in$

D^+ = set of strings of one or more digits

$L(L \cup D)^*$ = set of all strings consisting of a letter followed by a letter or a digit

$L(L \cup D)^*$ = set of all strings consisting of letters and digits beginning with a letter

تمييز Recognition of Tokens

How can we use concepts developed so far to assist in recognizing tokens of a source language ?

Given Tokens, What are Patterns?

$stmt \rightarrow \text{if } expr \text{ then } stmt$

| $\text{if } expr \text{ then } stmt \text{ else } stmt$

| ϵ

$expr \rightarrow term \text{ relop } term$

| $term$

$term \rightarrow \text{id} \mid \text{num}$

Recognition of Tokens

if → if

then → then

else → else

relop → < | <= | > | >= | = | <>

id → letter (letter | digit)*

num → digit+ (. digit+)? (E(+ | -)? digit+)?

letter → [A-Za-z]

digit → [0-9]

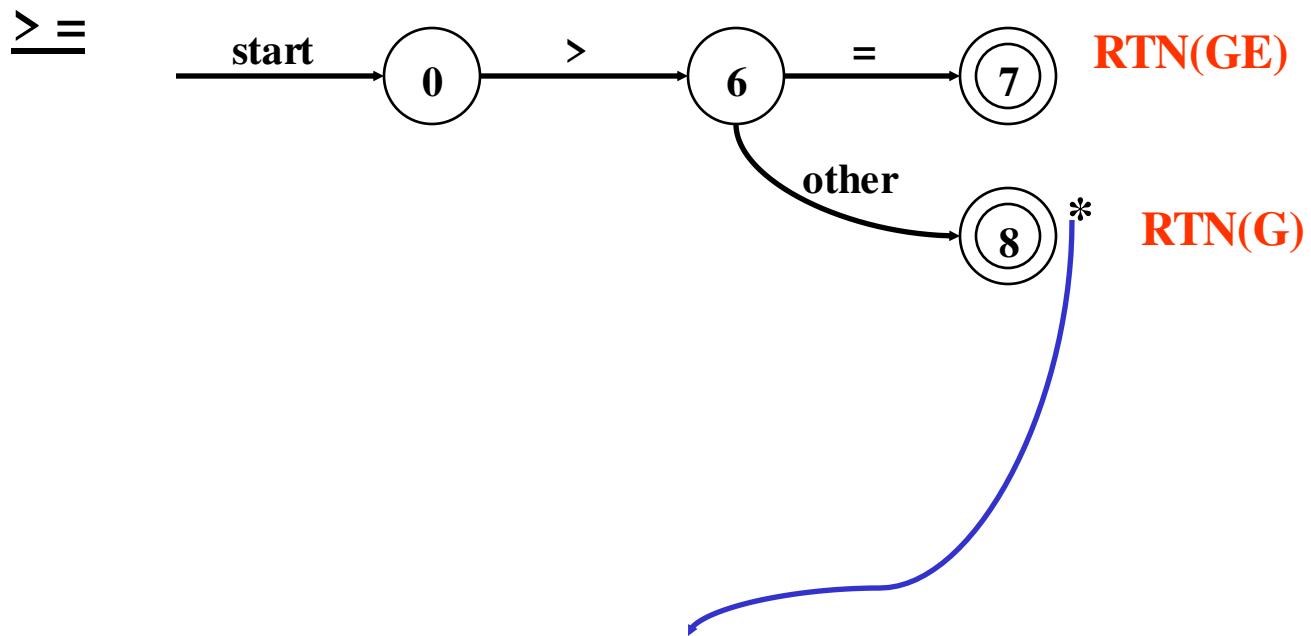
delim → blank | tab | newline

ws → delim⁺

Constructing Transition Diagrams for Tokens

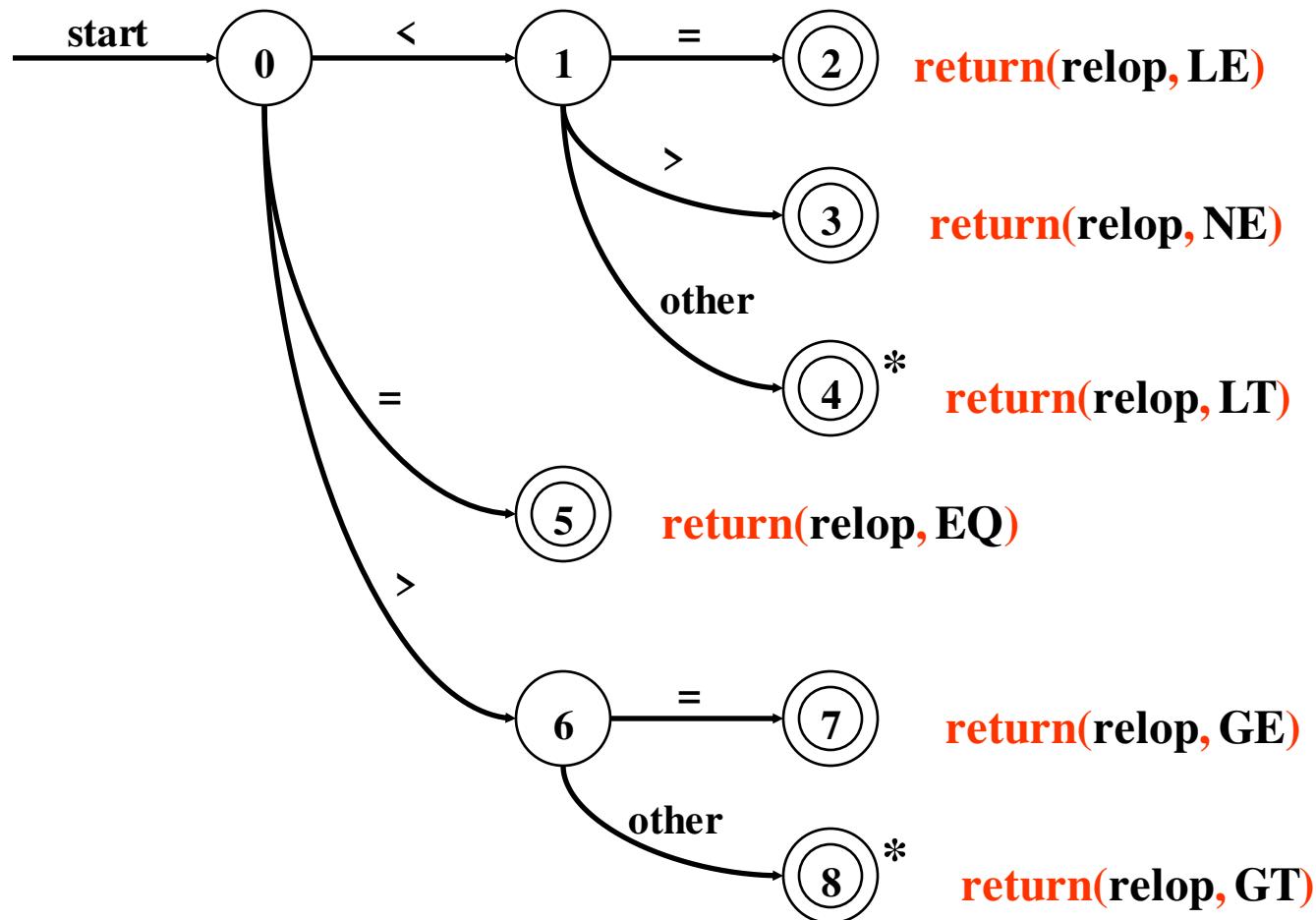
- Transition Diagrams (TD) are used to represent the tokens
- As characters are read, the relevant TDs are used to attempt to match lexeme to a pattern
- Each TD has:
 - States : Represented by Circles
 - Actions : Represented by Arrows between states
 - Start State : Beginning of a pattern (Arrowhead)
 - Final State(s) : End of pattern (Concentric Circles)
- Each TD is Deterministic - No need to choose between two different actions !

Example TDs



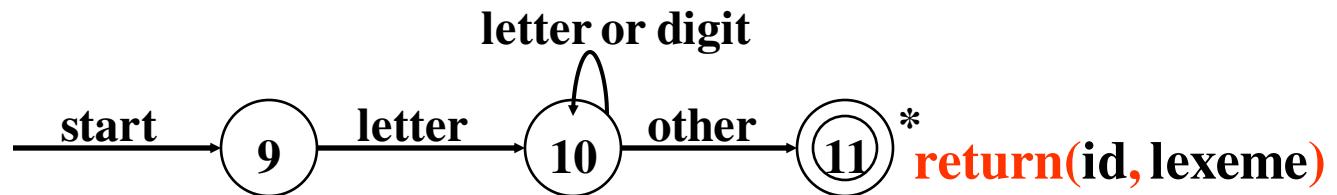
We've accepted “>” and have read other char that must be unread.

Example : All RELOPs

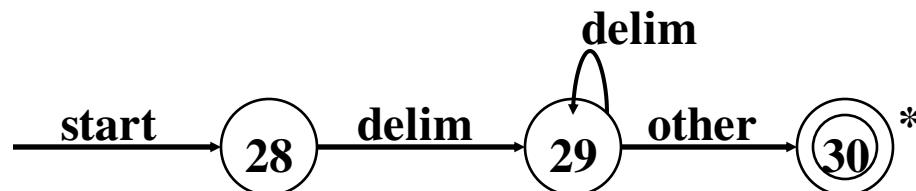


Example TDs : id and delim

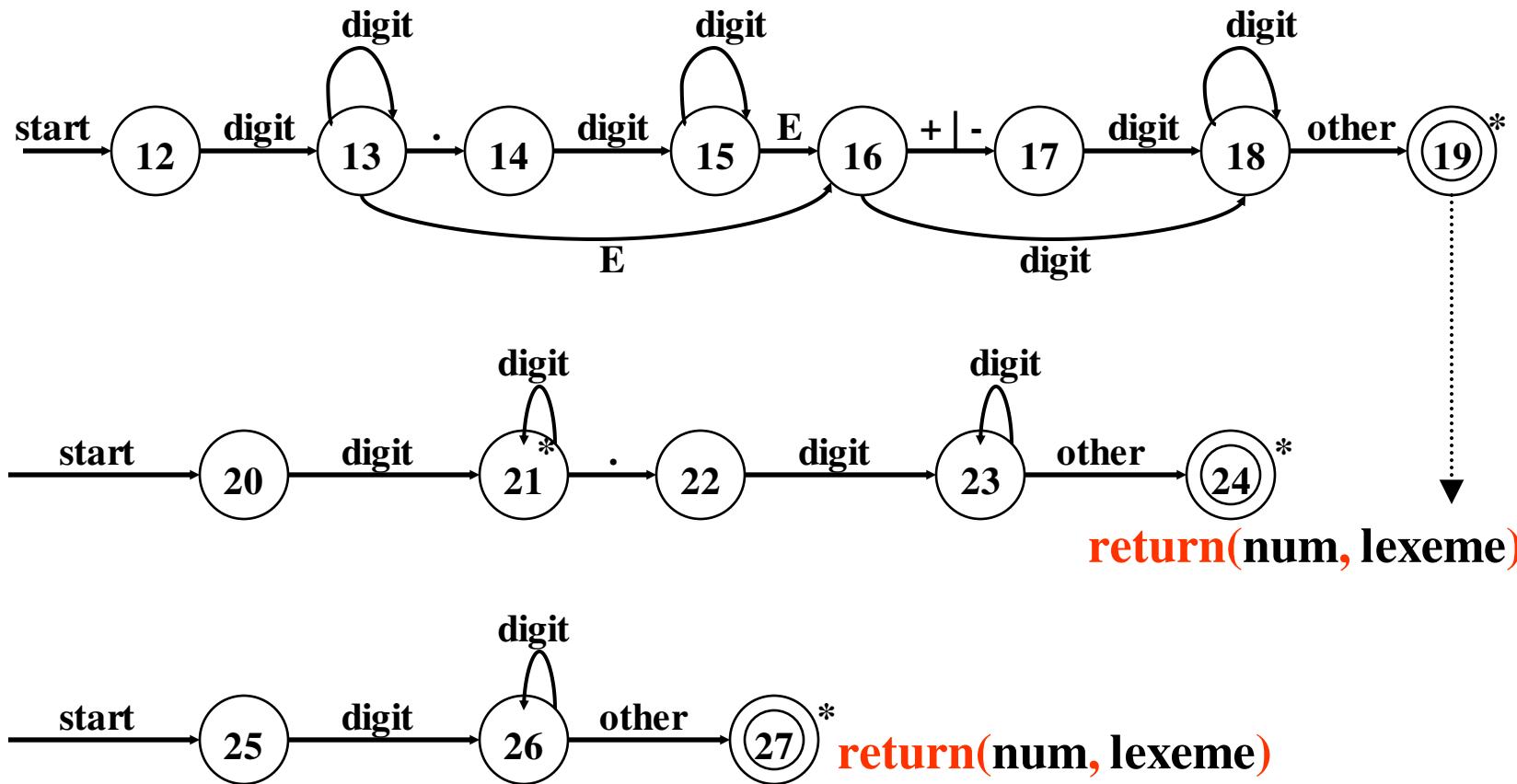
id



delim



Example TDs : Unsigned #s



Questions: Is ordering important for unsigned #s ?

Finite Automata

Finite Automata : A recognizer that takes an input string & determines whether it's a valid sentence of the language

Non-Deterministic : Has more than one alternative action for the same input symbol.

Deterministic : Has at most one action for a given input symbol.

Both types are used to recognize regular expressions.

Non-Deterministic Finite Automata

An **NFA** is a mathematical model that consists of :

- S , a set of **states**
- Σ , the symbols of the **input alphabet**
- *move*, a **transition function.**
 - $move(state, symbol) \rightarrow state$
 - $move : S \times \Sigma \rightarrow S$
- A state, $s_0 \in S$, the **start state**
- $F \subseteq S$, a set of **final or accepting states.**

Representing NFAs

Transition Diagrams :

Number states (circles),
arcs, final states, ...

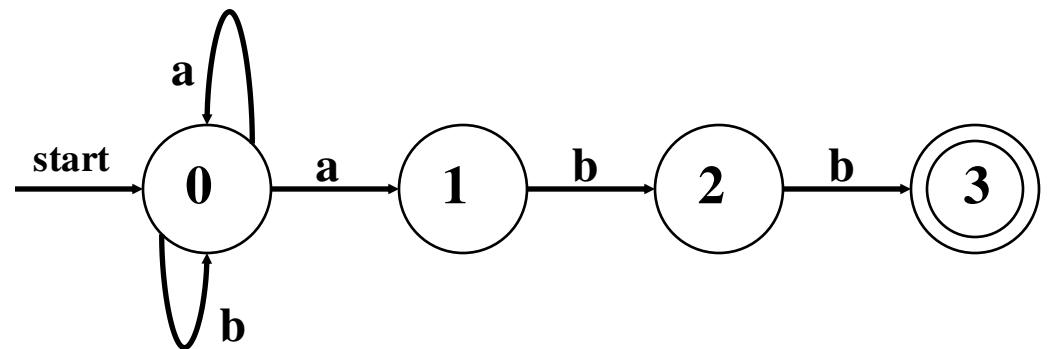
Transition Tables:

More suitable to
representation within a
computer

We'll see examples of both !

Example NFA

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



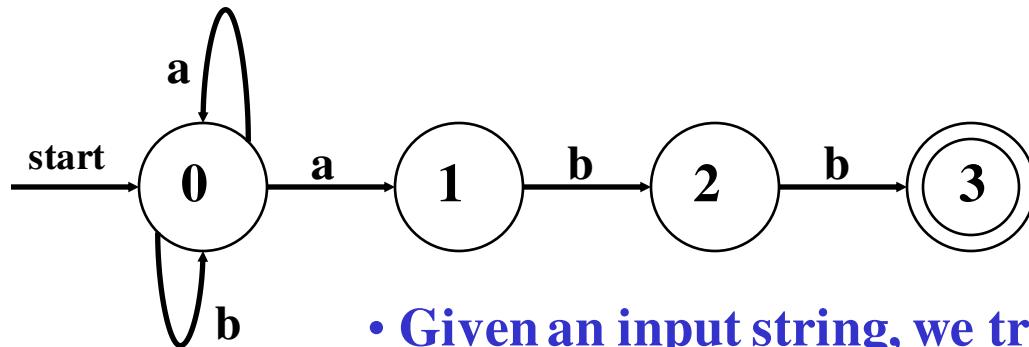
What Language is defined ?

What is the Transition Table ?

i n p u t

	a	b
s		
t	0	{ 0, 1 }
a	1	--
t	2	--
e		{ 3 }

How Does An NFA Work ?



- Given an input string, we trace moves
- If no more input & in final state, ACCEPT

EXAMPLE:

Input: ababb

$$move(0, a) = 1$$

$$move(1, b) = 2$$

$$move(2, a) = ? \text{ (undefined)}$$

REJECT !

-OR-

$$move(0, a) = 0$$

$$move(0, b) = 0$$

$$move(0, a) = 1$$

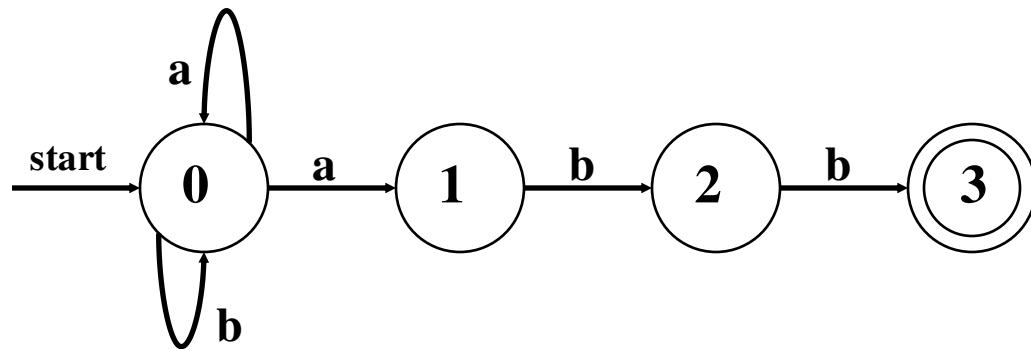
$$move(1, b) = 2$$

$$move(2, b) = 3 \text{ (final state)}$$

ACCEPT !

Other Concepts

Not all paths may result in acceptance.



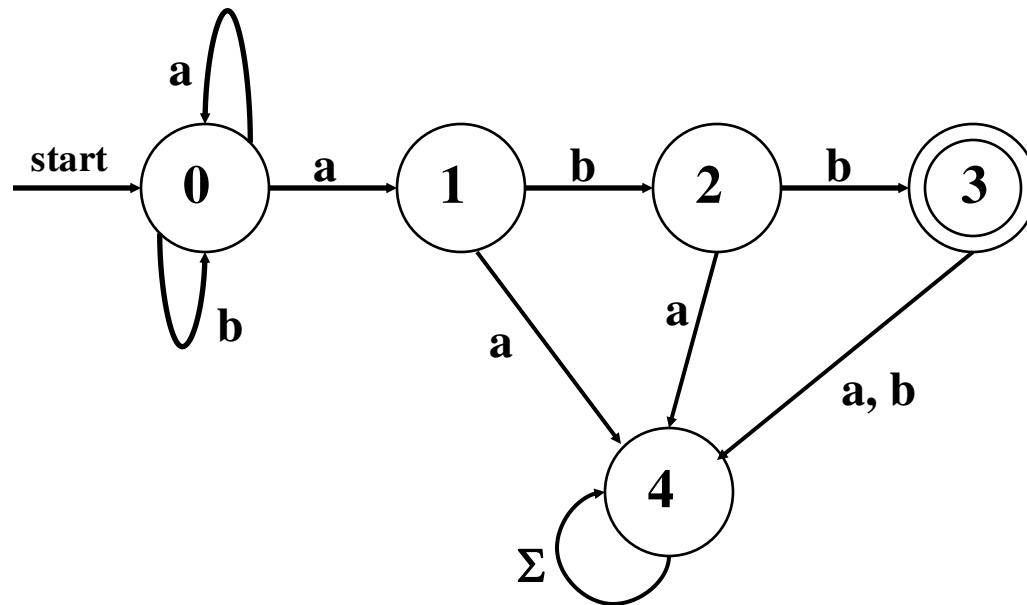
aabb is accepted along path : $0 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 3$

BUT... it is not accepted along the valid path:

$0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0$

Handling Undefined Transitions

We can handle undefined transitions by defining one more state, a “death” state, and transitioning all previously undefined transition to this death state



Second NFA Example

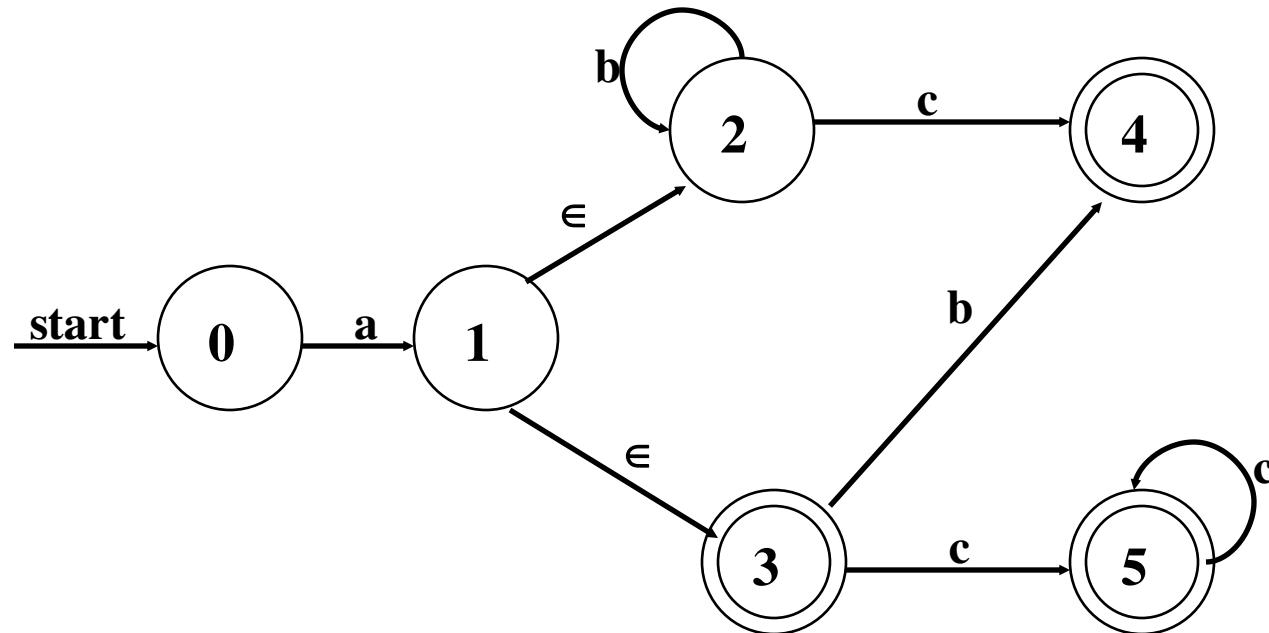
Given the regular expression : $a(b^*c) \mid a(b \mid c^+)$?

Find a transition diagram NFA that recognizes it.

Second NFA Example - Solution

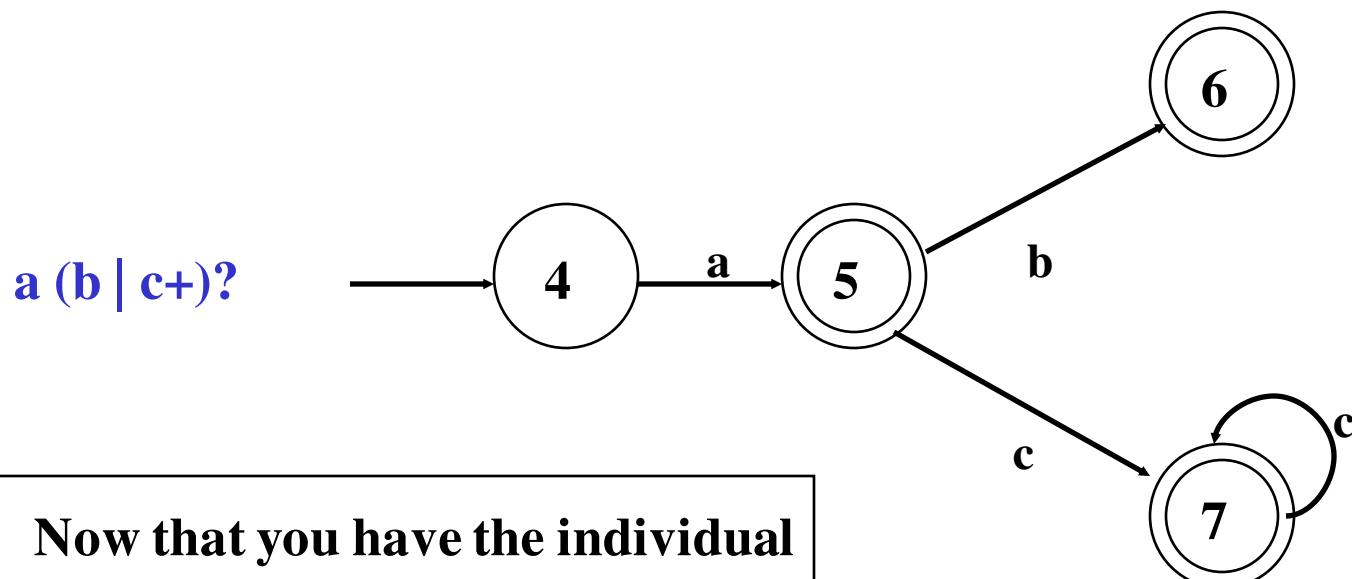
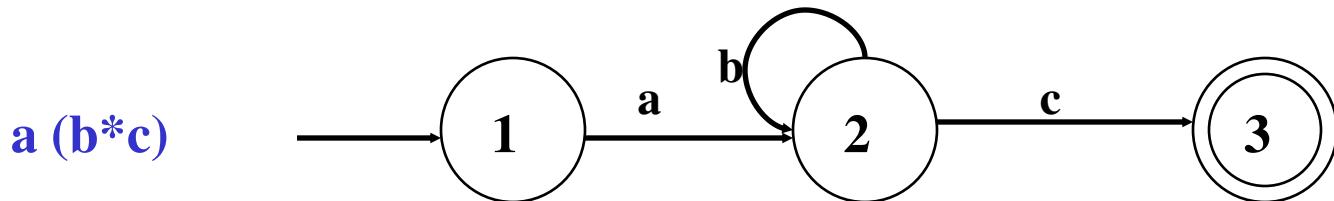
Given the regular expression : $a(b^*c) \mid a(b \mid c^+)$?

Find a transition diagram NFA that recognizes it.



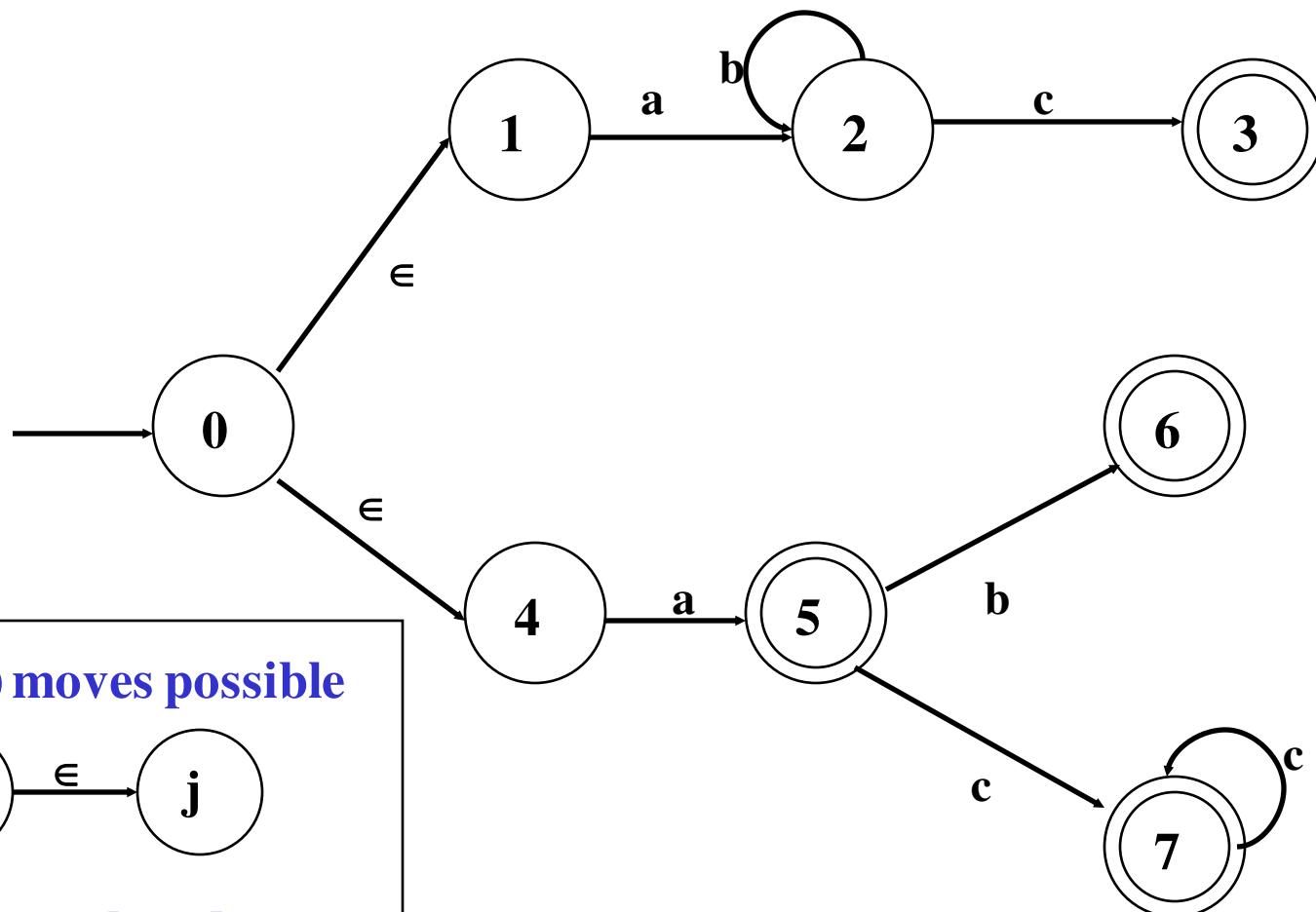
String abbc can be accepted.

Alternative Solution Strategy



Now that you have the individual diagrams, “or” them as follows:

Using Null Transitions to “OR” NFAs



Switch state but do not
use any input symbol

Deterministic Finite Automata (DFA)

- Problems with NFAs for Regular Expressions:
 - Valid input might not be accepted
 - NFA may behave differently on the same input
- **Deterministic Finite Automata DFA**
 - all outgoing edges are labelled with an input character
 - no state has ϵ -transition, **transition on input ϵ**
 - *no* two edges leaving a given state have the *same* label
 - for each state s and input symbol a , there is at most **one edge** label a leaving s
 - **Therefore:** the next state can be *determined* uniquely, given the current state and the current input character

Deterministic Finite Automata

A **DFA** is an NFA with the following restrictions:

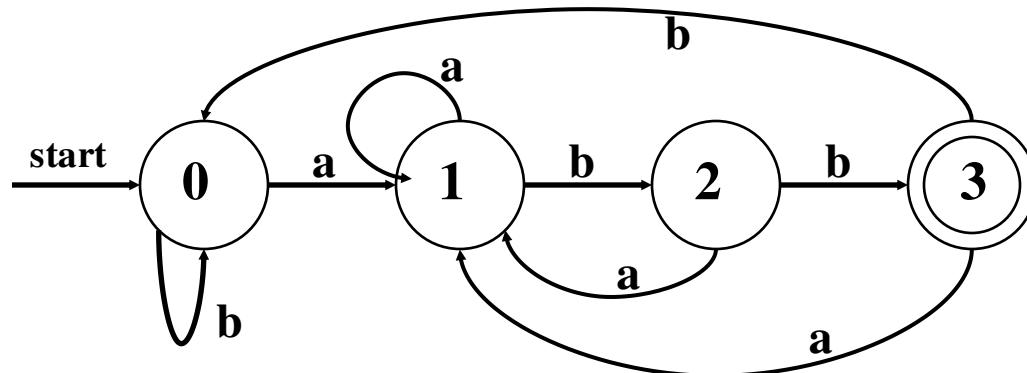
- no state has an ϵ -transitions (moves)
- for every state $s \in S$ and input symbol $a \in \Sigma$, there is at most one edge labeled a leaving s .

DFAs are easily simulated via an algorithm.

Algorithm 3.1

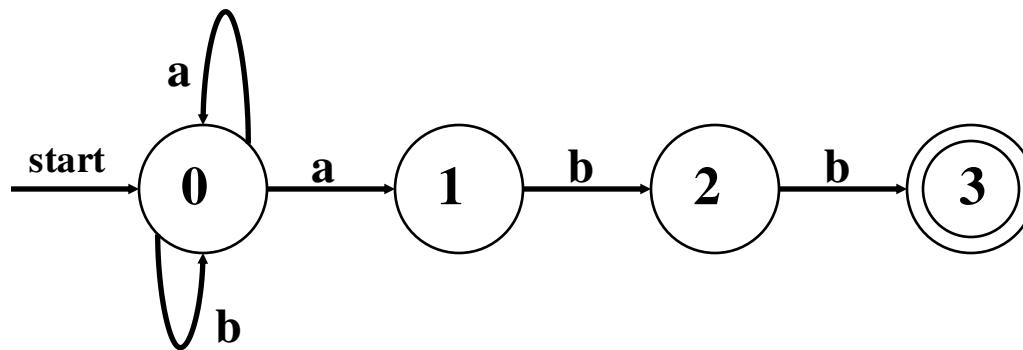
```
s ← s0
c ← nextchar();
while c ≠ eof do
    s ← move(s, c);
    c ← nextchar();
end;
if s is in F then return "yes"
else return "no"
```

Example - DFA



What Language is Accepted?

Recall the original NFA:



Conversion : NFA \rightarrow DFA Algorithm

- NFA inefficient to implement directly, so convert to a DFA that recognizes the same strings
- Why does this occur ?
 - \in moves
 - non-determinism
 - Both require us to characterize multiple situations that occur for accepting the same string.
 - (Recall : Same input can have multiple paths in NFA)
- Algorithm Constructs a Transition Table for DFA from NFA
- Idea:
 - Each DFA state corresponds to a distinct *set* of NFA states

Subset Construction Algorithm Concepts

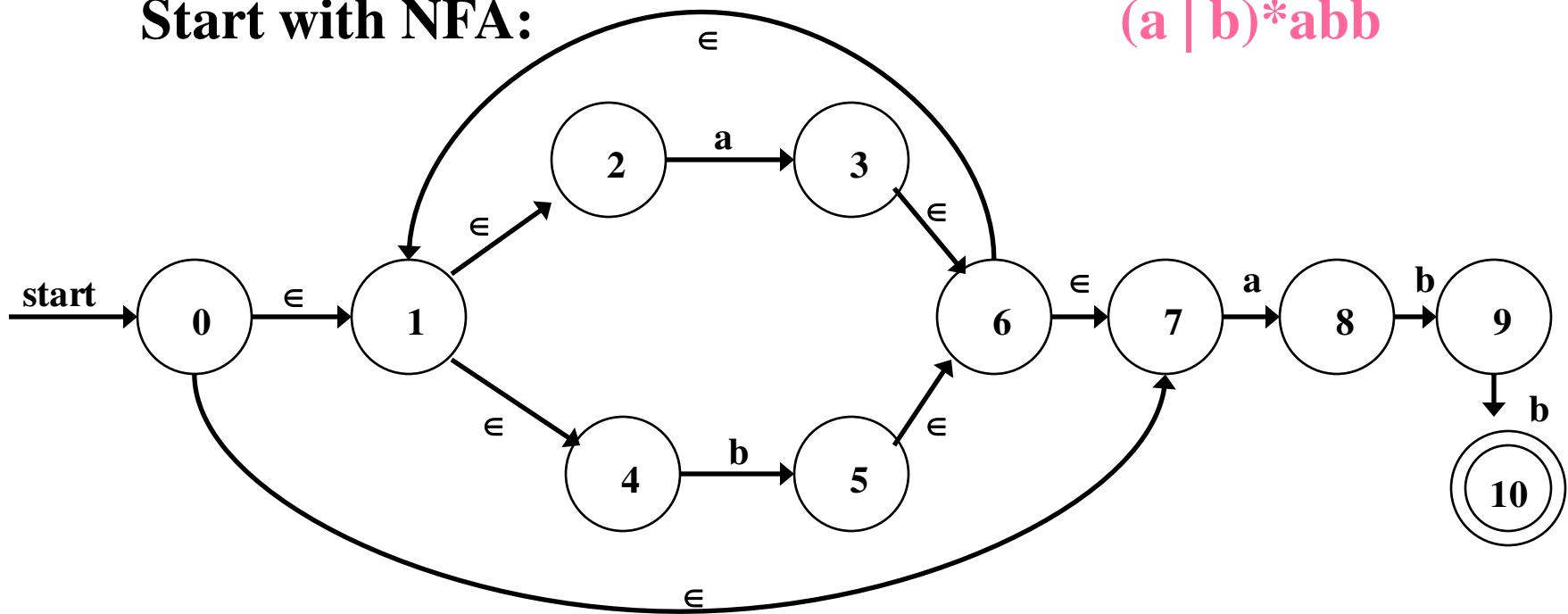
NFA $N = (S, \Sigma, s_0, F, \text{MOVE})$

- $\in\text{-Closure}(s) : s \in S$
: set of NFA states that are reachable
from s on $\in\text{-moves}$ only
- $\in\text{-Closure}(T) : T \subseteq S$
: NFA states reachable from some $t \in T$
on $\in\text{-moves}$ only
- $\text{move}(T, a) : T \subseteq S, a \in \Sigma$
: Set of states to which there is a transition on
input a from some $t \in T$
- No input is consumed

These 3 operations are utilized by algorithms /
techniques to facilitate the conversion process.

Illustrating Conversion – An Example

Start with NFA:



$(a \mid b)^*abb$

First we calculate: ϵ -closure(0) (i.e., state 0)

ϵ -closure(0) = {0, 1, 2, 4, 7} (all states reachable from 0 on ϵ -moves)

Let A={0, 1, 2, 4, 7} be a state of new DFA, D.

Conversion Example – continued (1)

2nd, we calculate: a : \in -closure($move(A, a)$) and
b : \in -closure($move(A, b)$)

a : \in -closure($move(A, a)$) = \in -closure($move(\{0,1,2,4,7\}, a)$)
adds {3, 8} (since $move(2, a) = 3$ and $move(7, a) = 8$)

From this we have : \in -closure({3, 8}) = {1, 2, 3, 4, 6, 7, 8}
(since $3 \rightarrow 6 \rightarrow 1 \rightarrow 4$, $6 \rightarrow 7$, and $1 \rightarrow 2$ all by \in -moves)

Let B = {1, 2, 3, 4, 6, 7, 8} be a new state. Define Dtran[A, a] = B.

b : \in -closure($move(A, b)$) = \in -closure($move(\{0,1,2,4,7\}, b)$)
adds {5} (since $move(4, b) = 5$)

From this we have : \in -closure({5}) = {1, 2, 4, 5, 6, 7}
(since $5 \rightarrow 6 \rightarrow 1 \rightarrow 4$, $6 \rightarrow 7$, and $1 \rightarrow 2$ all by \in -moves)

Let C = {1, 2, 4, 5, 6, 7} be a new state. Define Dtran[A, b] = C.

Conversion Example – continued (2)

3rd , we calculate for state B on {a,b}

$$\begin{aligned}\underline{\mathbf{a}} : \in\text{-closure}(\text{move}(B,a)) &= \in\text{-closure}(\text{move}(\{1,2,3,4,6,7,8\},a))\} \\ &= \{1,2,3,4,6,7,8\} = B\end{aligned}$$

Define Dtran[B,a] = B.

$$\begin{aligned}\underline{\mathbf{b}} : \in\text{-closure}(\text{move}(B,b)) &= \in\text{-closure}(\text{move}(\{1,2,3,4,6,7,8\},b))\} \\ &= \{1,2,4,5,6,7,9\} = D\end{aligned}$$

Define Dtran[B,b] = D.

4th , we calculate for state C on {a,b}

$$\begin{aligned}\underline{\mathbf{a}} : \in\text{-closure}(\text{move}(C,a)) &= \in\text{-closure}(\text{move}(\{1,2,4,5,6,7\},a))\} \\ &= \{1,2,3,4,6,7,8\} = B\end{aligned}$$

Define Dtran[C,a] = B.

$$\begin{aligned}\underline{\mathbf{b}} : \in\text{-closure}(\text{move}(C,b)) &= \in\text{-closure}(\text{move}(\{1,2,4,5,6,7\},b))\} \\ &= \{1,2,4,5,6,7\} = C\end{aligned}$$

Define Dtran[C,b] = C.

Conversion Example – continued (3)

5th , we calculate for state D on {a,b}

$$\begin{aligned}\underline{\mathbf{a}} : \in\text{-closure}(\text{move}(D,a)) &= \in\text{-closure}(\text{move}(\{1,2,4,5,6,7,9\},a))\} \\ &= \{1,2,3,4,6,7,8\} = B\end{aligned}$$

Define $\text{Dtran}[D,a] = B$.

$$\begin{aligned}\underline{\mathbf{b}} : \in\text{-closure}(\text{move}(D,b)) &= \in\text{-closure}(\text{move}(\{1,2,4,5,6,7,9\},b))\} \\ &= \{1,2,4,5,6,7,10\} = E\end{aligned}$$

Define $\text{Dtran}[D,b] = E$.

Finally, we calculate for state E on {a,b}

$$\begin{aligned}\underline{\mathbf{a}} : \in\text{-closure}(\text{move}(E,a)) &= \in\text{-closure}(\text{move}(\{1,2,4,5,6,7,10\},a))\} \\ &= \{1,2,3,4,6,7,8\} = B\end{aligned}$$

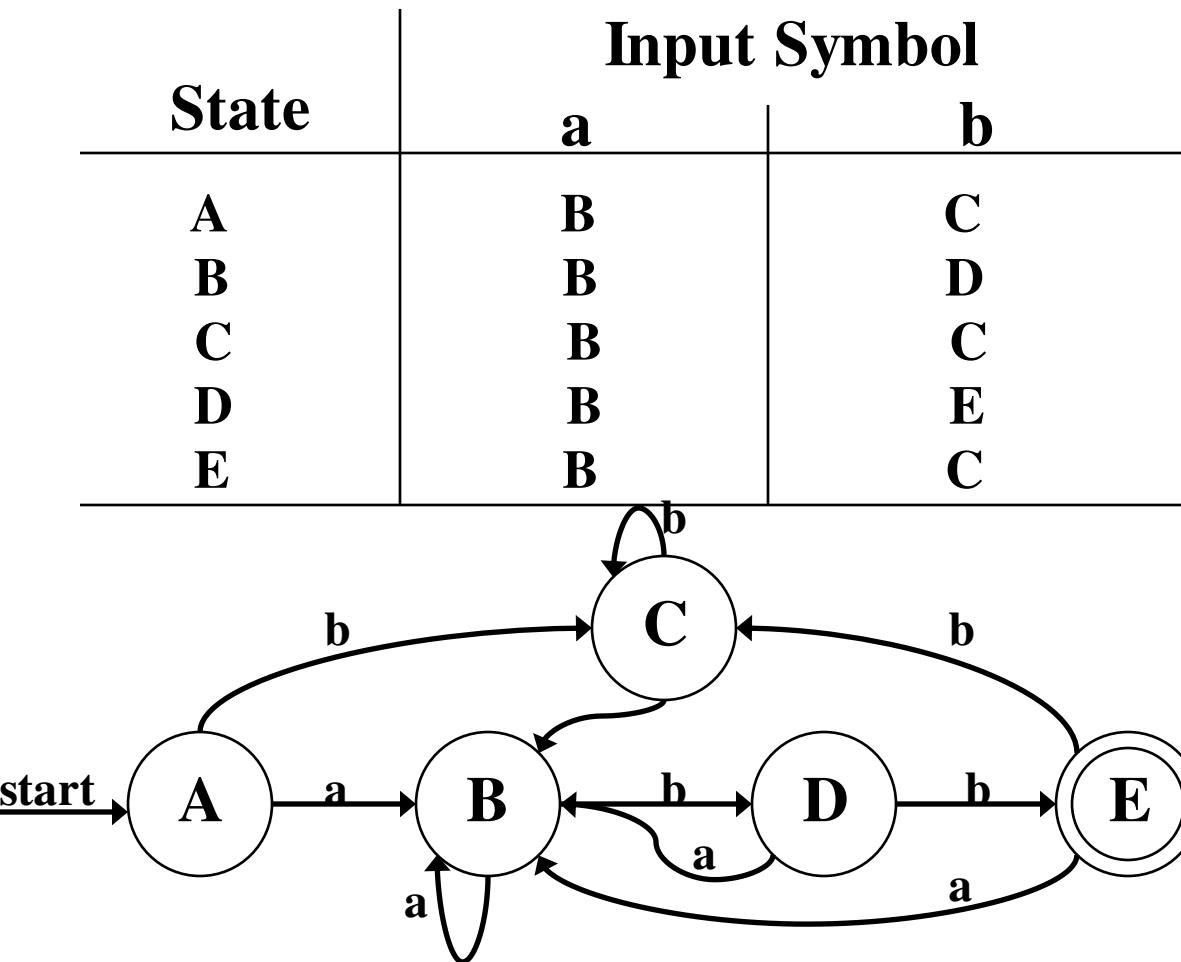
Define $\text{Dtran}[E,a] = B$.

$$\begin{aligned}\underline{\mathbf{b}} : \in\text{-closure}(\text{move}(E,b)) &= \in\text{-closure}(\text{move}(\{1,2,4,5,6,7,10\},b))\} \\ &= \{1,2,4,5,6,7\} = C\end{aligned}$$

Define $\text{Dtran}[E,b] = C$.

Conversion Example – continued (4)

This gives the transition table for the DFA of:



NFA → DFA with Subset Construction

Need to simulate the NFA with the DFA

The algorithm

- States of the DFA correspond to sets of states of NFA
 - DFA is keeping track of all the states the NFA can reach after reading an input string
- Start state of the DFA derived from start state of the NFA
 - Take ϵ -closure of the start state of the NFA
- Work forward, trying each $\alpha \in \Sigma$ and taking its ϵ -closure
- Iterative algorithm that halts when the states wrap back on themselves

NFA → DFA with Subset Construction

initial state of DFA

initial state of NFA

```
 $U_0 \leftarrow \varepsilon\text{-closure}(s_0);$ 
Initialize Dstates with  $U_0$ ;
while ( $\exists$  unmarked  $T \in Dstates$ ) begin
    mark  $T$ ;
    for each  $\alpha \in \Sigma$  begin;
         $U \leftarrow \varepsilon\text{-closure}(\text{move}(T, \alpha))$ 
        if ( $U \notin Dstates$ ) then
            add  $U$  as an unmarked state to  $Dstates$ ;
             $Dtran[T, \alpha] \leftarrow U$ ;
    end;
end;
```

- $Dstates$ is the set of states of the constructed DFA
- $Dtran$ is the transition function of the DFA
- A DFA state is an accepting state if it contains an accepting NFA state

Algorithm For Subset Construction

initially, \in -closure(s_0) is only (unmarked) state in **Dstates**;

while there is unmarked state **T** in **Dstates** do begin

 mark **T**;

 for each input symbol a do begin

$U := \in$ -closure($move(T,a)$);

 if **U** is not in **Dstates** then

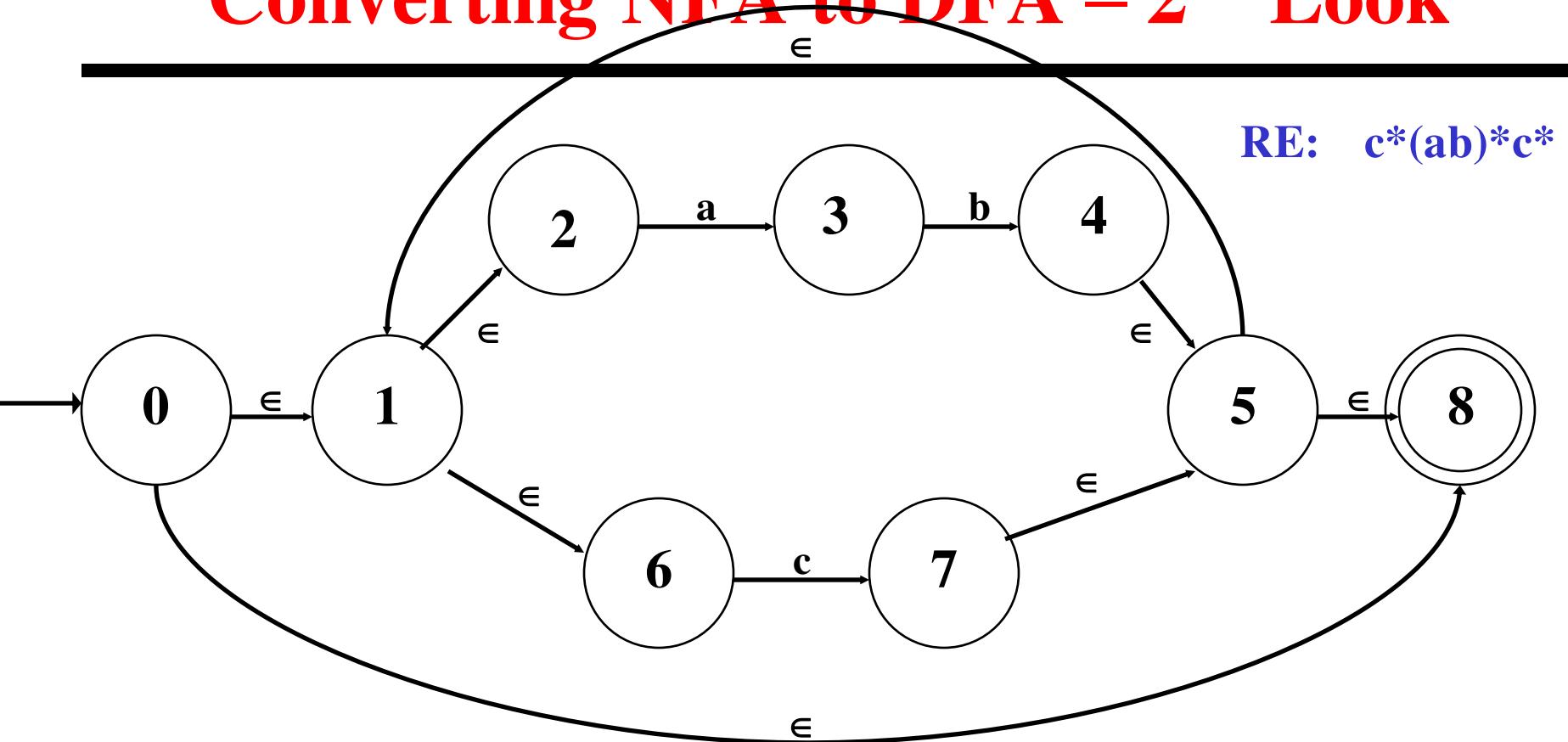
 add **U** as an unmarked state to **Dstates**;

Dtran[**T,a**] := **U**

 end

end

Converting NFA to DFA – 2nd Look



1st we calculate: ϵ -closure(0)

$$\epsilon\text{-closure}(0) = \{0, 1, 2, 6, 8\}$$

(all states reachable from 0 on ϵ -moves)

Let A={0, 1, 2, 6, 8} be a state of new DFA, D.

Conversion Example – continued (1)

2nd, we calculate: $a : \in\text{-closure}(\text{move}(A, a))$ and
 $b : \in\text{-closure}(\text{move}(A, b))$
 $c : \in\text{-closure}(\text{move}(A, c))$

a : $\in\text{-closure}(\text{move}(A, a)) = \in\text{-closure}(\text{move}(\{0,1,2,6,8\}, a))$
adds {3} (since $\text{move}(2, a) = 3$)

From this we have: $\in\text{-closure}(\{3\}) = \{3\}$
(since $3 \rightarrow 3$ by $\in\text{-moves}$)

Let $B = \{3\}$ be a new state. Define $D\text{tran}[A, a] = B$.

b : $\in\text{-closure}(\text{move}(A, b)) = \in\text{-closure}(\text{move}(\{0,1,2,6,8\}, b))$

There is NO transition on **b** for all states 0,1,2,6 and 8

Define $D\text{tran}[A, b] = \text{Reject}$.

c : $\in\text{-closure}(\text{move}(A, c)) = \in\text{-closure}(\text{move}(\{0,1,2,6,8\}, c))$
adds {7} (since $\text{move}(6, c) = 7$)

From this we have: $\in\text{-closure}(\{7\}) = \{1,2,5,6,7,8\}$

(since $7 \rightarrow 5 \rightarrow 8$, $7 \rightarrow 5 \rightarrow 1 \rightarrow 2$, and $7 \rightarrow 5 \rightarrow 1 \rightarrow 6$ by $\in\text{-moves}$)

Let $C = \{1,2,5,6,7,8\}$ be a new state. Define $D\text{tran}[A, c] = C$.

Conversion Example – continued (2)

3rd , we calculate: $a : \in\text{-closure}(\text{move}(B, a))$ and
 $b : \in\text{-closure}(\text{move}(B, b))$
 $c : \in\text{-closure}(\text{move}(B, c))$

a : $\in\text{-closure}(\text{move}(B, a)) = \in\text{-closure}(\text{move}(\{3\}, a))\}$

There is NO transition on **a** for state 3

Define **Dtran[B, a] = Reject.**

b : $\in\text{-closure}(\text{move}(B, b)) = \in\text{-closure}(\text{move}(\{3\}, b))$
adds {4} (since $\text{move}(3, b) = 4$)

From this we have : $\in\text{-closure}(\{4\}) = \{1, 2, 4, 5, 6, 8\}$

(since $4 \rightarrow 5 \rightarrow 8$, $4 \rightarrow 5 \rightarrow 1 \rightarrow 2$, and $4 \rightarrow 5 \rightarrow 1 \rightarrow 6$ by $\in\text{-moves}$)

Let $D = \{1, 2, 4, 5, 6, 8\}$ be a new state. Define **Dtran[B, b] = D.**

c : $\in\text{-closure}(\text{move}(B, c)) = \in\text{-closure}(\text{move}(\{3\}, c))$

There is NO transition on **c** for state 3

Define **Dtran[B, c] = Reject.**

Conversion Example – continued (3)

4th, we calculate: $a : \in\text{-closure}(\text{move}(C, a))$ and
 $b : \in\text{-closure}(\text{move}(C, b))$
 $c : \in\text{-closure}(\text{move}(C, c))$

a : $\in\text{-closure}(\text{move}(C, a)) = \in\text{-closure}(\text{move}(\{1,2,5,6,7,8\}, a))$
adds {3} (since $\text{move}(2, a) = 3$)
 $\in\text{-closure}(\{3\}) = \{3\}$ Define $D\text{tran}[C, a] = B$.

b : $\in\text{-closure}(\text{move}(C, b)) = \in\text{-closure}(\text{move}(\{1,2,5,6,7,8\}, b))$
There is NO transition on **b** for all states 1,2,5,6, and 7
Define $D\text{tran}[C, b] = \text{Reject}$.

c : $\in\text{-closure}(\text{move}(C, c)) = \in\text{-closure}(\text{move}(\{1,2,5,6,7,8\}, c))$
adds {7} (since $\text{move}(6, c) = 7$)
From this we have : $\in\text{-closure}(\{7\}) = \{1,2,5,6,7,8\}$
(since $7 \rightarrow 5 \rightarrow 8$, $7 \rightarrow 5 \rightarrow 1 \rightarrow 2$, and $7 \rightarrow 5 \rightarrow 1 \rightarrow 6$ by $\in\text{-moves}$)
Define $D\text{tran}[C, c] = C$.

Conversion Example – continued (4)

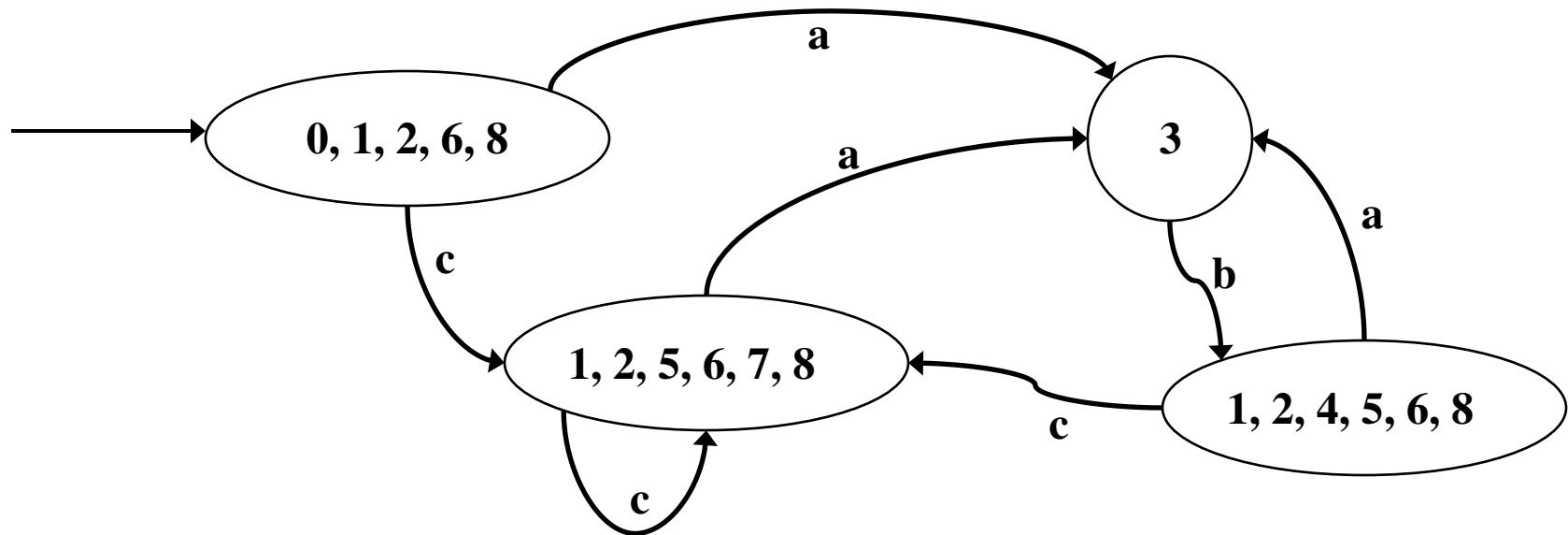
5th, we calculate: $a : \in\text{-closure}(\text{move}(D, a))$ and
 $b : \in\text{-closure}(\text{move}(D, b))$
 $c : \in\text{-closure}(\text{move}(D, c))$

a : $\in\text{-closure}(\text{move}(D, a)) = \in\text{-closure}(\text{move}(\{1,2,4,5,6,8\}, a))$
adds {3} (since $\text{move}(2, a) = 3$)
 $\in\text{-closure}(\{3\}) = \{3\}$ Define $D\text{tran}[D, a] = B$.

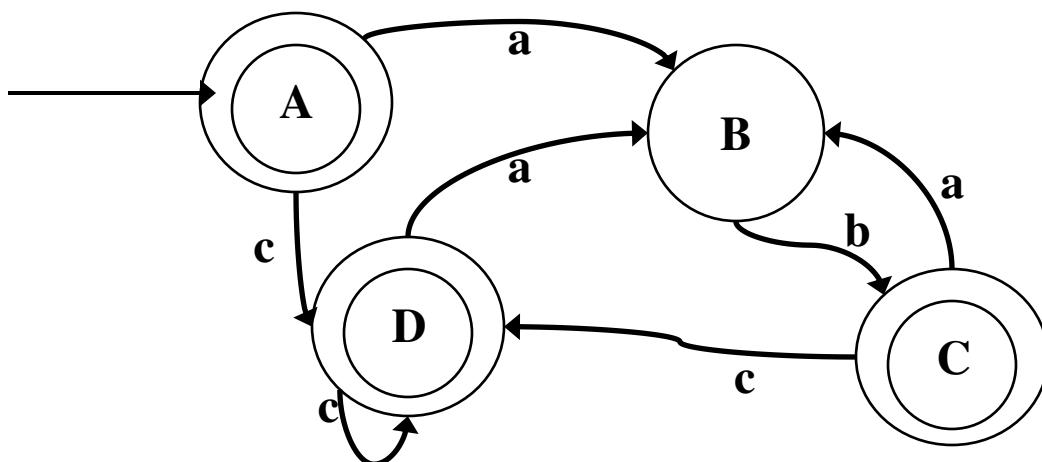
b : $\in\text{-closure}(\text{move}(D, b)) = \in\text{-closure}(\text{move}(\{1,2,4,5,6,8\}, b))$
There is NO transition on **b** for all states 1,2,4,5,6, and 8
Define $D\text{tran}[D, b] = \text{Reject}$.

c : $\in\text{-closure}(\text{move}(D, c)) = \in\text{-closure}(\text{move}(\{1,2,4,5,6,8\}, c))$
adds {7} (since $\text{move}(6, c) = 7$)
From this we have : $\in\text{-closure}(\{7\}) = \{1,2,5,6,7,8\}$
(since $7 \rightarrow 5 \rightarrow 8$, $7 \rightarrow 5 \rightarrow 1 \rightarrow 2$, and $7 \rightarrow 5 \rightarrow 1 \rightarrow 6$ by $\in\text{-moves}$)
Define $D\text{tran}[D, c] = C$.

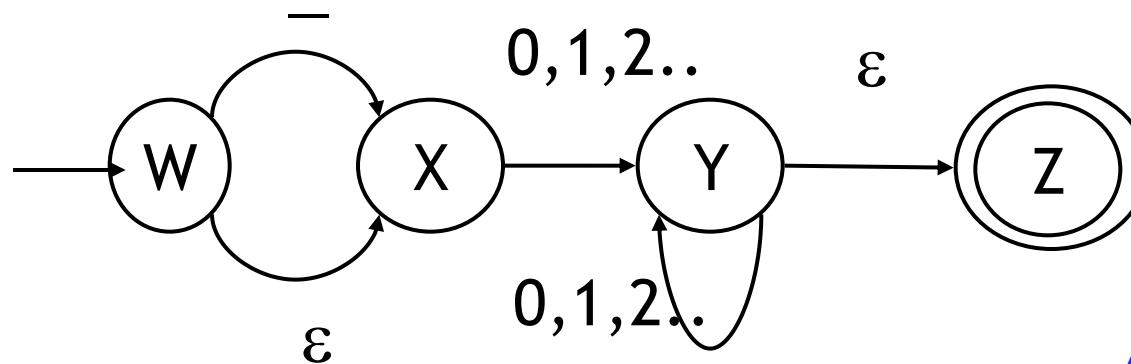
The Resulting DFA



Which States are **FINAL** States ?

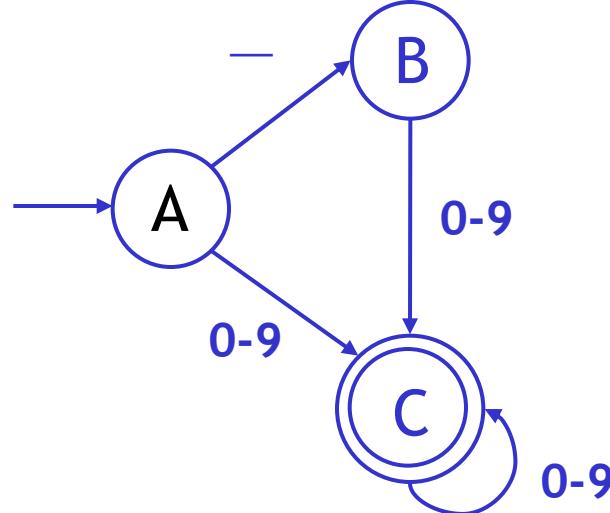


NFA states to DFA states



- Possible sets of states:
W&X, X, Y&Z
- DFA states:

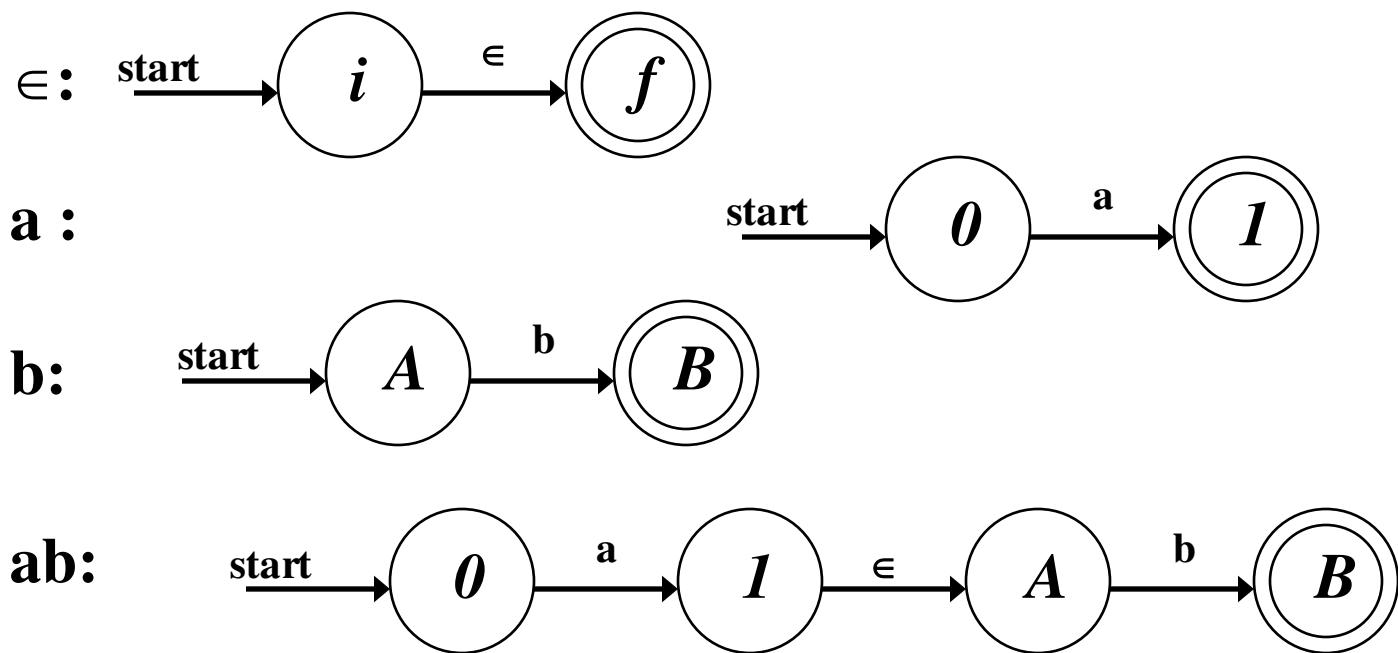
A B C



Regular Expression to NFA Construction

- We now focus on transforming a Reg. Expr. to an NFA
- This construction allows us to take:
 - Regular Expressions (which describe tokens)
 - To an NFA (to characterize language)
 - To a DFA (which can be computerized)
 - Builds NFA from components of the regular expression in a special order with particular techniques.

Motivation: Construct NFA For:



Construction Algorithm : R.E. \rightarrow NFA

Construction Process :

1st : Identify subexpressions of the regular expression

\in

Σ symbols

$r \mid s$

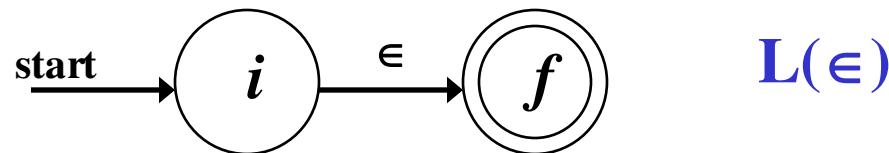
rs

r^*

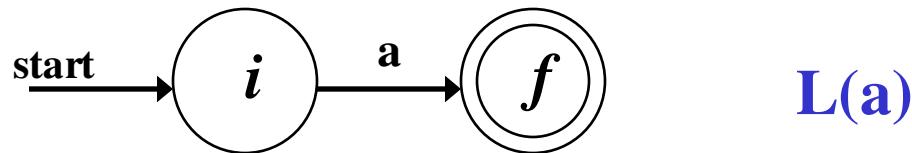
2nd : Characterize “pieces” of NFA for each subexpression

Thompson's Construction Algorithm

1. For ϵ in the regular expression, construct NFA

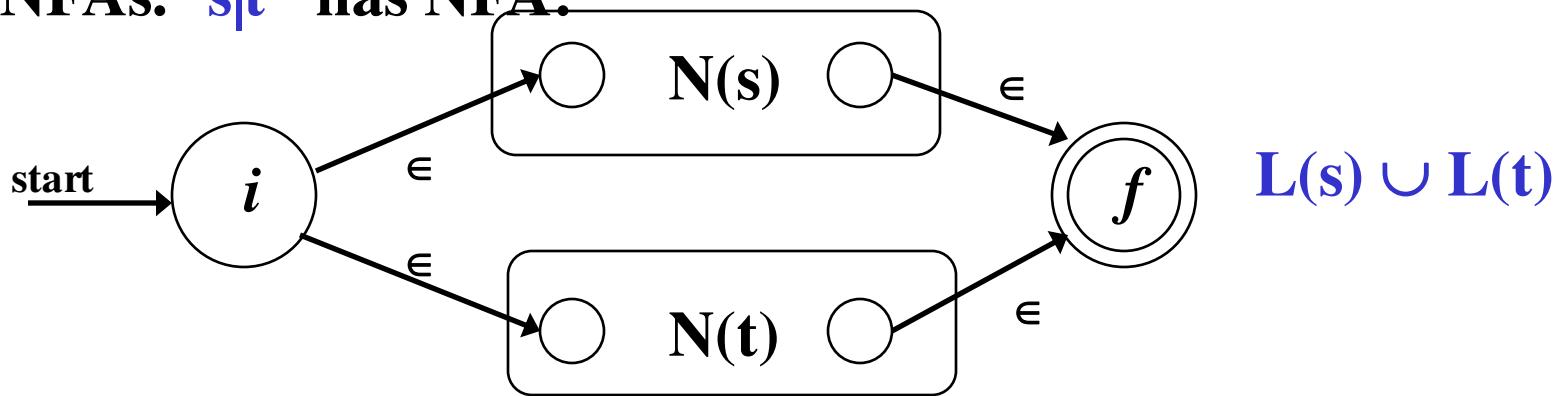


2. For $a \in \Sigma$ in the regular expression, construct NFA



Piecing Together NFAs – continued(1)

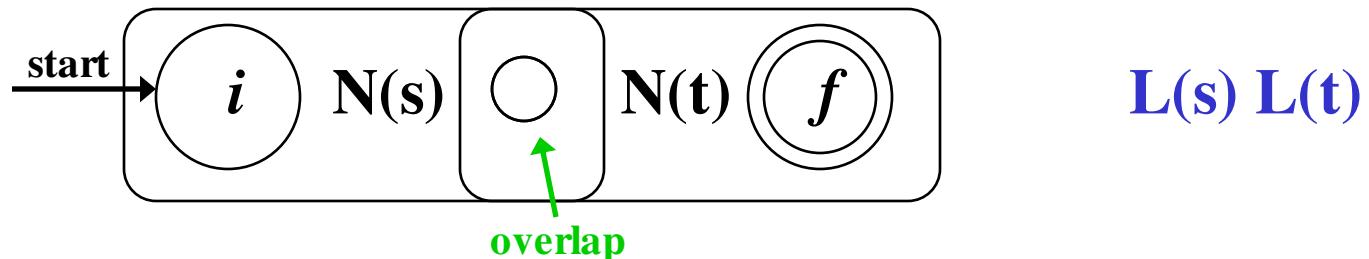
3.(a) If s and t are regular expressions, $N(s)$, $N(t)$ their NFAs. $s|t$ has NFA:



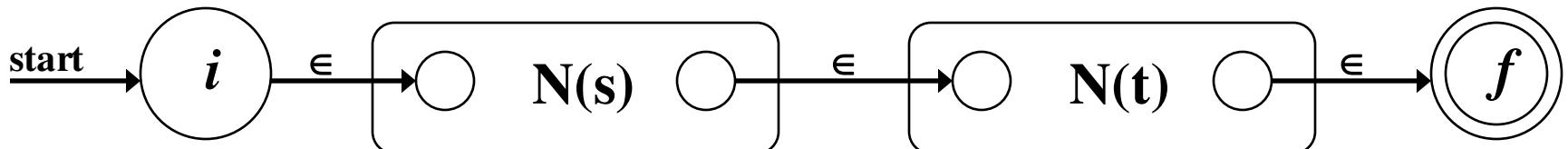
where i and f are new start / final states, and ϵ -moves are introduced from i to the old start states of $N(s)$ and $N(t)$ as well as from all of their final states to f .

Piecing Together NFAs – continued(2)

3.(b) If s and t are regular expressions, $N(s)$, $N(t)$ their NFAs. st (concatenation) has NFA:



Alternative:

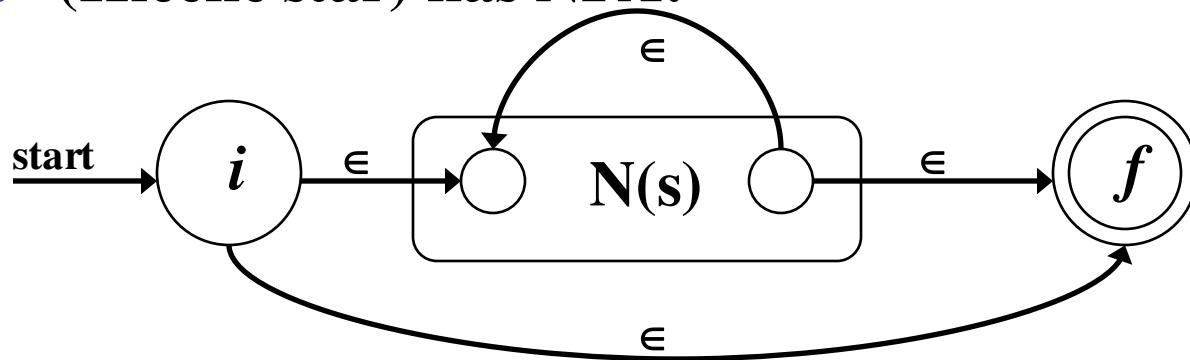


where i is the start state of $N(s)$ (or new under the alternative) and f is the final state of $N(t)$ (or new). Overlap maps final states of $N(s)$ to start state of $N(t)$.

Piecing Together NFAs – continued(3)

3.(c) If s is a regular expression, $N(s)$ its NFA,

s^* (Kleene star) has NFA:



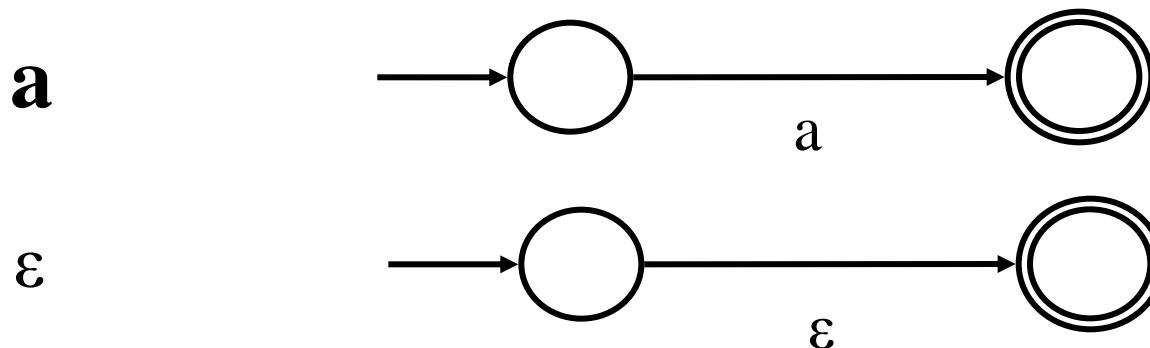
where : i is new start state and f is new final state

ϵ -move i to f (to accept null string)

ϵ -moves i to old start, old final(s) to f

ϵ -move old final to old start (WHY?)

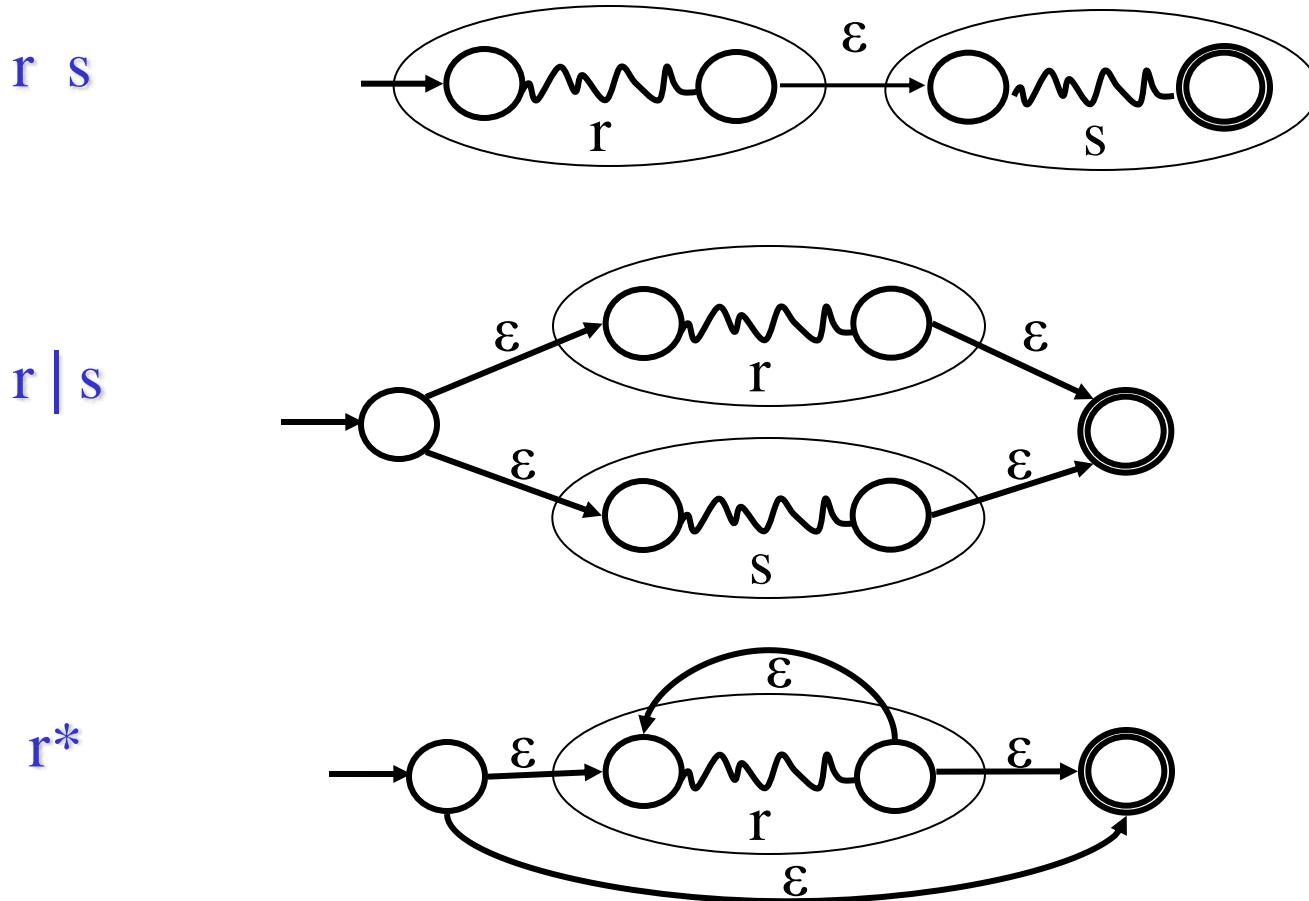
Converting an RE to an NFA



If r and s are regular expressions with the NFA's



Inductive Construction

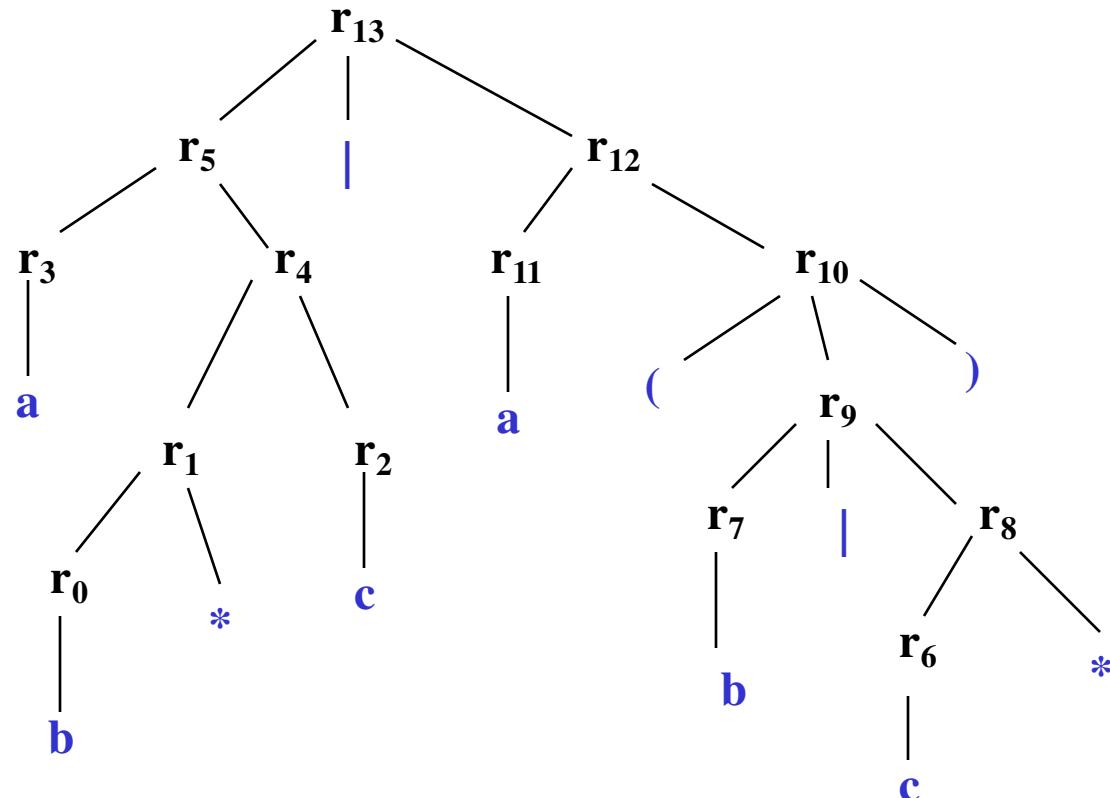


Detailed Example

See example 3.16 in textbook for $(a \mid b)^*abb$

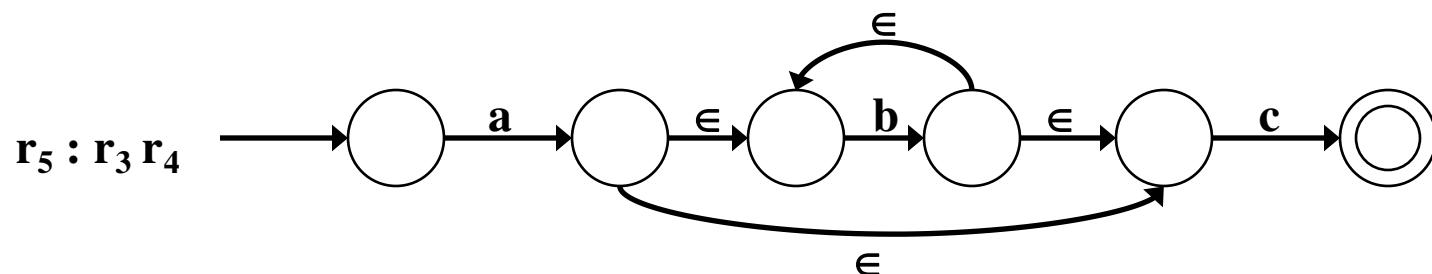
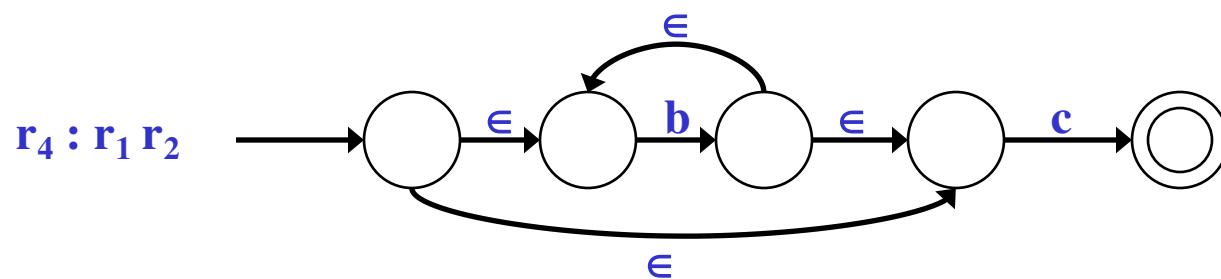
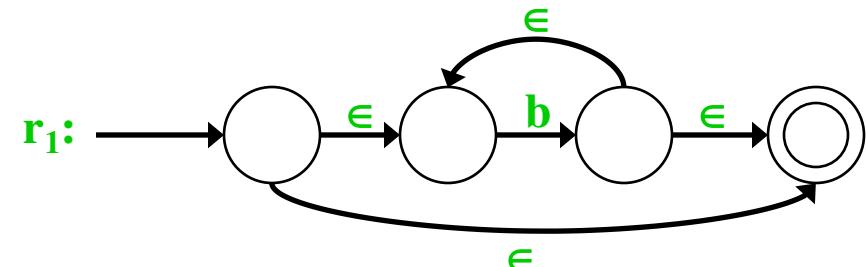
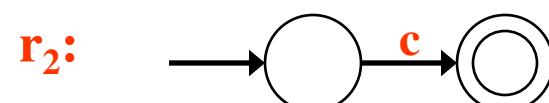
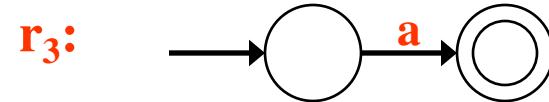
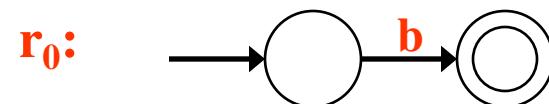
2nd Example: $(ab^*c) \mid (a(b \mid c^*))$

Decomposition for this regular expression:

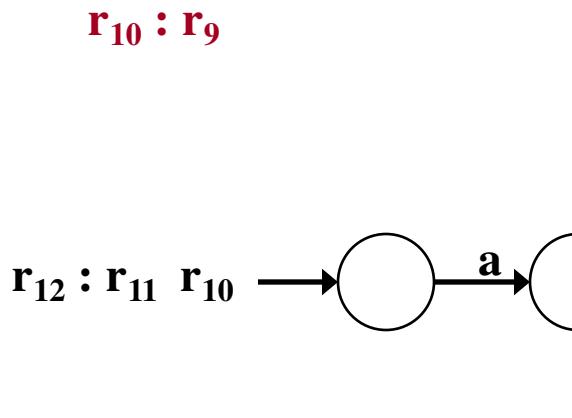
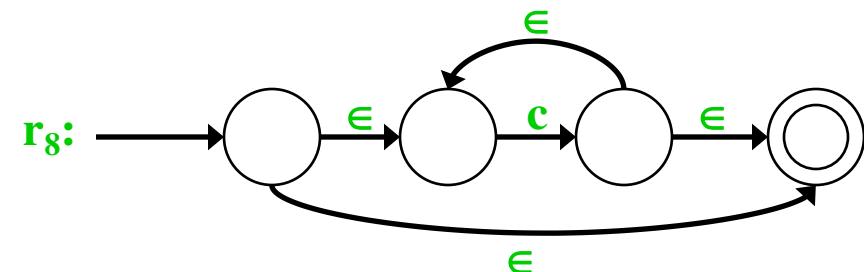
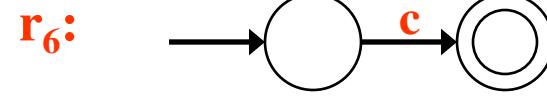
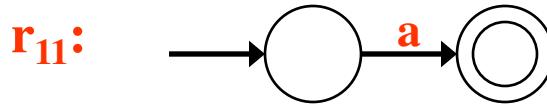
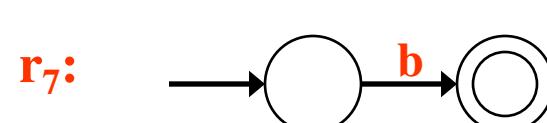


What is the NFA? Let's construct it !

Detailed Example – Construction(1)

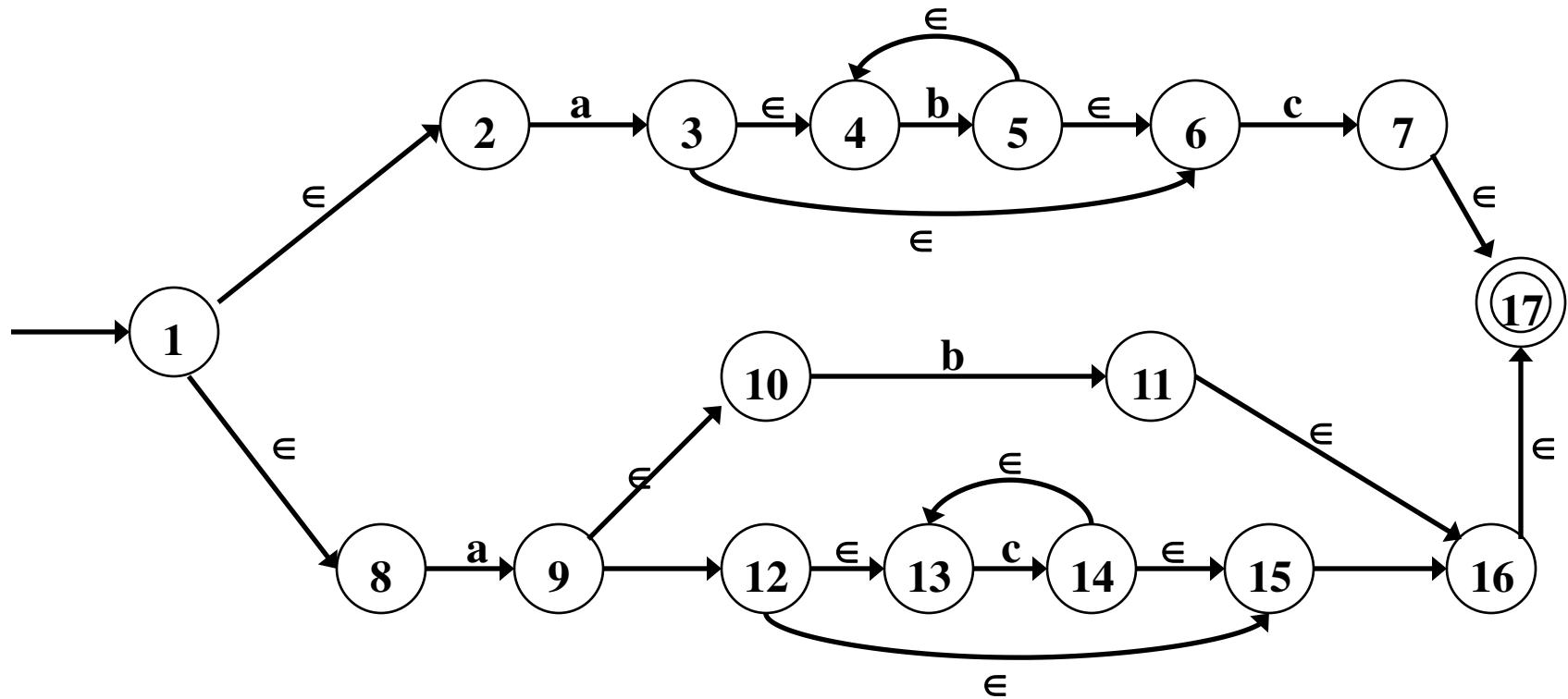


Detailed Example – Construction(2)



Detailed Example – Final Step

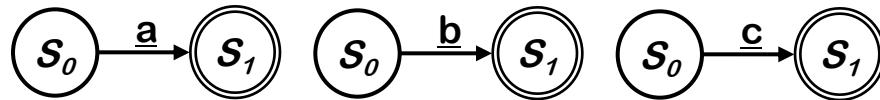
$$r_{13} : r_5 \mid r_{12}$$



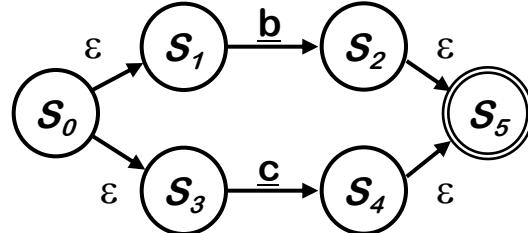
Example of Thompson's Construction

Let's try $\underline{a} (\underline{b} \mid \underline{c})^*$

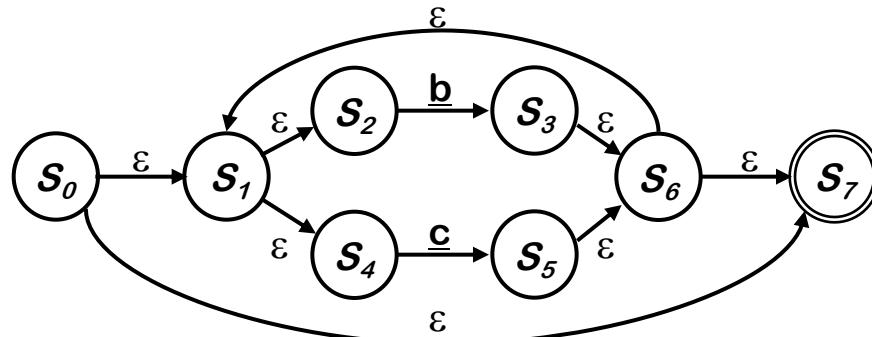
1. \underline{a} , \underline{b} , & \underline{c}



2. $\underline{b} \mid \underline{c}$

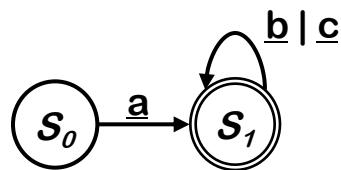
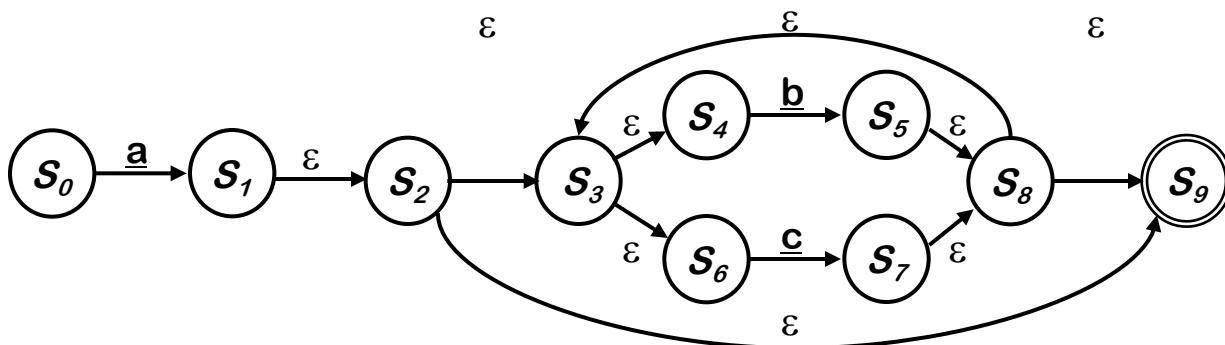


3. $(\underline{b} \mid \underline{c})^*$



Example of Thompson's Construction (continued)

4. $a (\underline{b} \mid \underline{c})^*$



Simulating an NFA (Algorithm 3.4)

Input: An NFA N constructed by Algorithm 3.3 and an input string x

Output: the answer “yes” if N accept x ; “no” otherwise

```
s ← ε-closure({s0})  
c ← nextchar(x) ;  
while c ≠ eof do  
    s ← ε-closure(move(s, c)) ;  
    c ← nextchar(x) ;  
end;  
if s is in F then return “yes”  
else return “no”
```

Design of a Lexical Analyzer

Relationship of NFAs to Compilation:

1. Regular expression “**recognized**” by NFA
 2. Regular expression is “**pattern**” for a “**token**”
 3. Tokens are building blocks for lexical analysis
 4. Lexical analyzer can be described by a collection of NFAs. Each NFA is for a language token.
-
- To build a token recognizer:
 - Write down the RE for the input language
 - Build a big NFA
 - Build the DFA that simulates the NFA
 - Minimize DFA
 - Turn it into code

Design of a Lexical Analyzer

- RE→NFA : Alg. 3.3 (Thompson's construction)
- Simulate NFA: Algorithm 3.4

○ OR

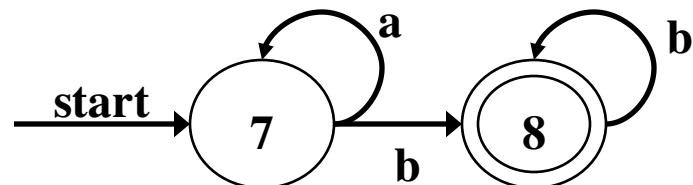
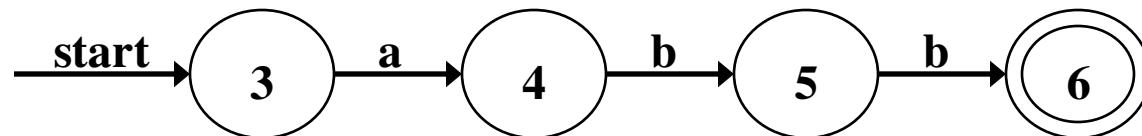
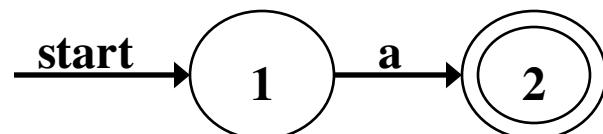
- RE→NFA : Alg. 3.3 (Thompson's construction)
- NFA →DFA: Alg. 3.2 (Subset construction)
- Simulate DFA: Algorithm 3.1

Example

Let : a
 abb
 a^*b^+

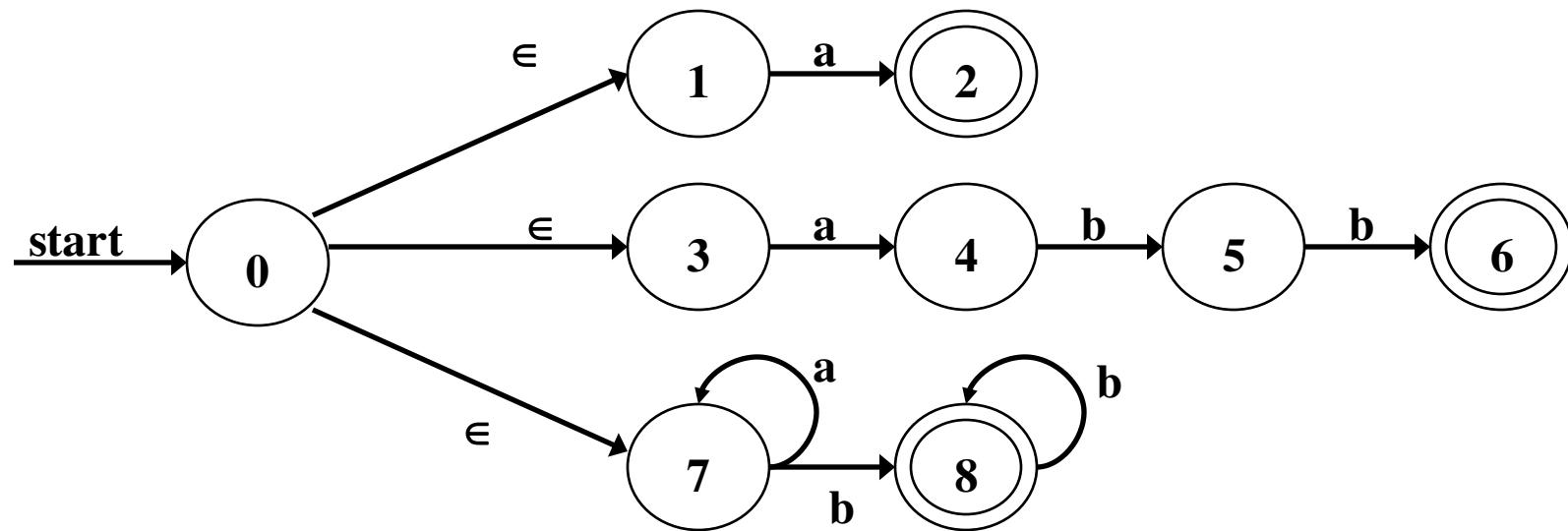
} 3 patterns

NFA's :



Example – continued(1)

Combined NFA :



Construct DFA : (It has 6 states)

{0,1,3,7}, {2,4,7}, {5,8}, {6,8}, {7}, {8}

Can you do this conversion ???

Example – continued(2)

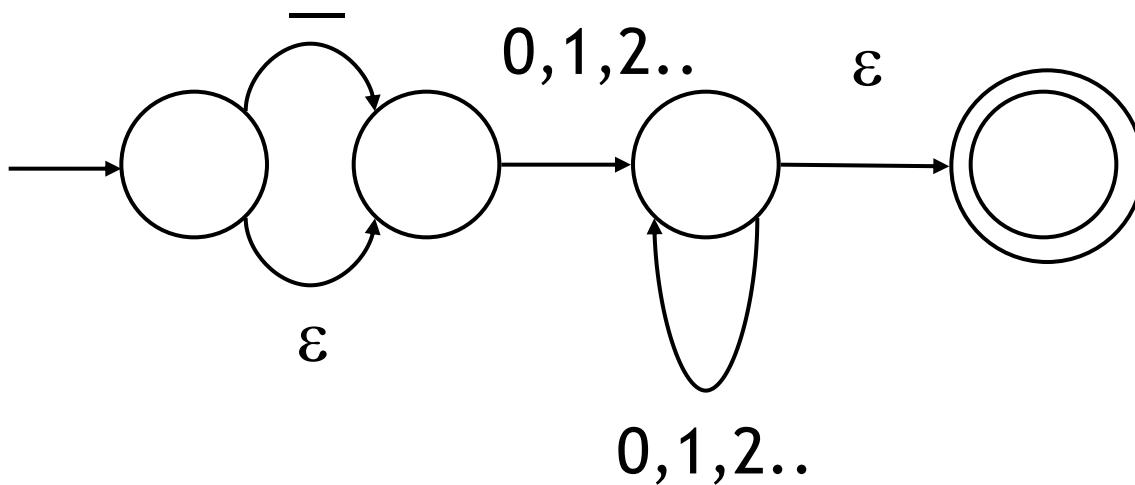
Dtran for this example:

	Input Symbol	
STATE	a	b
{0,1,3,7}	{2,4,7}	{8}
{2,4,7}	{7}	{5,8}
{8}	-	{8}
{7}	{7}	{8}
{5,8}	-	{6,8}
{6,8}	-	{8}

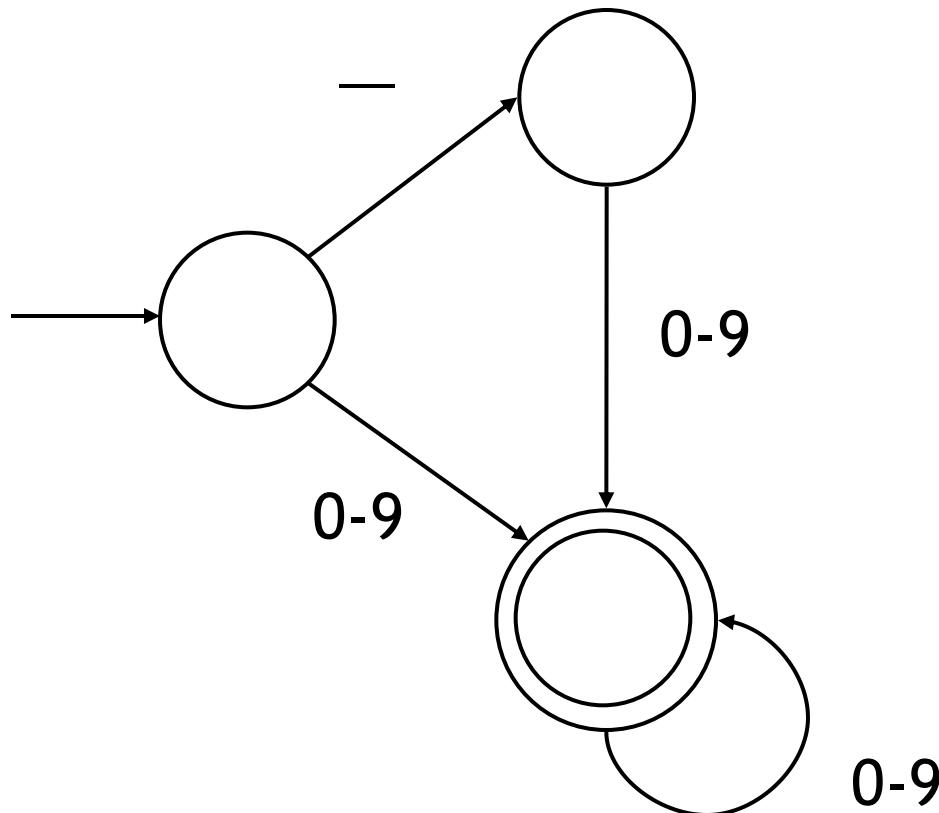
RE \Rightarrow NFA

-?[0-9]+

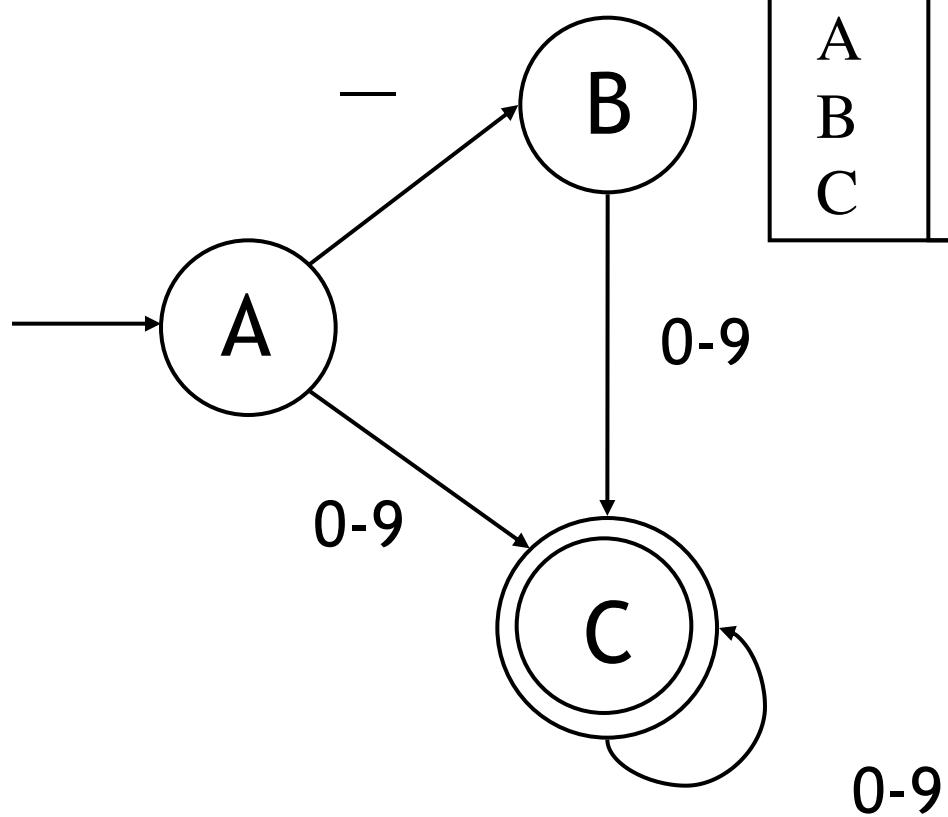
(-|ε) [0-9][0-9]*



NFA \Rightarrow minimized DFA



DFA \Rightarrow table



	-	0-9
A		
B		
C	ERR	ERR

Summary

- Lexical analyzer converts a text stream to tokens
- Tokens defined using regular expressions
- Regular expressions can be converted to a simple table-driven tokenizer by converting them to NFAs and then to DFAs.

Compilers Construction

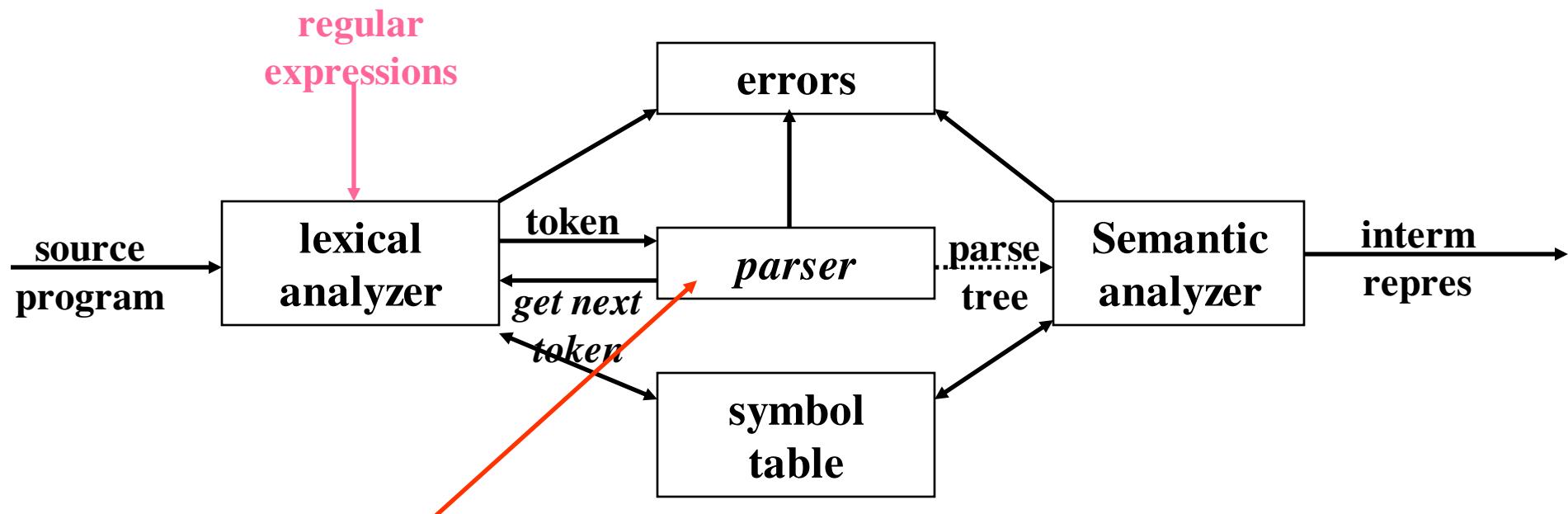
Chapter 3

Context-Free Grammars and Parsing

Dr. Mohammed Salem Atoum

M.Atoum@inu.edu.jo

4.1: The Role of the PARSER



- Uses a grammar to check structure of tokens: CFG
- Produces a parse tree: top-down and bottom-up parsing
- Syntax error handling and error recovery: panic-mode, phrase-level, and error production recovery

4.2: Context-Free Grammars (CFG)

- CFG's are language-description mechanisms defined by the following 4-tuple: $G = (NT, T, S, P)$ where
- NT is the set of non-terminal symbols
 - These are syntactic variables that denote sets of strings
- T is the set of terminal symbols
 - These correspond to tokens returned by the scanner
 - For the parser, tokens are indivisible units of syntax
 - $NT \cap T = \emptyset$
- S is the starting symbol $S \in NT$
 - All the strings in $L(G)$ are derived from the start symbol
- P a set of production rules: LHS \rightarrow RHS
 - LHS $\in NT$: single non-terminal
 - RHS $\in (NT \cup T)^*$: string of terminals and non-terminals

Notational Conventions

- Terminals: a,b,c,+,-,punc,0,1,...,9,
- Non-terminals: A, B, C, S
- T or NT: X, Y, Z (grammar symbols)
- Strings of Terminals: u,v,...,z in T^*
- Strings of T and/or NT: α, β, γ in $(T \cup NT)^*$
- Alternatives of production rules:
 - $A \rightarrow \alpha_1; A \rightarrow \alpha_2; \dots; A \rightarrow \alpha_k; \Rightarrow$
 - $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$
- First NT on LHS of 1st production rule is designated as start symbol

CFG: Example (1)

$$\begin{aligned} E &\rightarrow E \ op \ E \mid (E) \mid - E \mid id \\ op &\rightarrow + \mid - \mid * \mid / \mid \uparrow \end{aligned}$$

Black : NT

Blue : T

E : Start symbol

9 Production rules

CFG: Example (2)

1	<i>Goal</i>	\rightarrow	<i>Expr</i>
2	<i>Expr</i>	\rightarrow	<i>Expr Op Expr</i>
3			num
4			id
5	<i>Op</i>	\rightarrow	+
6			-
7			*
8			/

Start Symbol:

$$S = Goal$$

Non-terminal Symbols:

$$NT = \{ Goal, Expr, Op \}$$

Terminal symbols:

$$T = \{ num, id, +, -, *, / \}$$

Productions:

$$P = \{ 1, 2, 3, 4, 5, 6, 7, 8 \} \text{ (shown above)}$$

CFG: Example (3)

$$\begin{array}{l} S \rightarrow S + E \\ S \rightarrow E \\ E \rightarrow \text{number} \\ E \rightarrow (S) \end{array}$$

4 productions
2 non-terminals (S , E)
4 terminals: (,), +, number
start symbol S

CFG: Example (4)

assign_stmt \rightarrow id := *expr* ;

expr \rightarrow *term operator term*

term \rightarrow id

term \rightarrow real

term \rightarrow integer

operator \rightarrow +

operator \rightarrow -

Derivations

- **Derivation:** A sequence of grammar rule applications and substitutions that transform a starting non-terminal into a collection of terminals / tokens.
- If a grammar accepts a string, there is a *derivation* of that string using the productions of the grammar
- The following derivation $A \Rightarrow \beta$ is read “A derives β ”

Constructing a derivation

- Start from start symbol (S)
- Productions are used to derive a sequence of tokens from the start symbol
- At each step, we make two choices
 1. Choose a non-terminal to replace
 2. Choose a production to apply
- Different choices lead to different derivations
- For arbitrary strings α , β and γ and a production $A \rightarrow \beta$, a single step of derivation is $\alpha A \gamma \Rightarrow \alpha \beta \gamma$
 - i.e., substitute β for an occurrence of A

Derivation Example

$$\begin{aligned} S &\rightarrow S + E \mid E \\ E &\rightarrow \text{number} \mid (S) \end{aligned}$$

Derive $(1+2+(3+4))+5$:

$$\begin{aligned} S &\Rightarrow S + E \Rightarrow E + E \Rightarrow (S) + E \Rightarrow (S + E) + E \\ &\Rightarrow (S + E + E) + E \Rightarrow (E + E + E) + E \\ &\Rightarrow (1 + E + E) + E \Rightarrow (1 + 2 + E) + E \Rightarrow \dots \\ &\Rightarrow (1 + 2 + (3+4)) + E \\ &\Rightarrow (1 + 2 + (3+4)) + 5 \end{aligned}$$

Derivation Concepts

- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \rightarrow \gamma$ is a production rule
- if $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$ then
 $\alpha_1 \xrightarrow{*} \alpha_n$, we say α_1 derives α_n
- $\alpha \xrightarrow{*} \alpha$ for all α
- If $\alpha \xrightarrow{*} \beta$ and $\beta \rightarrow \gamma$ then $\alpha \xrightarrow{*} \gamma$

DEFINITION: \Rightarrow derives in one step

$\stackrel{+}{\Rightarrow}$ derives in \geq one step

$\stackrel{*}{\Rightarrow}$ derives in \geq zero steps

Derivation order

- The derivation $S \xrightarrow{*} w$ is *leftmost* if at each step the leftmost non-terminal is replaced.
- The derivation $S \xrightarrow{*} w$ is *rightmost* if at each step the rightmost non-terminal is replaced

Example

- $S \rightarrow S + E \mid E$
- $E \rightarrow \text{number} \mid (S)$

○ Left-most derivation

- $S \Rightarrow S+E \Rightarrow E+E \Rightarrow (S)+E \Rightarrow (S+E)+E \Rightarrow (S+E+E)+E$
 $\Rightarrow (E+E+E)+E \Rightarrow (1+E+E)+E \Rightarrow (1+2+E)+E$
 $\Rightarrow (1+2+(S))+E \Rightarrow (1+2+(S+E))+E \Rightarrow (1+2+(E+E))+E$
 $\Rightarrow (1+2+(3+E))+E \Rightarrow (1+2+(3+4))+E \Rightarrow (1+2+(3+4))+5$

○ Right-most derivation

- $S \Rightarrow S+E \Rightarrow E+5 \Rightarrow (S)+5 \Rightarrow (S+E)+5 \Rightarrow (S+(S))+5$
 $\Rightarrow (S+(S+E))+5 \Rightarrow (S+(S+4))+5 \Rightarrow (S+(E+4))+5$
 $\Rightarrow (S+(3+4))+5 \Rightarrow (S+E+(3+4))+5 \Rightarrow (S+2+(3+4))+5$
 $\Rightarrow (E+2+(3+4))+5 \Rightarrow (1+2+(3+4))+5$

Derivation order

Important Notes: $A \rightarrow \delta$

If $\beta A \gamma \xrightarrow{\text{lm}} \beta \delta \gamma$, what's true about β ? terminals

If $\beta A \gamma \xrightarrow{\text{rm}} \beta \delta \gamma$, what's true about γ ? terminals

Sentences

Given a grammar G with a start symbol S

- A string of *terminal* symbols that can be derived from S by applying the productions is called a *sentence* of the grammar
 - These strings are the members of set $L(G)$, the language defined by the grammar

Let G be a CFG with start symbol S . Then $S \xrightarrow{+} w$ (where w has no non-terminals) represents the language generated by G , denoted $L(G)$. So $w \in L(G) \Leftrightarrow S \xrightarrow{+} w$

w : is a sentence of G

Sentential Forms

- A string of **terminal and non-terminal** symbols that can be derived from ***S*** by applying the productions of the grammar is called a ***sentential form*** of the grammar
 - Each step of derivation forms a sentential form
 - Sentences are sentential forms with no nonterminal symbols

When $S \xrightarrow{+} \alpha$ (and α **may have NTs**) α is called a **sentential form of G.**

Example

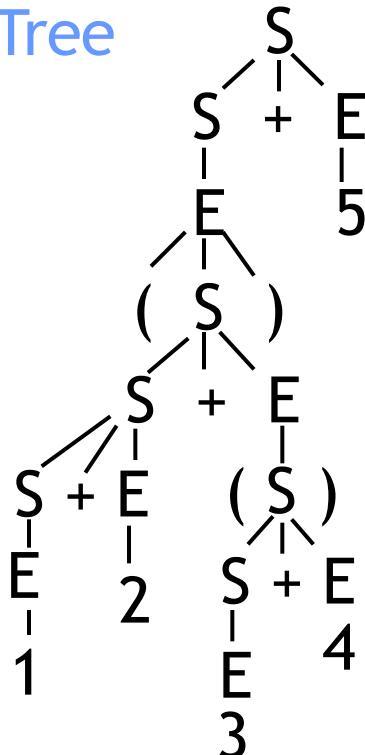
id * id is a sentence

Here's the derivation:

Sentential forms

Parse Trees & Derivations

Parse
Tree



$$\begin{aligned}S &\rightarrow S + E \mid E \\E &\rightarrow \text{number} \mid (S)\end{aligned}$$

$$(1+2+(3+4))+5$$

Derivation

$$\begin{aligned}S &\Rightarrow S + E \Rightarrow E + E \Rightarrow (S) + E \Rightarrow (S + E) + E \Rightarrow (S + E + E) + E \Rightarrow (E + E + E) + E \\&\Rightarrow (1 + E + E) + E \Rightarrow (1 + 2 + E) + E \Rightarrow \dots \Rightarrow (1 + 2 + (3 + 4)) + E \Rightarrow \\&\quad (1 + 2 + (3 + 4)) + 5\end{aligned}$$

$$E \rightarrow E \ op \ E \mid (\ E \) \mid - \ E \mid ^* \ E \mid / \ E$$

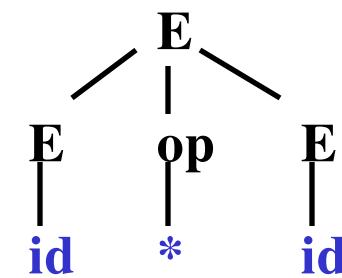
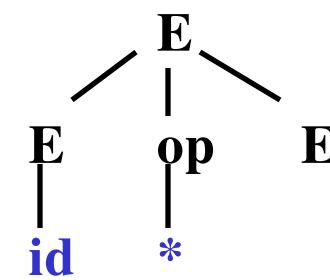
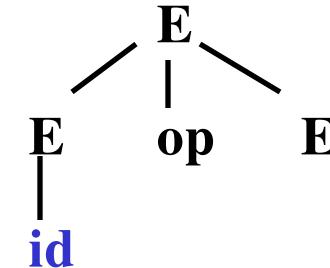
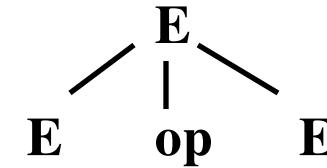
op $\rightarrow + \mid - \mid ^* \mid /$

$E \Rightarrow E \ op \ E$

$\Rightarrow id \ op \ E$

$\Rightarrow id \ * \ E$

$\Rightarrow id \ * \ id$



Parsing

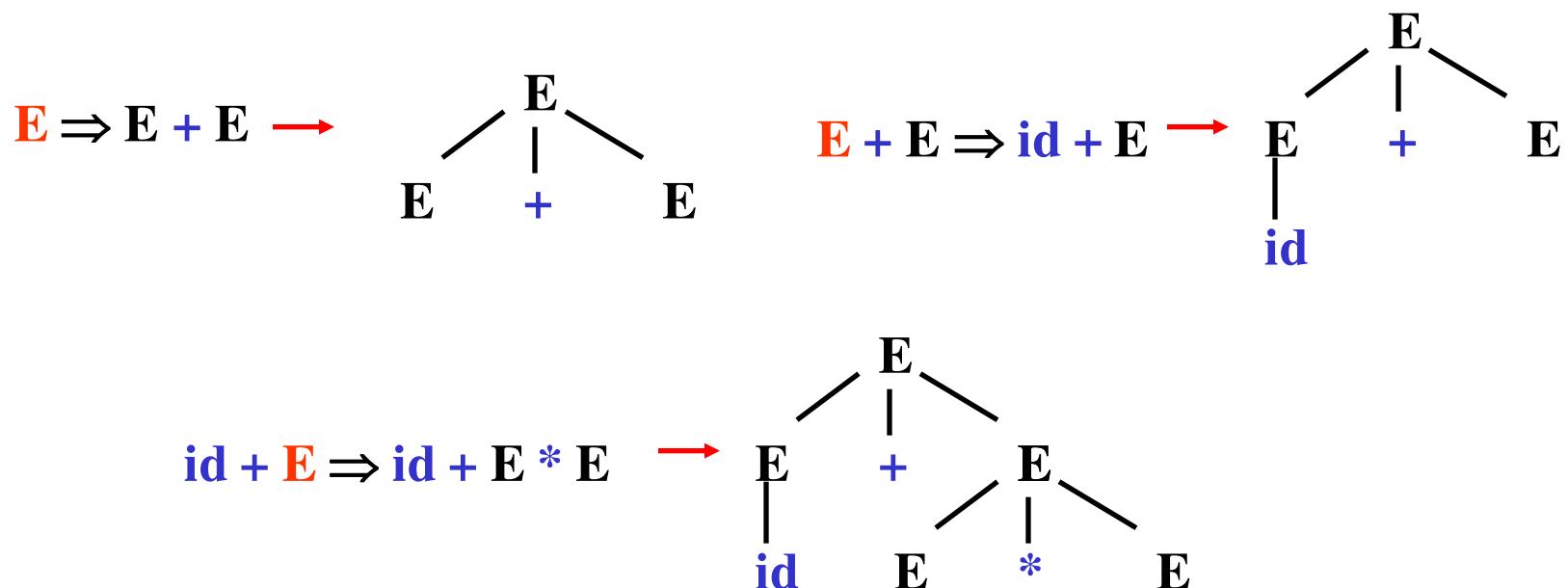
- Each parse tree has corresponding **unique** leftmost derivation & **unique** rightmost derivation
- Not every *sentence* necessarily has only one parse tree or only one leftmost or rightmost derivation

Parse Trees and Derivations

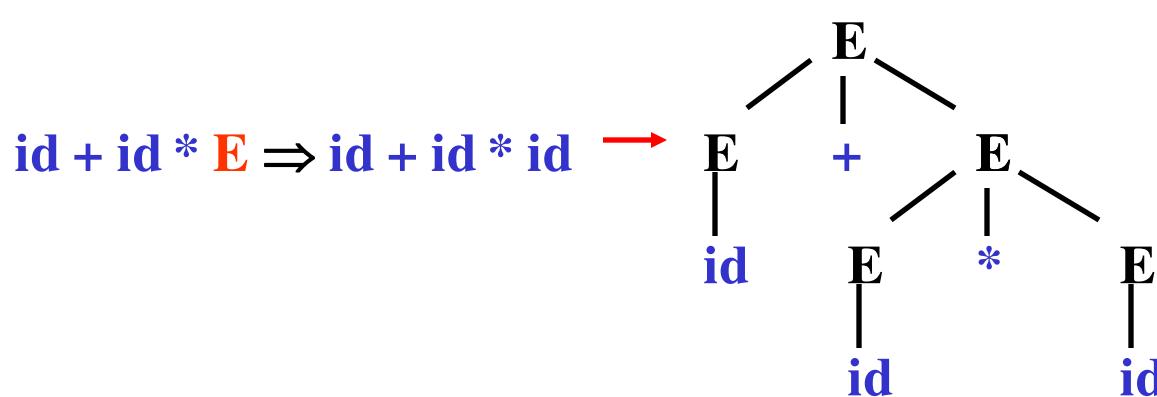
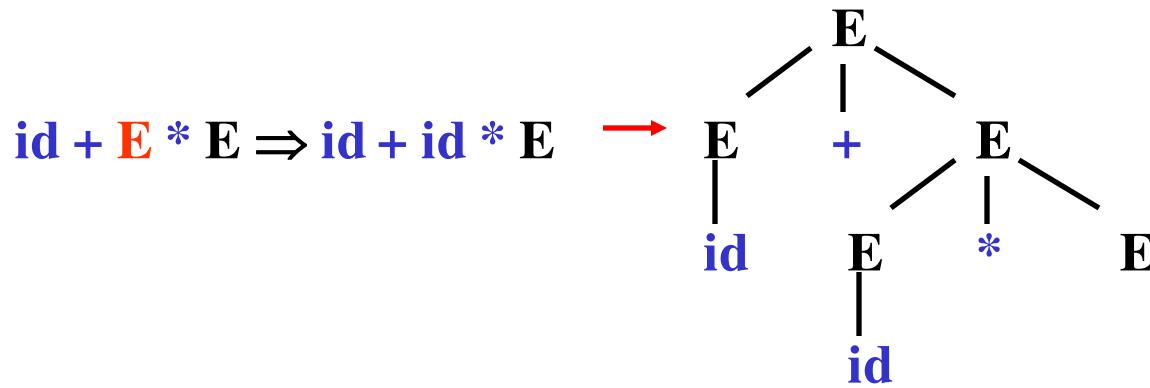
Consider the expression grammar:

$$E \rightarrow E+E \mid E^*E \mid (E) \mid -E \mid id$$

Leftmost derivations of $id + id * id$



Parse Tree & Derivations - continued



Alternative Parse Tree & Derivation

- Another leftmost derivation yields to another Parse Tree

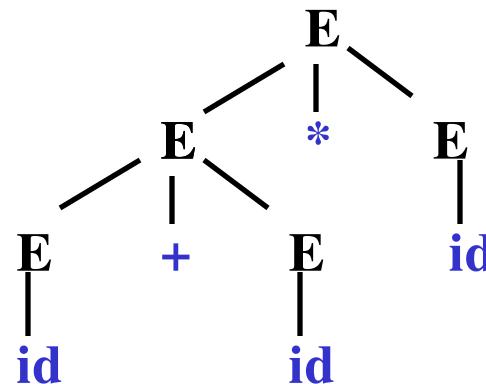
$$E \Rightarrow E * E$$

$$\Rightarrow E + E * E$$

$$\Rightarrow id + E * E$$

$$\Rightarrow id + id * E$$

$$\Rightarrow id + id * id$$



WHAT'S THE ISSUE HERE ?

Ambiguity

- A grammar G is **ambiguous** if there exists a *sentence* in G that has more than **one** derivation tree
- An **ambiguous** grammar is one that produce more than one leftmost or more than one rightmost derivation for the same sentence
- If G is ambiguous then $L(G)$ is **not** necessarily ambiguous
- A language L is **inherently ambiguous** if there is **no** un-ambiguous grammar that generates it

Writing a Grammar

- Regular Expressions : Basis of Lexical Analysis
 - Reg. Expr. → generate/represent regular languages
 - Reg. Languages → smallest, most well defined class of languages
- Context Free Grammars: Basis of Parsing
 - CFGs → represent context free languages
 - CFLs → contain more powerful languages



$a^n b^n$ – CFL that's
not regular!

Try to write DFA/NFA for it !

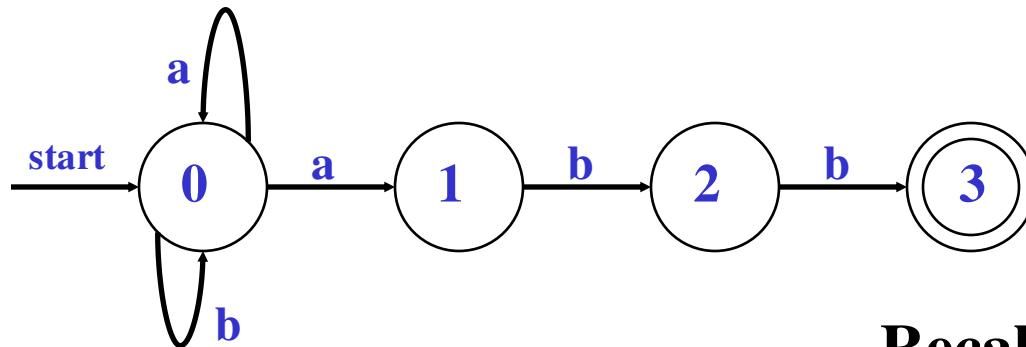
Writing a Grammar

- Given a CFG $G = (NT, T, S, P)$, the context-free language generated by G is given by
 - $L(G) = \{w \mid w \in T^* \text{ and } S \xrightarrow{+} w\}$

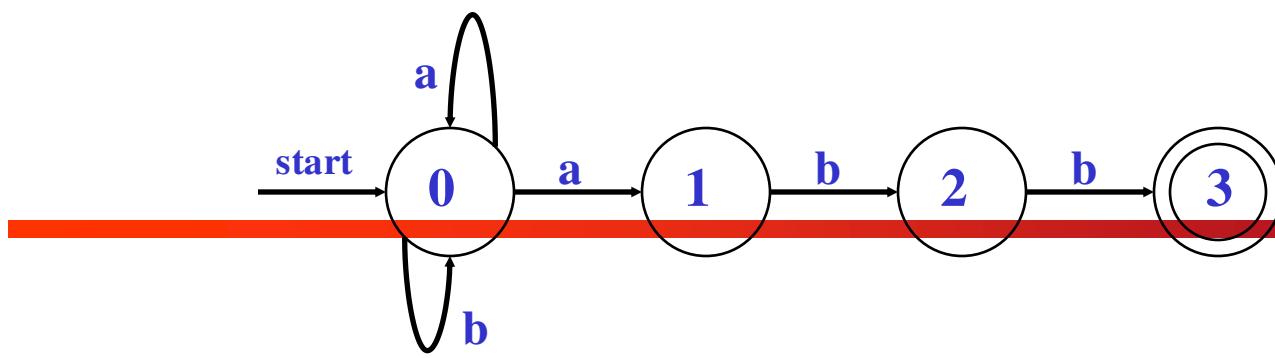
- Strings in $L(G)$ may contain only terminals
 - We say a string of terminals w is in $L(G)$
 - iff $S \xrightarrow{+} w$
 - The string w is called a *sentence* of $L(G)$

REs vs. CFGs

Since Reg. Lang. \subset Context Free Lang., it is possible to go from REs to CFGs via NFA



Recall: $(a \mid b)^*abb$



Construct CFG as follows:

1. Each State i has non-terminal A_i : A_0, A_1, A_2, A_3
2. If then add $A_i \rightarrow a A_j$
3. If then add $A_i \rightarrow A_j$
4. If i is an accepting state, add $A_i \rightarrow \epsilon$: $A_3 \rightarrow \epsilon$
5. If i is a starting state, then A_i is the start symbol : A_0

$T = \{a, b\}$, $NT = \{A_0, A_1, A_2, A_3\}$, $S = A_0$

$PR = \{ A_0 \rightarrow aA_0 \mid aA_1 \mid bA_0 ;$
 $A_1 \rightarrow bA_2 ; A_2 \rightarrow bA_3 ; A_3 \rightarrow \epsilon \}$

Grammar Problems

- Ambiguity:
 - more than one parsing tree for a sentence
- Left recursion:
 - $A \xrightarrow{+} A\alpha$
- Left factoring:
 - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
- ϵ -production:
 - $A \rightarrow \epsilon$
- Cycles:
 - $A \xrightarrow{+} A$

Eliminating Ambiguity (1)

Consider the following grammar segment:

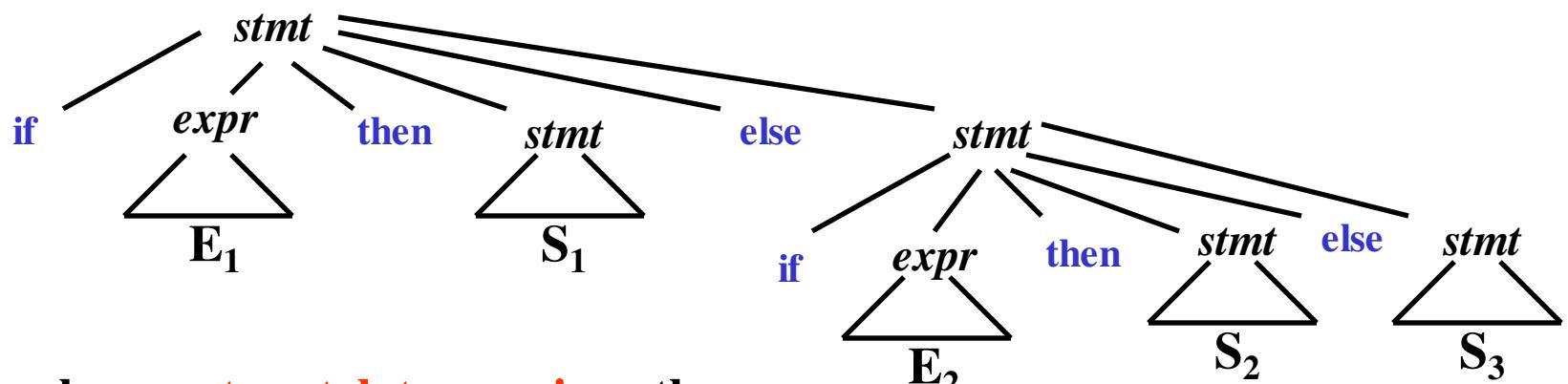
$stmt \rightarrow \text{if } expr \text{ then } stmt$

| $\text{if } expr \text{ then } stmt \text{ else } stmt$

| **other (any other statement)**

Let's consider a simple parse tree:

if E₁ then S₁ else if E₂ then S₂ else S₃



else must match to previous then

Eliminating Ambiguity (2)

if E₁ then if E₂ then S₁ else S₂

How is this parsed ?

if E₁ then
 if E₂ then
 S₁
 else
 S₂

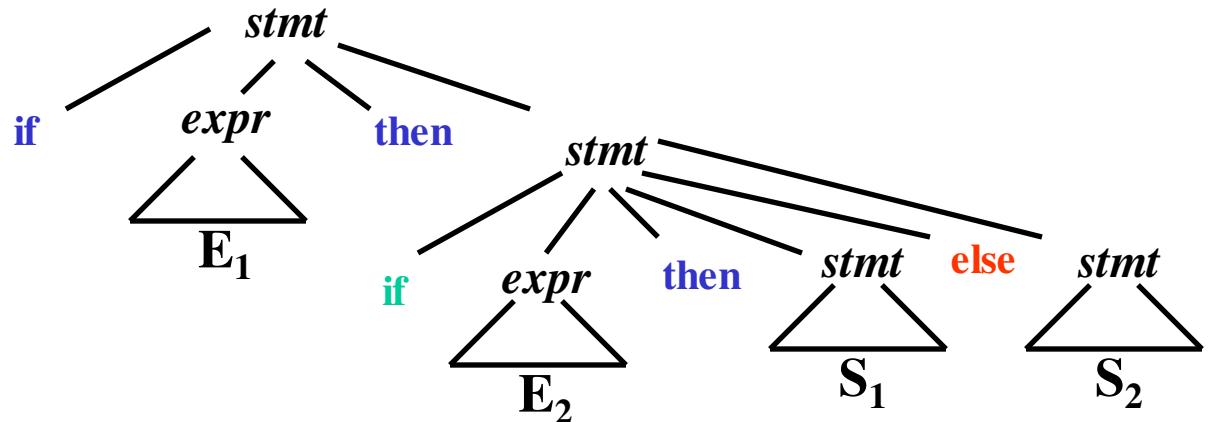
vs.

if E₁ then
 if E₂ then
 S₁
 else
 S₂

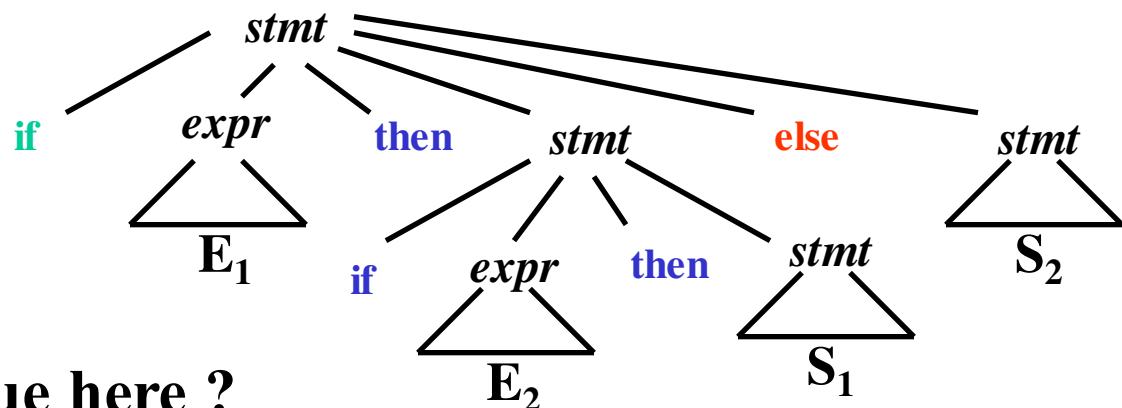
What's the issue here ?

Eliminating Ambiguity (3)

Form 1:



Form 2:



What's the issue here ?

Eliminating Ambiguity (4)

Take Original Grammar:

$$\begin{aligned}stmt \rightarrow & \text{ if } expr \text{ then } stmt \\& | \text{ if } expr \text{ then } stmt \text{ else } stmt \\& | \text{ other } \quad \quad \quad \text{(any other statement)}\end{aligned}$$

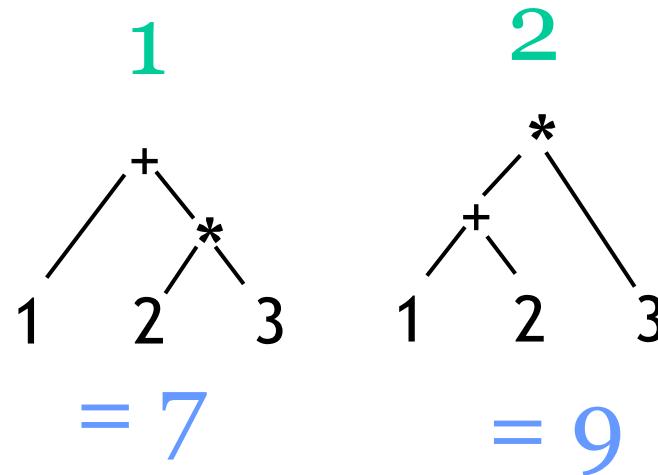
Revise to remove ambiguity:

$$\begin{aligned}stmt \rightarrow & \text{ matched_stmt } | \text{ unmatched_stmt } \\matched_stmt \rightarrow & \text{ if } expr \text{ then } matched_stmt \text{ else } matched_stmt | \text{ other } \\unmatched_stmt \rightarrow & \text{ if } expr \text{ then } stmt \\& | \text{ if } expr \text{ then } matched_stmt \text{ else } unmatched_stmt\end{aligned}$$

How does this grammar work ?

Eliminating Ambiguity (6)

- $S \rightarrow S + S \mid S * S \mid \text{num}$
- Consider expression $1 + 2 * 3$
 - Derivation 1: $S \Rightarrow S + S \Rightarrow 1 + S \Rightarrow 1 + S * S \Rightarrow 1 + 2 * S \Rightarrow 1 + 2 * 3$
 - Derivation 2: $S \Rightarrow S * S \Rightarrow S * 3 \Rightarrow S + S * 3 \Rightarrow S + 2 * 3 \Rightarrow 1 + 2 * 3$
- Different parse trees correspond to different evaluations



Eliminating Ambiguity (7)

- $S \rightarrow S + S \mid S * S \mid \text{num}$
 - is ambiguous since both * and + operators have the same precedence

- Often can eliminate ambiguity by adding non-terminals

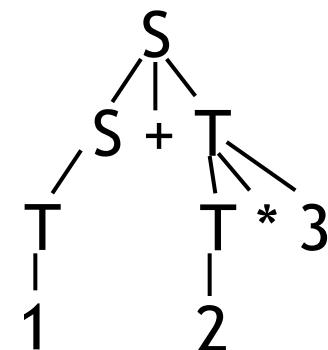
$$S \rightarrow S + T \mid T$$

+ & * are left-associative

$$T \rightarrow T * \text{num} \mid \text{num}$$

* has higher precedence

- See Example in Pages 30 & 31
- $1 + 2 * 3 = 7$



Eliminating Ambiguity (8)

- $S \rightarrow S + S \mid S * S \mid \text{num}$
 - is ambiguous since both * and + operators have the same precedence

- Often can eliminate ambiguity by adding non-terminals

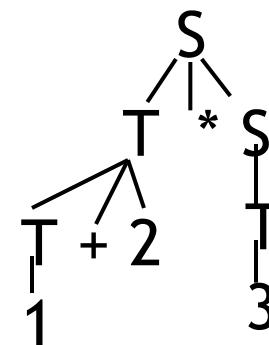
$$S \rightarrow T * S \mid T$$

+ & * are right-associative

$$T \rightarrow \text{num} + T \mid \text{num}$$

+ has higher precedence

- $1 + 2 * 3 = 9$



Elimination of Left Recursion (1)

A left recursive grammar has rules that support the derivation : $A \xrightarrow{*} A\alpha$, for some α

Top-Down parsing can't handle this type of grammar,

- **Left Recursion - Avoid infinite loop when choosing rules**

$$A \rightarrow A\alpha \mid \beta$$

$$A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow A\alpha\alpha\alpha \dots \text{etc.}$$

Take left recursive grammar:

$$A \rightarrow A\alpha \mid \beta$$

To the following:

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

Elimination of Left Recursion (2)

Informal Discussion:

Take all productions for A and order as:

$$A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_m | \beta_1 | \beta_2 | \dots | \beta_n$$

Where no β_i begins with A

Now apply concepts of previous slide:

$$A \rightarrow \beta_1 A' | \beta_2 A' | \dots | \beta_n A'$$

$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \dots | \alpha_m A' | \in$$

For our example:

$$\begin{array}{l} E \rightarrow E + T \mid T \xrightarrow{\quad} \left\{ \begin{array}{l} E \rightarrow TE' \\ E' \rightarrow + TE' \mid \in \end{array} \right. \\ T \rightarrow T * F \mid F \xrightarrow{\quad} \left\{ \begin{array}{l} T \rightarrow FT' \\ T' \rightarrow * FT' \mid \in \end{array} \right. \\ F \rightarrow (E) \mid id \xrightarrow{\quad} F \rightarrow (E) \mid id \end{array}$$

Elimination of Left Recursion (3)

Problem: If left recursion is two-or-more levels deep, this isn't enough

$$\left. \begin{array}{l} S \rightarrow Aa \mid b \\ A \rightarrow Ac \mid Sd \mid \epsilon \end{array} \right\} \quad S \Rightarrow Aa \Rightarrow Sda$$

Algorithm:

Input: Grammar G with no cycles or ϵ -productions

Output: An equivalent grammar with no left recursion

1. Arrange the non-terminals in some order A_1, A_2, \dots, A_n
2. `for i := 1 to n do begin`
`for j := 1 to i - 1 do begin`
 replace each production of the form $A_i \rightarrow A_j \gamma$
 by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$
 where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ are all current A_j productions;
`end`
 eliminate the immediate left recursion among A_i productions
`end`

Elimination of Left Recursion (4)

Apply the algorithm to:

$$S \rightarrow Aa \mid b$$

$$A_1 \rightarrow A_2a \mid b$$

$$A \rightarrow Ac \mid Sd \mid \epsilon$$

$$A_2 \rightarrow A_2c \mid A_1d \mid \epsilon$$

i = 1

For A_1 there is no left recursion

i = 2

for j=1 to 1 do

Take productions: $A_2 \rightarrow A_1\gamma$ and replace with

$$A_2 \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$$

where $A_1 \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ are A_1 productions

in our case $A_2 \rightarrow A_1d$ becomes $A_2 \rightarrow A_2ad \mid bd$

What's left:

$$A_1 \rightarrow A_2a \mid b$$

$$A_2 \rightarrow A_2c \mid A_2ad \mid bd \mid \epsilon$$

Are we done ?

Elimination of Left Recursion (5)

No ! We must still remove A_2 left recursion !

$$A_1 \rightarrow A_2 \mathbf{a} \mid \mathbf{b}$$

$$A_2 \rightarrow A_2 \mathbf{c} \mid A_2 \mathbf{ad} \mid \mathbf{bd} \mid \epsilon$$

Recall:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$$

Apply to above case. What do you get ?

$$A_1 \rightarrow A_2 \mathbf{a} \mid \mathbf{b}$$

$$A_2 \rightarrow \mathbf{bd} A_3 \mid A_3$$

$$A_3 \rightarrow \mathbf{c} A_3 \mid \mathbf{ad} A_3 \mid \epsilon$$

Left Factoring

Problem : Uncertain which of 2 rules to choose:

$$\begin{aligned}stmt \rightarrow & \underline{\text{if } expr \text{ then } stmt} \text{ else } stmt \\& | \underline{\text{if } expr \text{ then } stmt}\end{aligned}$$

When do you know which one is valid ?

What's the general form of *stmt* ?

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

$$\alpha : \text{if } expr \text{ then } stmt$$

$$\beta_1 : \text{else } stmt \quad \beta_2 : \in$$

Transform to:

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

EXAMPLE:

$$stmt \rightarrow \text{if } expr \text{ then } stmt \text{ rest}$$

$$rest \rightarrow \text{else } stmt \mid \in$$

Removing \in -Productions (1)

Very Simple: $A \rightarrow \in$ and $B \rightarrow uAv$ implies add $B \rightarrow uv$ to the grammar G.

Why does this work ?

Examples:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow + TE' \mid \in \\ T &\rightarrow FT' \\ T' &\rightarrow * FT' \mid \in \\ F &\rightarrow (E) \mid id \end{aligned}$$

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow + TE' \mid \in \\ E' &\rightarrow + TE' \mid \in \\ E' &\rightarrow + T \end{aligned}$$


Non-Context-Free Language Constructs

Note: Not all aspects of a programming language can be represented by context free grammars / languages.

Examples:

- 1. Declaring ID before its use**
- 2. Valid typing within expressions**
- 3. Parameters in definition vs. in call**

These features are called **context-sensitive** and define yet another language class, **CSL**.



Non-Context-Free Language Constructs

- $G = (\{S\}, \{a,b\}, S, \{S \rightarrow aSb, S \rightarrow ab\})$
 - What is $L(G)$? $L(G) = \{a^n b^n \mid n \geq 1\}$
 - $G = (\{S\}, \{a,b\}, S, \{S \rightarrow aSb, S \rightarrow \epsilon\})$
- What is the CFG for this language?
 - $L = \{a^n b^n c^n \mid n \geq 1\}$
 - L is NOT CFL. It is CSL
- What about this language?
 - $L = \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$
 - L is a CFL
 - $G = (\{S\}, \{a,b\}, S, \{S \rightarrow aSd \mid aAd \rightarrow bAc \mid bc\})$

Context-Sensitive Languages - Examples

The following languages are NOT CFLs. They are CSLs

$$L_1 = \{ wcw \mid w \text{ is in } (a \mid b)^* \}$$

$$L_2 = \{ a^n b^m c^n d^m \mid n \geq 1, m \geq 1 \}$$

$$L_3 = \{ a^n b^n c^n \mid n \geq 0 \}$$

Write a CFG for the following Languages

$$L'_1 = \{ w c w^R \mid w \text{ is in } (a \mid b)^* \}$$

$$L'_2 = \{ a^n b^m c^m d^n \mid n \geq 1, m \geq 1 \}$$

$$L''_2 = \{ a^n b^n c^m d^m \mid n \geq 1, m \geq 1 \}$$

$$L'_3 = \{ a^n b^n \mid n \geq 1 \}$$

Solutions

$L_1 = \{ w c w^R \mid w \text{ is in } (a \mid b)^* \}$

$S \rightarrow a S a \mid b S b \mid c$

$L'_2 = \{ a^n b^m c^m d^n \mid n \geq 1, m \geq 1 \}$

$S \rightarrow a S d \mid a A d$

$A \rightarrow b A c \mid b c$

$L''_2 = \{ a^n b^n c^m d^m \mid n \geq 1, m \geq 1 \}$

$S \rightarrow X Y$

$X \rightarrow a X b \mid a b$

$Y \rightarrow c Y d \mid c d$

$L'_3 = \{ a^n b^n \mid n \geq 1 \}$

$S \rightarrow a S b \mid a b$

Compilers Construction

Chapter 4

Top-Down Parsing

Dr. Mohammed Salem Atoum

M.Atoum@inu.edu.jo

Top-Down Parsing

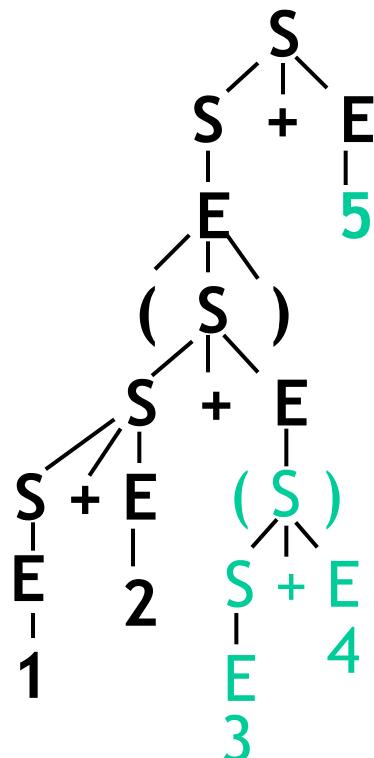
- Recursive-descent parsing
- Predictive parsing
- Non-Recursive Predictive parsing
- Error recovery

Top-down Parsing

- We normally scan in tokens from left to right
- Left-most derivation reflects top-down parsing
 - Start with the start symbol
 - End with the string of tokens
- $S \Rightarrow S+E \Rightarrow E+E \Rightarrow (S)+E \Rightarrow (S+E)+E$
 $\Rightarrow (S+E+E)+E \Rightarrow (E+E+E)+E \Rightarrow (1+E+E)+E$
 $\Rightarrow (1+2+E)+E \Rightarrow (1+2+(S))+E \Rightarrow (1+2+(S+E))+E$
 $\Rightarrow (1+2+(E+E))+E \Rightarrow (1+2+(3+E))+E$
 $\Rightarrow (1+2+(3+4))+E \Rightarrow (1+2+(3+4))+5$

Top-down parsing

$S \Rightarrow S+E \Rightarrow E+E \Rightarrow (S)+E \Rightarrow (S+E)+E \Rightarrow (S+E+E)+E \Rightarrow (E+E+E)+E$
 $\Rightarrow (1+E+E)+E \Rightarrow (1+2+E)+E \dots$



Recursive Descent Parsing

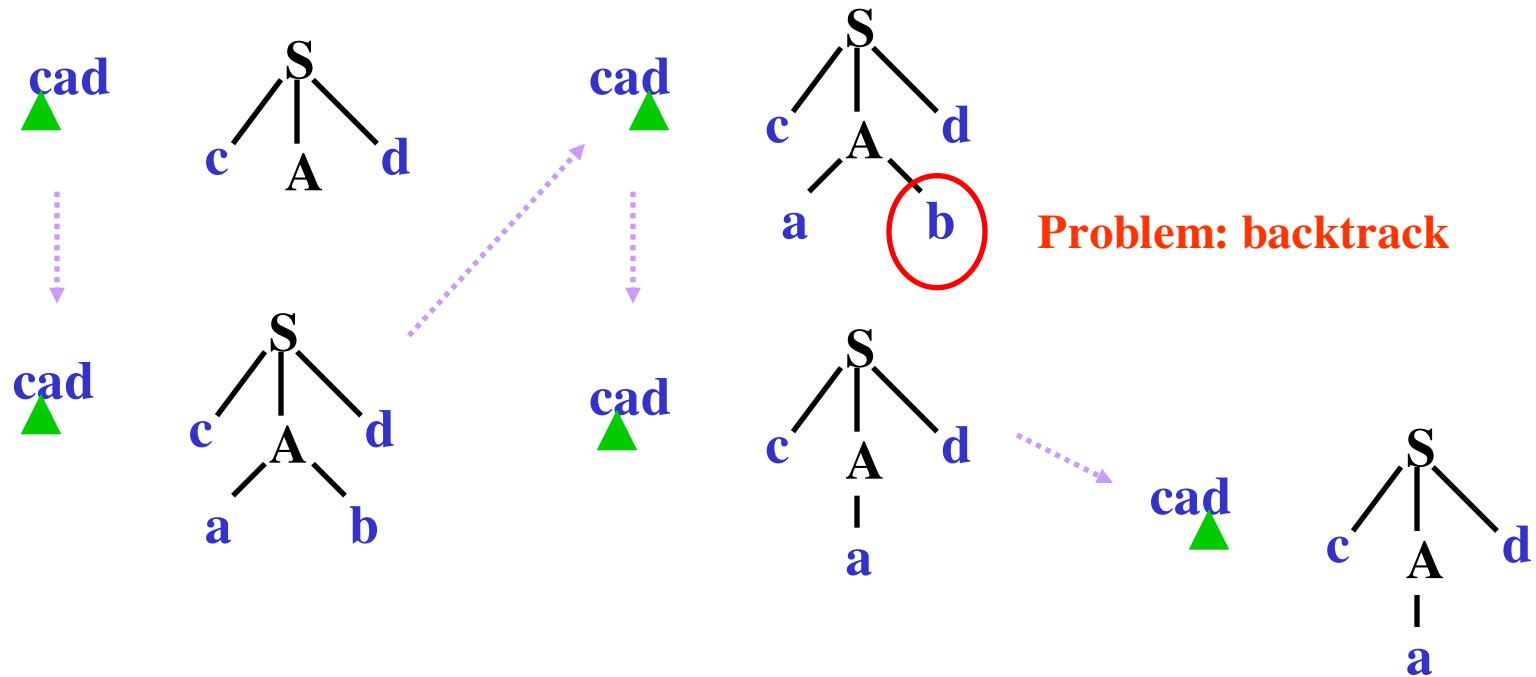
- A top-down parsing method
- Use a set of *mutually recursive* procedures (**one procedure for each nonterminal symbol**)
 - Start the parsing process by calling the procedure that corresponds to the start symbol
 - Each production becomes one clause in the procedure

Recursive Descent Parsing

- Choose production rule based on input symbol
- The current token being scanned in the input is referred to as the *lookahead* symbol.
- We may have to try a production and **backtrack** to try another production if the first is found to be unsuitable

Recursive Descent Parsing

- Example: $\begin{array}{l} S \rightarrow c A d \\ A \rightarrow ab \mid a \end{array}$ } input: cad



Recursive Descent Parsing

array [num dotdot num] of integer

```
type → simple
  | ↑ id
  | array [ simple ] of type
simple → integer
  | char
  | num dotdot num
```

```
procedure match ( t : token ) ;
begin
  if lookahead = t then
    lookahead := nexttoken()
  else error
end ;
```

Top-Down Algorithm (Continued)

```
procedure type ;
begin
    if lookahead is in { integer, char, num } then simple
    else if lookahead = ‘↑’ then begin match(‘↑’); match(id) end
    else if lookahead= array then begin
        match(array); match(‘[’); simple; match(‘]’); match(of); type
        end
    else error
end ;
procedure simple ;
begin
    if lookahead= integer then match( integer );
    else if lookahead= char then match( char );
    else if lookahead= num then begin
        match(num); match(dotdot); match(num)
        end
    else error
end ;
```

Problem with Recursive-Descent Parser

- **Left Recursion** in CFG May Cause Parser to Loop Forever
- Solution: Algorithm to Remove Left Recursion

$$expr \rightarrow expr + term \mid expr - term \mid term$$
$$term \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$
$$expr \rightarrow term \ rest$$
$$rest \rightarrow + term \ rest \mid - term \ rest \mid \epsilon$$
$$term \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Predictive Parsing

- It is a special form of recursive descent parsing
- In which **backtracking** does not occur
- Look-ahead symbol **unambiguously** determines the procedure selected for each nonterminal
 - Use a lookahead symbol to decide which production to use
- It consists of procedures for **each nonterminal**

Predictive Parsing

- To eliminate backtracking, what must we do/be sure of for grammar?
 - no left recursion
 - apply left factoring
- When grammar satisfies above conditions, we can utilize current input symbol in conjunction with non-terminals to be expanded to uniquely determine the next action

Predictive Parsing

- It relies on information about what first symbols can be generated by the **right side** of a production
- Let α be the **right side** of a production for nonterminal A
- We define **FIRST(α)** to be the set of tokens that appear as the first symbols of one or more strings generated from α
- If α is ϵ or can generate ϵ , then ϵ is also in **FIRST(α)**

Predictive Parsing

- The **FIRST** sets must be considered if there are two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$
- Recursive-descent parsing **without backtracking** (predictive parsing) requires $\text{FIRST}(\alpha)$ and $\text{FIRST}(\beta)$ to be disjoint
- The lookahead can be then used to decide which production to use

Predictive Parsing

- If the lookahead symbol is in $\text{FISRT}(\alpha)$, then α is used, otherwise
- If the lookahead symbol is in $\text{FISRT}(\beta)$, then β is used
- $\text{FIRST}(simple)$
 - = {integer, char, num}
- $\text{FIRST}(\uparrow id)$
 - = { \uparrow }
- $\text{FIRST}(\text{array } [simple] \text{ of type})$
 - = {array}

$type \rightarrow simple$
$\uparrow id$
array [simple] of type
$simple \rightarrow integer$
char
num dotdot num

Transition Diagrams for Predictive Parsers

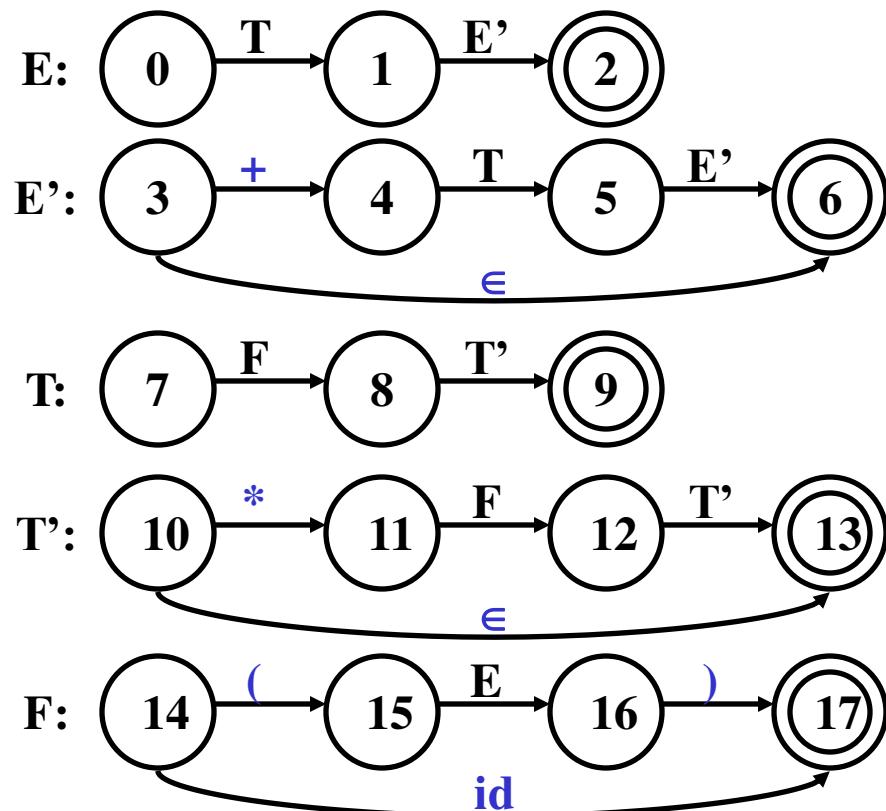
- The labels of edges are *terminals* and *nonterminals*
- A transition on terminal means *we should take that transition* if that terminal is the *next input symbol*
- A transition on non-terminal A is a *call of the procedure of A*
- For each non-terminal of the grammar do following:
 - 1. Create an initial and final (return) state
 - 2. If $A \rightarrow X_1X_2\dots X_n$ is a production, add path with edges X_1, X_2, \dots, X_n

Transition Diagrams for Predictive Parsers

- Once transition diagrams have been developed, apply a straightforward technique to algorithmicize transition diagrams with procedure and possible recursion

Transition Diagrams

$$\begin{array}{c} E \rightarrow TE' \\ E' \rightarrow + TE' | \quad \in \\ T \rightarrow FT' \\ T' \rightarrow * FT' | \quad \in \\ F \rightarrow (E) | id \end{array}$$



How are Transition Diagrams Used ?

```
main()
{
    TD_E();
}
```

```
TD_E()
{
    TD_T();
    TD_E'();
}
```

```
TD_T()
{
    TD_F();
    TD_T'();
}
```

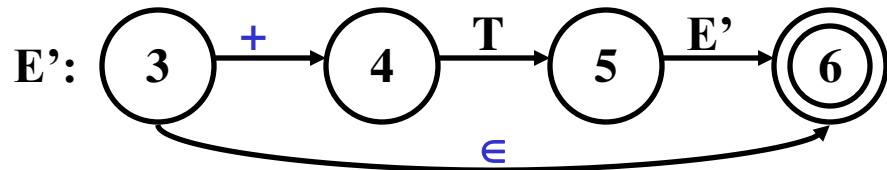
```
TD_E'()
{
    token = get_token();
    if token = '+' then
        { TD_T(); TD_E'(); }
}
```

```
TD_F()
{
    token = get_token();
    if token = '(' then
        { TD_E(); match(')'); }
    else
        if token.value <> id then
            {error + EXIT}
        else
            ...
}
```

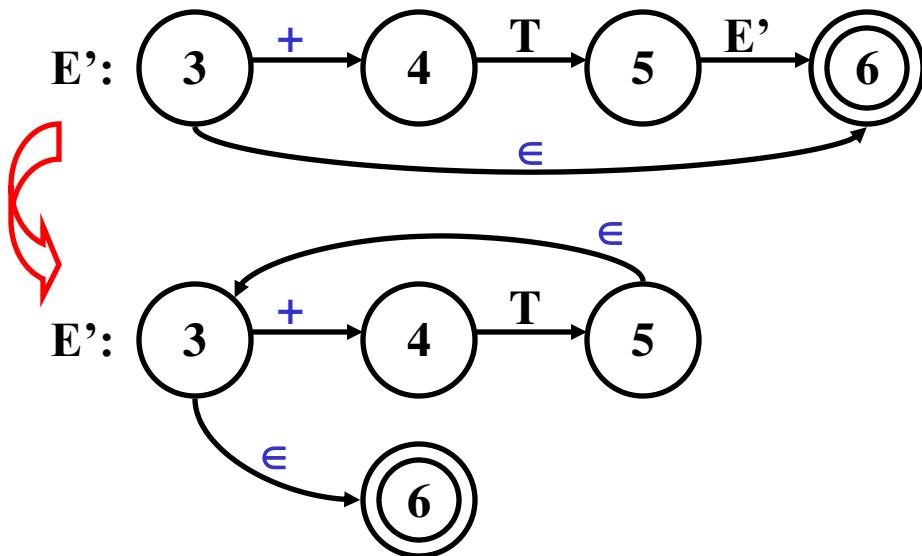
```
TD_T'()
{
    token = get_token();
    if token = '*' then
        { TD_F(); TD_T'(); }
}
```

What
happened to
 ϵ -moves?

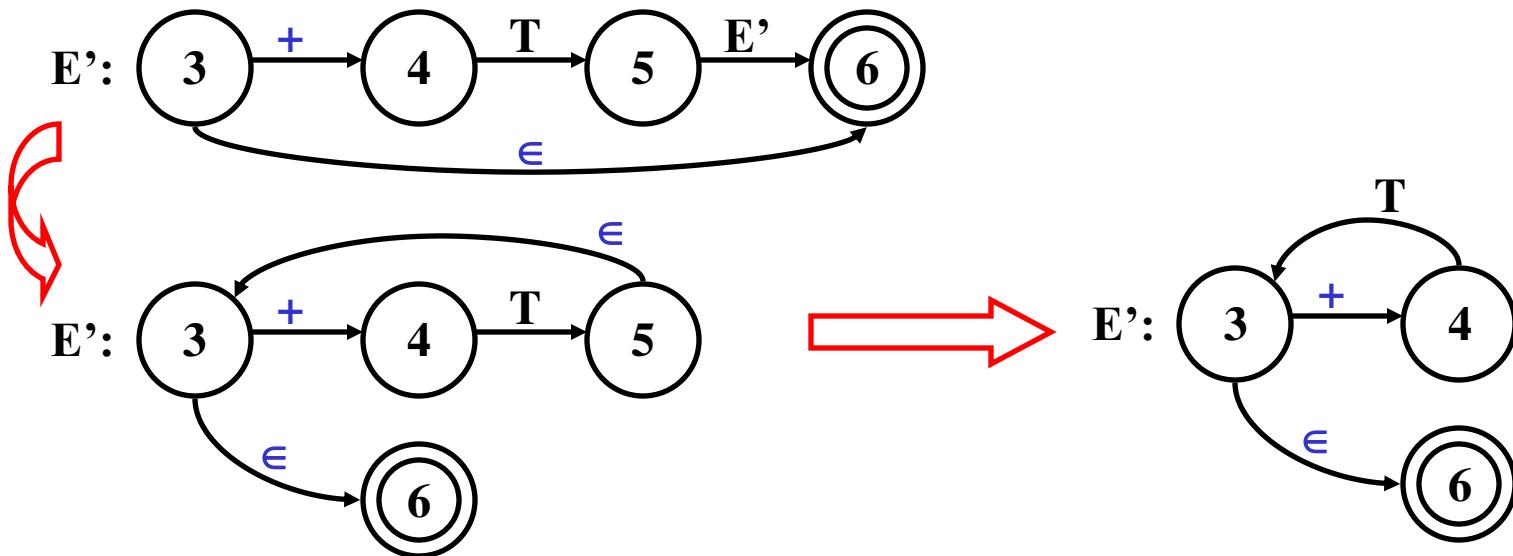
How can Transition Diagrams be Simplified ?



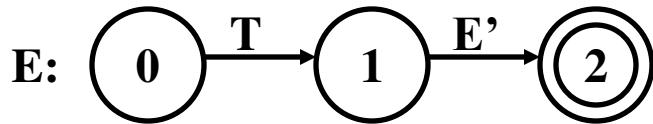
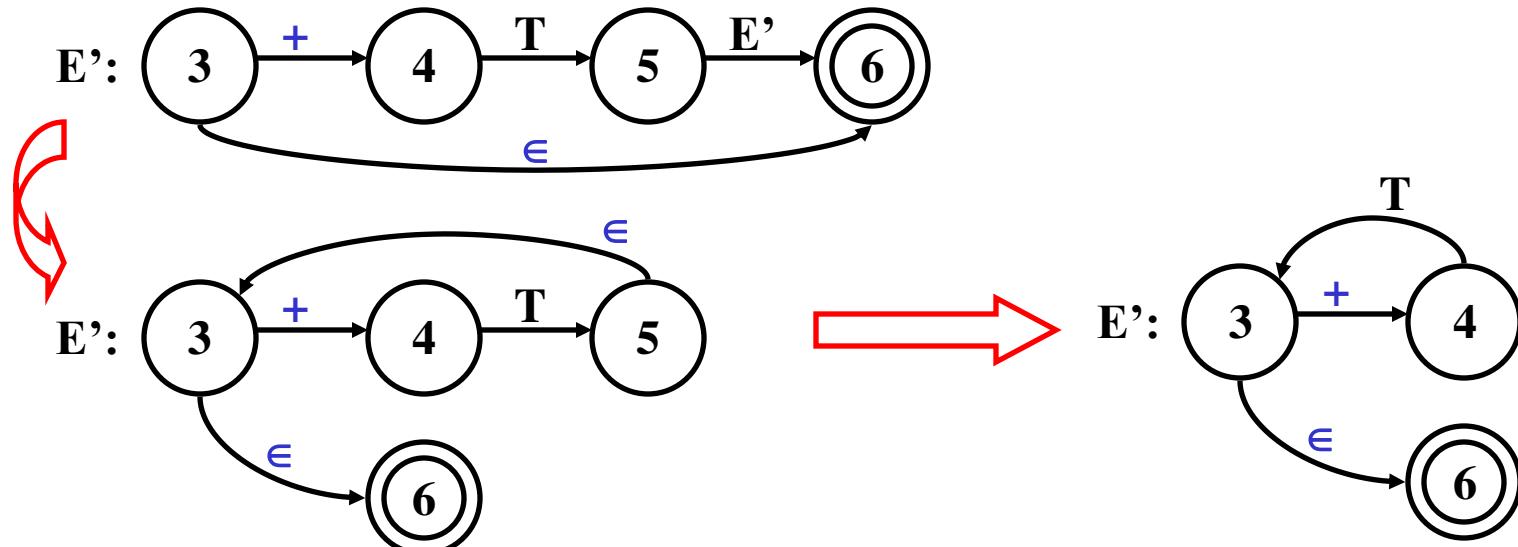
How can Transition Diagrams be Simplified ? (2)



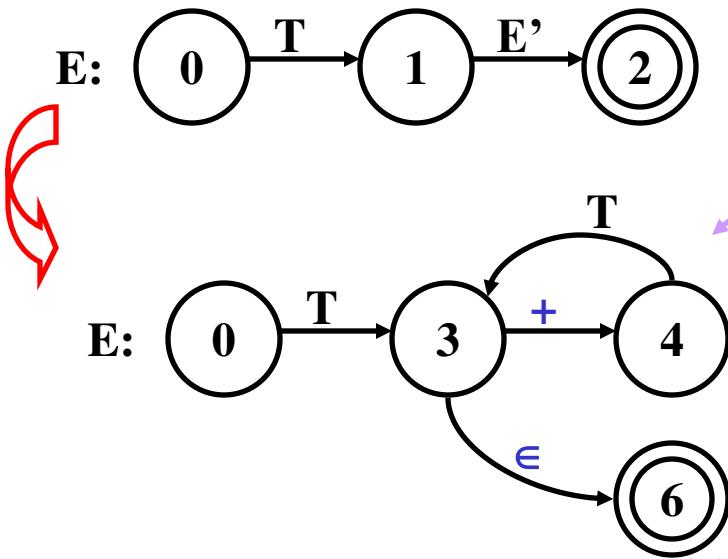
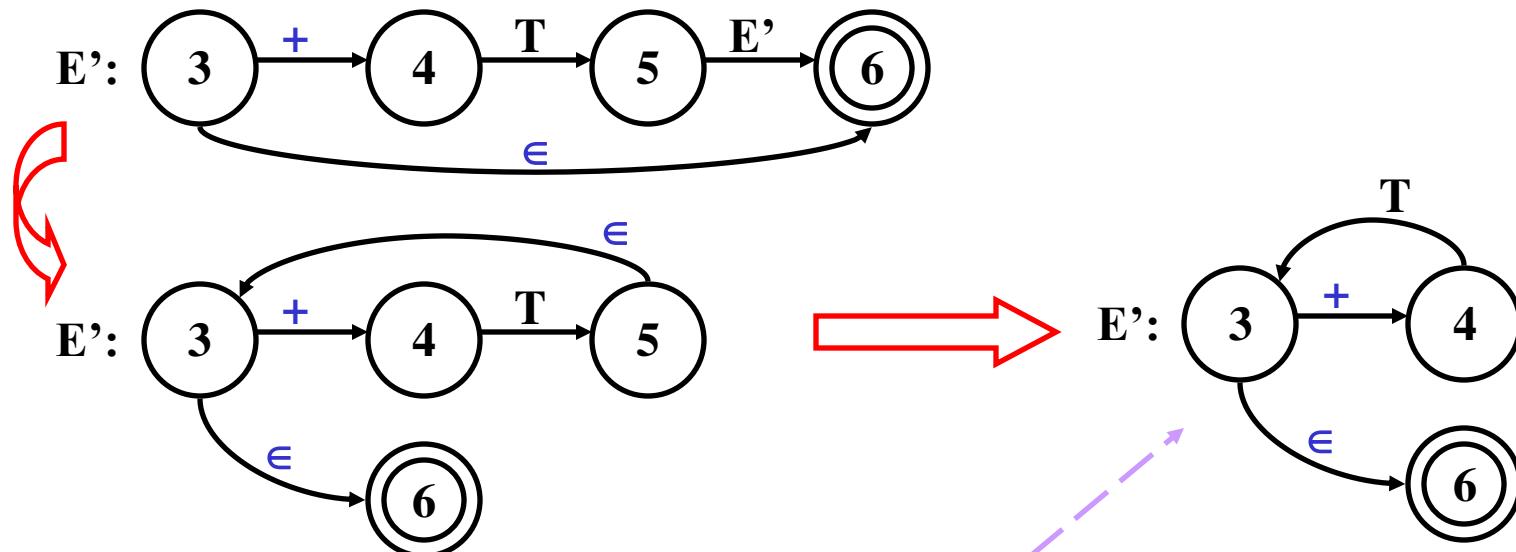
How can Transition Diagrams be Simplified ? (3)



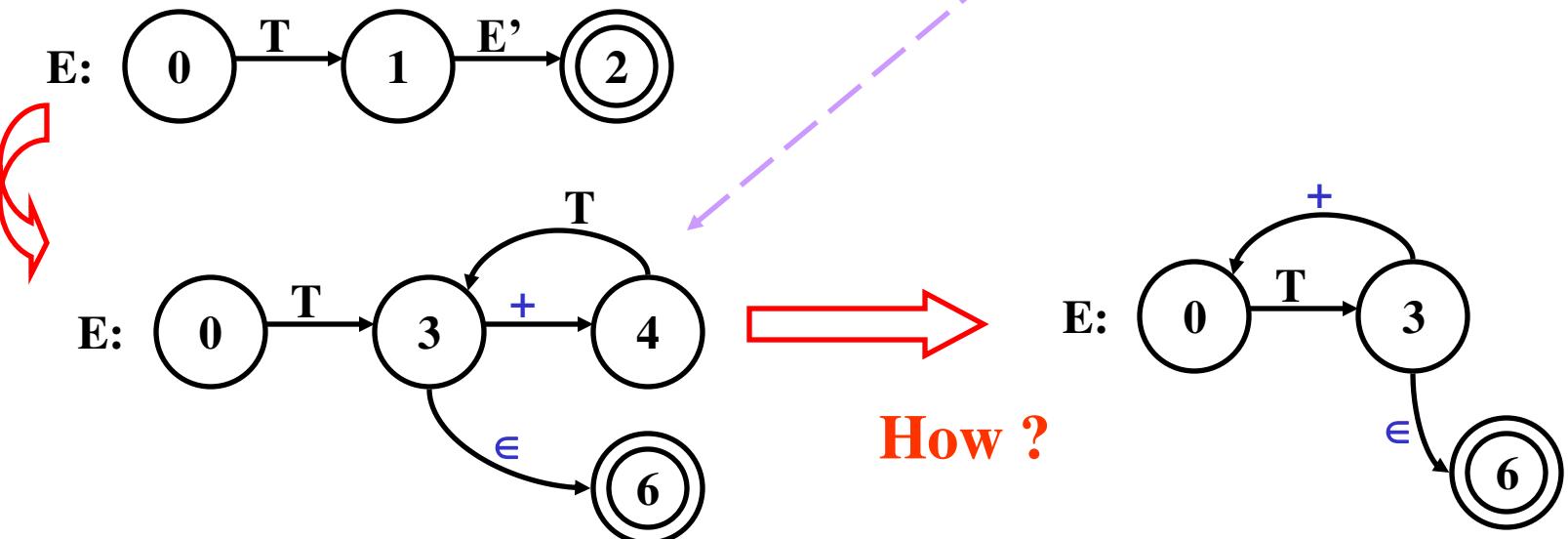
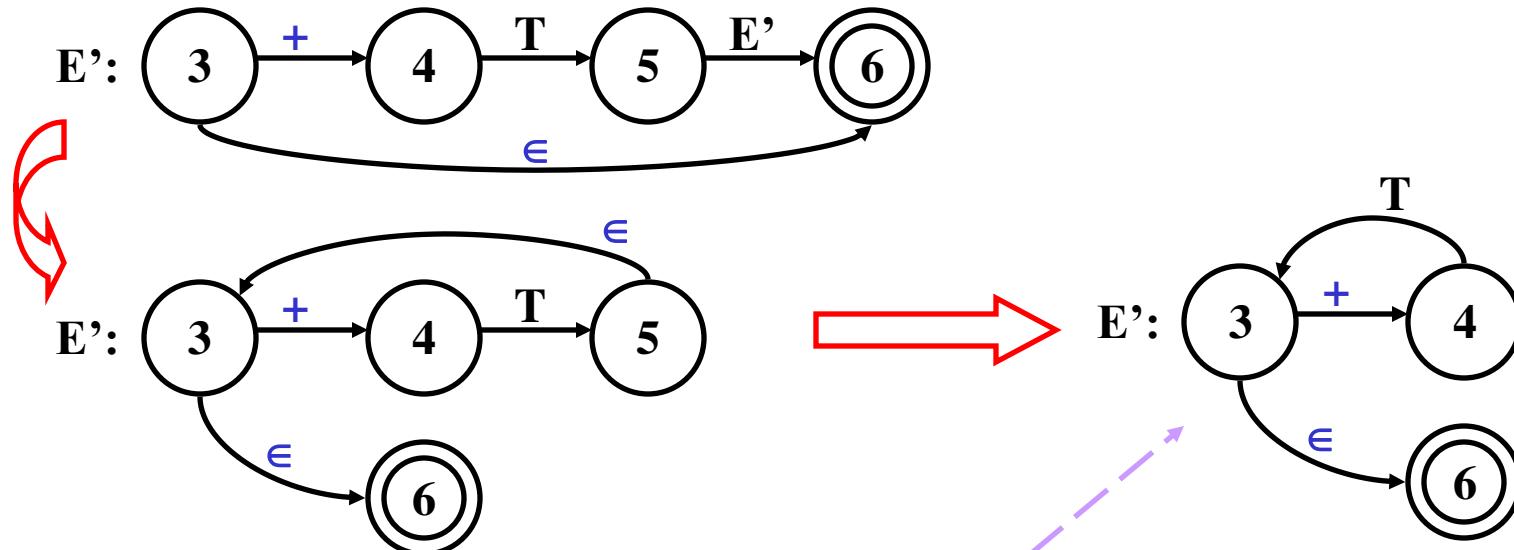
How can Transition Diagrams be Simplified ? (4)



How can Transition Diagrams be Simplified ? (5)

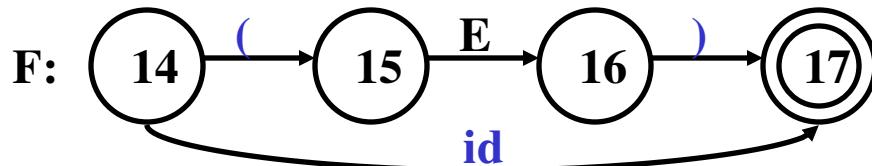
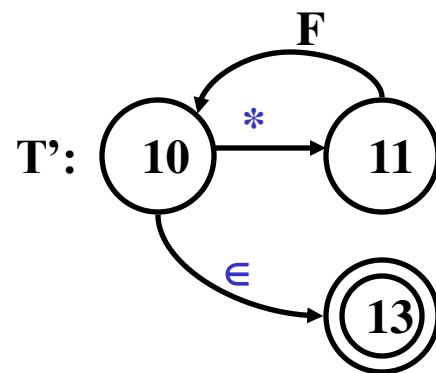
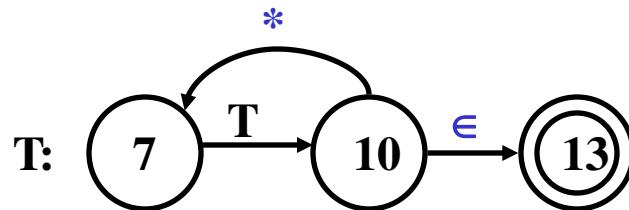


How can Transition Diagrams be Simplified ? (6)



Additional Transition Diagram Simplifications

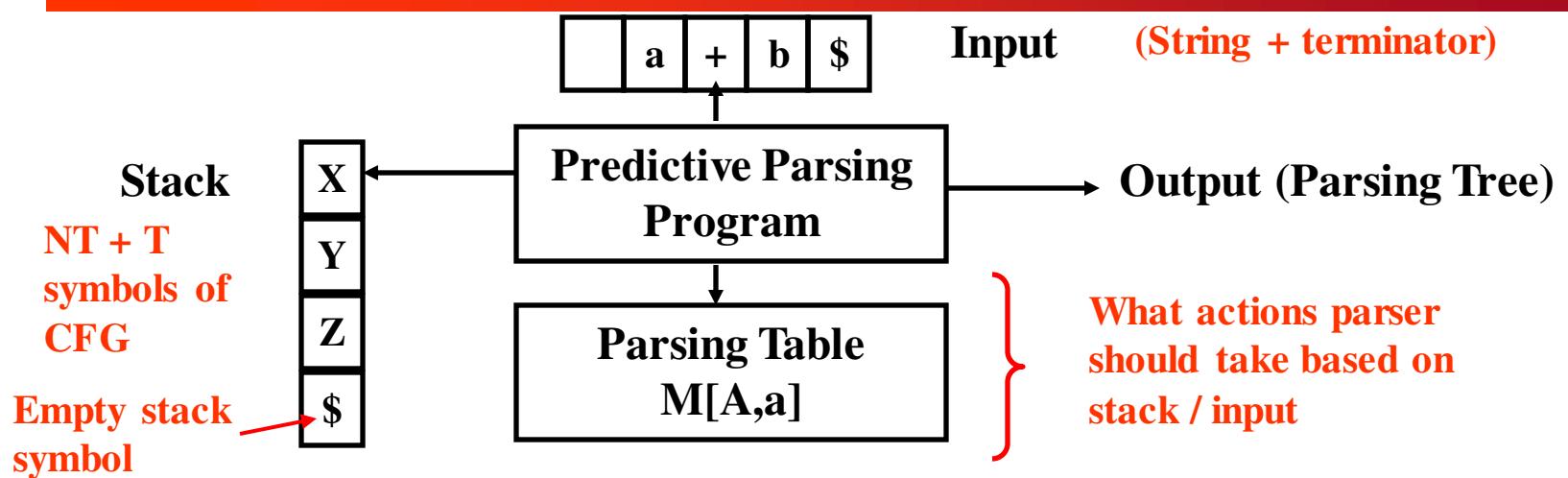
- Similar steps for T and T'
- Simplified Transition diagrams:



Nonrecursive Predictive Parsing

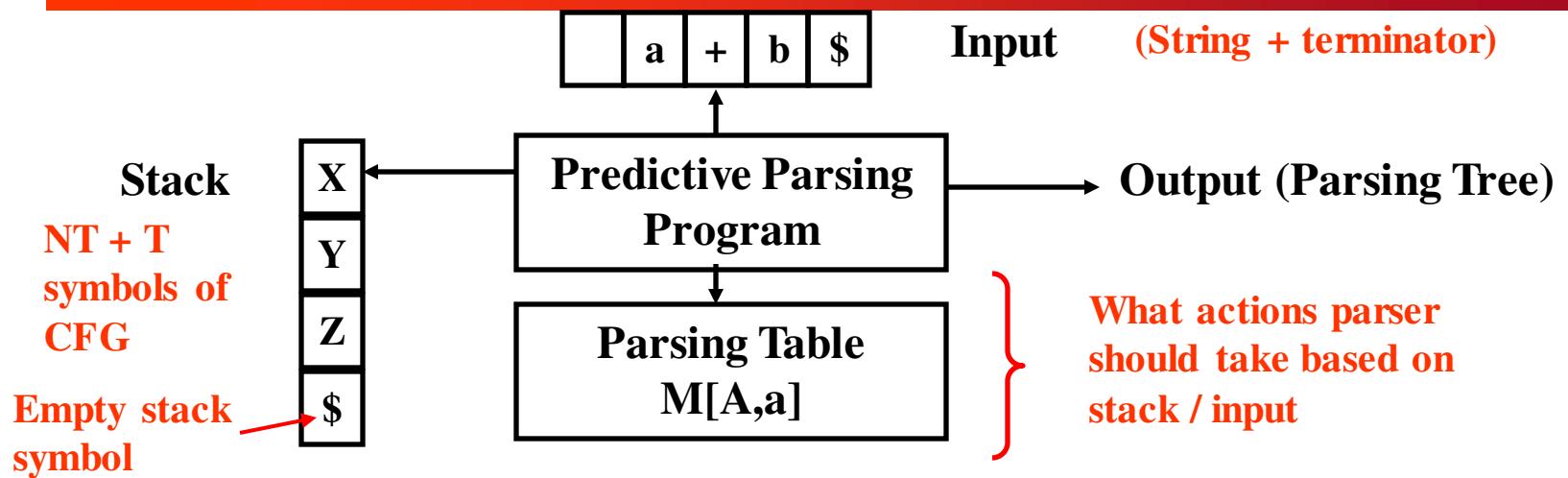
It is possible to build a *nonrecursive predictive parser* by maintaining a *stack* explicitly

Non-Recursive / Table Driven



- Input buffer contains the string to be parsed, followed by \$, as a terminator
- The stack contains a sequence of grammar symbols with \$ on the bottom
- Initially, the stack contains the *start symbol* of the grammar on top of \$
- The parsing table is a two-dimensional array $M[A, a]$, where A is a nonterminal, and a is a terminal or the symbol \$

Non-Recursive / Table Driven



General parser behavior: X : top of stack a : current input

1. When $X=a = \$$ halt, accept
2. When $X=a \neq \$$, POP X off stack, advance input, go to 1
3. When X is a non-terminal, examine $M[X, a]$
 - if it is an error → call recovery routine
 - if $M[X, a] = \{X \rightarrow UVW\}$, POP X, PUSH W,V,U
 - Add $X \rightarrow UVW$ to the output
 - DO NOT expend any input

Algorithm for Non-Recursive Parsing

Set *ip* to point to the first symbol of *w\$*;

repeat

 let *X* be the top stack symbol and *a* the symbol pointed to by *ip*;

if *X* is terminal or \$ **then**

if *X=a* **then**

 pop *X* from the stack and advance *ip*

else *error()*

else /* *X* is a non-terminal */

if *M[X,a] = X→Y₁Y₂...Y_k* **then begin**

 pop *X* from stack;

 push *Y_k, Y_{k-1}, ..., Y₁* onto stack, with *Y₁* on top

 output the production *X→Y₁Y₂...Y_k*

end

else *error()*

until *X=\$* /* stack is empty */

Input pointer

**May also execute other code
based on the production used**

Parsing Table

- A two dimensional array
 - $M[A, a]$ → gives a production
 - A : a nonterminal symbol
 - a : a terminal symbol
- What does it mean?
 - If top of the stack is A and the lookahead symbol is a then we apply the production $M[A, a]$

Example

$E \rightarrow TE'$
 $E' \rightarrow + TE' \mid \in$
 $T \rightarrow FT'$
 $T' \rightarrow * FT' \mid \in$
 $F \rightarrow (E) \mid id$

Table M

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow + TE'$			$E' \rightarrow \in$	$E' \rightarrow \in$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \in$	$T' \rightarrow * FT'$		$T' \rightarrow \in$	$T' \rightarrow \in$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Blank cells are error entries

Non-Blank cell indicates a production with which to expand the top nonterminal on the stack

Trace of Example

STACK	INPUT	OUTPUT
\$E	id + id * id\$	
\$E'T	id + id * id\$	E → TE'
\$E'T'F	id + id * id\$	T → FT'
\$E'T'id	id + id * id\$	F → <u>id</u>
\$E'T'	+ id * id\$	
\$E'	+ id * id\$	T' → ∈
\$E'T+	+ id * id\$	E' → +TE'
\$E'T	id * id\$	
\$E'T'F	id * id\$	T → FT'
\$E'T'id	id * id\$	F → <u>id</u>
\$E'T'	* id\$	
\$E'T'F*	* id\$	T' → *FT'
\$E'T'F	id\$	
\$E'T'id	id\$	F → <u>id</u>
\$E'T'	\$	
\$E'	\$	T' → ∈
\$	\$	E' → ∈

Expend Input

Leftmost Derivation for the Example

The leftmost derivation for the example is as follows:

$$\begin{aligned} E &\Rightarrow TE' \Rightarrow FT'E' \Rightarrow \mathbf{id\ T'E'} \Rightarrow \mathbf{id\ E'} \Rightarrow \mathbf{id + TE'} \Rightarrow \mathbf{id + FT'E'} \\ &\Rightarrow \mathbf{id + id\ T'E'} \Rightarrow \mathbf{id + id * FT'E'} \Rightarrow \mathbf{id + id * id\ T'E'} \\ &\Rightarrow \mathbf{id + id * id\ E'} \Rightarrow \mathbf{id + id * id} \end{aligned}$$

What's the Missing Puzzle Piece ?

Constructing the Parsing Table M !

1st : Calculate First & Follow for Grammar

**2nd: Apply Construction Algorithm for Parsing Table
(We'll see this shortly)**

FIRST: If α is any string of grammar symbols. Let

First(α) be the set of terminals that begin the strings derived from $^*\alpha$

NOTE: If $\alpha \Rightarrow \epsilon$, then ϵ is First(α)

That is, x ($\in T$) \in FIRST(α) iff $\alpha \stackrel{*}{\Rightarrow} x \gamma$, for some γ

What's the Missing Puzzle Piece ?

- **Follow:**

- Let A be a non-terminal. $\text{Follow}(A)$ is the set of terminals that can appear directly to the right of A in some sentential form
 - If $S \xrightarrow{*} \alpha A a \beta$, for some α and β , implies
 - $a \in \text{Follow}(A)$, where $a \in T$

Computing First(X) : All Grammar Symbols

1. If X is a terminal, $\text{First}(X) = \{X\}$
2. If $X \rightarrow \epsilon$ is a production rule, add ϵ to $\text{First}(X)$
3. If X is a non-terminal, and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production rule

Place $\text{First}(Y_1)$ in $\text{First}(X)$ and
if $Y_1 \xrightarrow{*} \epsilon$, Place $\text{First}(Y_2)$ in $\text{First}(X)$ and
if $Y_2 \xrightarrow{*} \epsilon$, Place $\text{First}(Y_3)$ in $\text{First}(X)$ and
...
if $Y_{k-1} \xrightarrow{*} \epsilon$, Place $\text{First}(Y_k)$ in $\text{First}(X)$

NOTE: As soon as $Y_i \xrightarrow{*} \epsilon$, Stop.

May repeat 1, 2, and 3, above for each Y_j

Computing First(X) : All Grammar Symbols - continued

Informally, suppose we want to compute

$$\text{First}(X_1 X_2 \dots X_n) = \text{First}(X_1) \text{ "+"}$$

$$\text{First}(X_2) \text{ if } \in \text{ is in } \text{First}(X_1) \text{ "+"}$$

$$\text{First}(X_3) \text{ if } \in \text{ is in } \text{First}(X_2) \text{ "+"}$$

...

$$\text{First}(X_n) \text{ if } \in \text{ is in } \text{First}(X_{n-1})$$

Note 1: As soon as $X_i \not\Rightarrow \in$, Stop

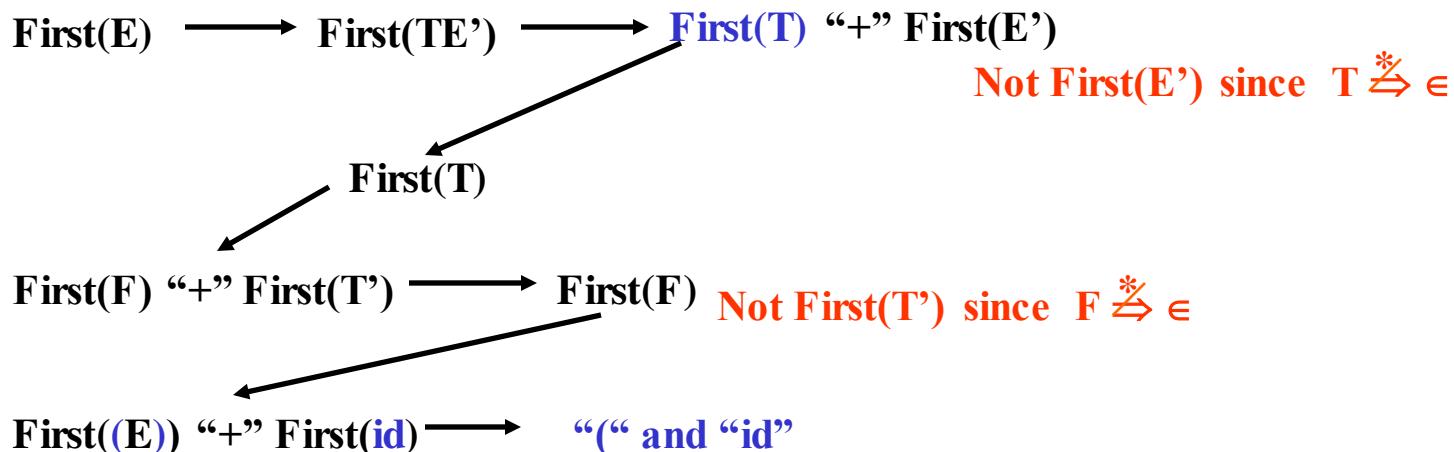
Note 2: Only add \in to $\text{First}(X_1 X_2 \dots X_n)$ if \in is in $\text{First}(X_i)$ for all i

Note 3: For $\text{First}(X_1)$, if $X_1 \rightarrow Z_1 Z_2 \dots Z_m$, then we need to compute $\text{First}(Z_1 Z_2 \dots Z_m)$!

Example

Computing First for:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow + \text{ } TE' \mid \in \\ T &\rightarrow FT' \\ T' &\rightarrow * \text{ } FT' \mid \in \\ F &\rightarrow (\text{ } E \text{ }) \mid \text{id} \end{aligned}$$



Overall: $\text{First}(E) = \{ (, \text{id}) \} = \text{First}(F) = \text{First}(T)$

$\text{First}(E') = \{ + , \in \} \quad \text{First}(T') = \{ * , \in \}$

Computing FOLLOW(A) : All Non-Terminals

- To construct FOLLOW(A) for a non-terminal symbol A apply the following rules until no more symbols can be added to FOLLOW(A)
 - Place \$ in FOLLOW(S)
 - \$ is the end-of-file symbol, S is the start symbol
 - If there is a production $A \rightarrow \alpha B \beta$, then
 - everything in FIRST(β) except ϵ is placed in FOLLOW(B)
 - If there is a production $A \rightarrow \alpha B$, then
 - everything in FOLLOW(A) is placed in FOLLOW(B)
 - If there is a production $A \rightarrow \alpha B \beta$, and ϵ is in FIRST(β) then
 - everything in FOLLOW(A) is placed in FOLLOW(B)

Example

-
- $E \rightarrow TE'$ $E' \rightarrow + TE' \mid \in \quad T \rightarrow FT'$
 - $T' \rightarrow * FT' \mid \in \quad F \rightarrow (E) \mid id$

○ Follow(E):

- contains \$
 - since E is the start symbol.

- $F \rightarrow (E):$

- First(")") is in Follow(E),

- $\text{Follow}(E) = \{ \), \$ \ }$

○ Follow(E'):

- $E \rightarrow TE':$

- $\text{Follow}(E)$ is in $\text{Follow}(E')$,

- $\text{Follow}(E') = \{ \), \$ \}$

Example

-
- $E \rightarrow TE'$ $E' \rightarrow + TE' \mid \in \quad T \rightarrow FT'$
 - $T' \rightarrow * FT' \mid \in \quad F \rightarrow (E) \mid id$

○ Follow(T):

- $E \rightarrow TE':$

- First(E') is in Follow(T), and
 - since $E' \xrightarrow{*} \in$, Follow(E) is in Follow(T).

- $E' \rightarrow +TE':$

- First(E') is in Follow(T), and
 - since $E' \xrightarrow{*} \in$, Follow(E') is in Follow(T).

- $\text{Follow}(T) = \{ +,), \$ \}.$

Example

-
- $T \rightarrow FT' \quad T' \rightarrow *FT' \mid \in \quad F \rightarrow (E) \mid id$
 - **Follow(T'):**
 - $T \rightarrow FT':$
 - Follow(T) is in Follow(T'),
 - $Follow(T') = \{+,), \$\}$
 - **Follow(F):**
 - $T \rightarrow FT':$
 - First(T') is in Follow(F), and
 - since $T' \xrightarrow{*} \in$, Follow(T) is in Follow(F).
 - $T' \rightarrow *FT':$
 - First(T') is in Follow(F), and
 - since $T' \xrightarrow{*} \in$, Follow(T') is in Follow(F).
 - $Follow(F) = \{+, *,), \$\}.$

Motivation Behind First & Follow

First: Is used to indicate the relationship between **non-terminals** (in the stack) and **input symbols** (in input stream)

Example: If $A \rightarrow \alpha$, and a is in $\text{First}(\alpha)$, then when $a = \text{input}$, and A is top of the stack, then replace A with α .

(a is one of first symbols of α , so when A is on the stack and a is input, POP A and PUSH α .

Follow: When $\alpha \rightarrow \in$ or $\alpha \xrightarrow{*} \in$, then what follows A dictates the next choice to be made.

Example: If $A \rightarrow \alpha$, and b is in $\text{Follow}(A)$, then when $\alpha \xrightarrow{*} \in$, and if b is an input character, then we expand A with α : POP A and PUSH α

Constructing Parsing Table

Algorithm:

1. Repeat Steps 2 & 3 for each rule $A \rightarrow \alpha$
2. Terminal a in $\text{First}(\alpha)$? Add $A \rightarrow \alpha$ to $M[A, a]$
- 3.1 ϵ in $\text{First}(\alpha)$? Add $A \rightarrow \alpha$ to $M[A, b]$ for all terminals b in $\text{Follow}(A)$
- 3.2 ϵ in $\text{First}(\alpha)$ and $\$$ in $\text{Follow}(A)$? Add $A \rightarrow \alpha$ to $M[A, \$]$
4. All undefined entries are errors

Constructing Parsing Table - Example

$E \rightarrow TE'$
 $E' \rightarrow +TE' \mid \in$
 $T \rightarrow FT'$
 $T' \rightarrow *FT' \mid \in$
 $F \rightarrow (E) \mid id$

$\text{First}(E, F, T) = \{ (, id \}$
 $\text{First}(E') = \{ +, \in \}$
 $\text{First}(T') = \{ *, \in \}$

$\text{Follow}(E, E') = \{), \$ \}$
 $\text{Follow}(F) = \{ *, +,), \$ \}$
 $\text{Follow}(T, T') = \{ +,), \$ \}$

Expression Example: $E \rightarrow TE' \Rightarrow \text{First}(TE') = \text{First}(T) = \{ (, id \}$

$$\left. \begin{array}{l} M[E, (] = E \rightarrow TE' \\ M[E, id] = E \rightarrow TE' \end{array} \right\} \text{by rule 2}$$

(by rule 2) $E' \rightarrow +TE' \Rightarrow \text{First}(+TE') = + \Rightarrow M[E', +] = E' \rightarrow +TE'$

(by rule 3) $E' \rightarrow \in : \in \text{ in } \text{First}(\in)$

$$M[E', ()] = E' \rightarrow \in \quad (3.1)$$

$$M[E', \$] = E' \rightarrow \in \quad (3.2)$$

(Due to $\text{Follow}(E')$)

2. Terminal a in $\text{First}(\alpha)$? Add $A \rightarrow \alpha$ to $M[A, a]$
- 3.1 \in in $\text{First}(\alpha)$? Add $A \rightarrow \alpha$ to $M[A, b]$ for all terminals b in $\text{Follow}(A)$
- 3.2 \in in $\text{First}(\alpha)$ and $\$$ in $\text{Follow}(A)$? Add $A \rightarrow \alpha$ to $M[A, \$]$

Constructing Parsing Table - Example

$E \rightarrow TE'$
 $E' \rightarrow + TE' \mid \in$
 $T \rightarrow FT'$
 $T' \rightarrow * FT' \mid \in$
 $F \rightarrow (E) \mid id$

$\text{First}(E, F, T) = \{ (, id \}$
 $\text{First}(E') = \{ +, \in \}$
 $\text{First}(T') = \{ *, \in \}$

$\text{Follow}(E, E') = \{), \$ \}$
 $\text{Follow}(F) = \{ *, +,), \$ \}$
 $\text{Follow}(T, T') = \{ +,), \$ \}$

Expression Example: $T \rightarrow FT' \Rightarrow \text{First}(FT') = \text{First}(F) = \{ (, id \}$

$$\left. \begin{array}{l} M[T, ()] = T \rightarrow FT' \\ M[T, id] = T \rightarrow FT' \end{array} \right\} \text{by rule 2}$$

(by rule 2) $T' \rightarrow *FT' \Rightarrow \text{First}(*FT') = *$ $\Rightarrow M[T', *] = T' \rightarrow *FT'$

(by rule 3) $T' \rightarrow \in \Rightarrow \in \text{ in } \text{First}(\in)$

$$M[T', +] = T' \rightarrow \in \quad (3.1)$$

$$M[T', ()) = T' \rightarrow \in \quad (3.1)$$

$$M[T', \$] = T' \rightarrow \in \quad (3.2)$$

(Due to Follow(T'))

2. Terminal a in $\text{First}(\alpha)$? Add $A \rightarrow \alpha$ to $M[A, a]$
- 3.1 \in in $\text{First}(\alpha)$? Add $A \rightarrow \alpha$ to $M[A, b]$ for all terminals b in $\text{Follow}(A)$
- 3.2 \in in $\text{First}(\alpha)$ and $\$$ in $\text{Follow}(A)$? Add $A \rightarrow \alpha$ to $M[A, \$]$

Constructing Parsing Table - Example

$E \rightarrow TE'$
 $E' \rightarrow + TE' \mid \in$
 $T \rightarrow FT'$
 $T' \rightarrow * FT' \mid \in$
 $F \rightarrow (E) \mid id$

$\text{First}(E, F, T) = \{ (, id \}$
 $\text{First}(E') = \{ +, \in \}$
 $\text{First}(T') = \{ *, \in \}$

$\text{Follow}(E, E') = \{), \$ \}$
 $\text{Follow}(F) = \{ *, +,), \$ \}$
 $\text{Follow}(T, T') = \{ +,), \$ \}$

$F \rightarrow (E) \Rightarrow \text{First}((E)) = \text{First}(()) = \{ (\}$

$M[F, (] = F \rightarrow (T) \quad \text{by rule 2}$

$F \rightarrow id \Rightarrow \text{First}(id) = \{ id \}$

$M[F, id] = F \rightarrow id \quad \text{by rule 2}$

2. Terminal a in $\text{First}(\alpha)$? Add $A \rightarrow \alpha$ to $M[A, a]$

3.1 \in in $\text{First}(\alpha)$? Add $A \rightarrow \alpha$ to $M[A, b]$ for all terminals b in $\text{Follow}(A)$

3.2 \in in $\text{First}(\alpha)$ and $\$$ in $\text{Follow}(A)$? Add $A \rightarrow \alpha$ to $M[A, \$]$

Example

$E \rightarrow TE'$
 $E' \rightarrow + TE' \mid \in$
 $T \rightarrow FT'$
 $T' \rightarrow * FT' \mid \in$
 $F \rightarrow (E) \mid id$

Table M

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow + TE'$			$E' \rightarrow \in$	$E' \rightarrow \in$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \in$	$T' \rightarrow * FT'$		$T' \rightarrow \in$	$T' \rightarrow \in$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Blank cells are error entries

Non-Blank cell indicates a production with which to expand the top nonterminal on the stack

Example

$E \rightarrow TE'$
 $E' \rightarrow + TE' \mid \in$
 $T \rightarrow FT'$
 $T' \rightarrow * FT' \mid \in$
 $F \rightarrow (E) \mid id$

Table M

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	TE'			TE'		
E'		+TE'			\in	\in
T	FT'			FT'		
T'		\in	*FT'		\in	\in
F	id			(E)		

Blank cells are error entries

Non-Blank cell indicates a production with which to expand the top nonterminal on the stack

Constructing Parsing Table – Example 2

$S \rightarrow i E t SS' a$	$\text{First}(S) = \{ i, a \}$	$\text{Follow}(S) = \{ e, \$ \}$
$S' \rightarrow eS \epsilon$	$\text{First}(S') = \{ e, \epsilon \}$	$\text{Follow}(S') = \{ e, \$ \}$
$E \rightarrow b$	$\text{First}(E) = \{ b \}$	$\text{Follow}(E) = \{ t \}$

$S \rightarrow i E t SS'$

$\text{First}(i E t SS') = \{ i \}$

$S \rightarrow a$

$\text{First}(a) = \{ a \}$

$E \rightarrow b$

$\text{First}(b) = \{ b \}$

$S' \rightarrow eS$

$\text{First}(eS) = \{ e \}$

$S' \rightarrow \epsilon$

$\text{First}(\epsilon) = \{ \epsilon \}$

$\text{Follow}(S') = \{ e, \$ \}$

Non-terminal	INPUT SYMBOL					
	a	b	e	i	t	\$
S	<u>$S \rightarrow a$</u>				<u>$S \rightarrow iEtSS'$</u>	
S'			<u>$S' \rightarrow \epsilon$</u> <u>$S' \rightarrow eS$</u>			<u>$S \rightarrow \epsilon$</u>
E		<u>$E \rightarrow b$</u>				

LL(1) grammars

L : Scan input from Left to Right

L : Construct a Leftmost Derivation

1 : Use “1” input symbol as lookahead in conjunction with stack to decide on the parsing action

- Two alternative definitions of LL(1) grammars:
 - A grammar G is LL(1) if there are no multiple entries in its LL(1) parse table
 - A grammar G is LL(1) if for each set of its productions
$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$
 - $\text{FIRST}(\alpha_1), \text{FIRST}(\alpha_2), \dots, \text{FIRST}(\alpha_n)$, are all pairwise disjoint
 - If $\alpha_i \Rightarrow^* \epsilon$, then $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \emptyset$ for all $1 \leq i \leq n, i \neq j$
 - At most one α_i can derive the empty string ϵ

Provable Facts about LL(1) grammars

- No **left-recursive grammar** is LL(1)
- No **ambiguous grammar** is LL(1)
- Not all context-free grammars are LL(1)
 - $S \rightarrow a S \mid a$ is not LL(1)
 - FIRST($a S$) = FIRST(a) = $\{a\}$
 - But, by **left-factoring** we can convert it to an LL(1) grammar
 - $S \rightarrow a S'$ $S' \rightarrow a S' \mid \epsilon$

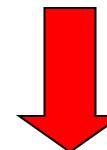
Disambiguating the LL(1) parsers

1	$S \rightarrow$	<u>if E then S</u>
2		<u>if E then S else S</u>
3		<u>other</u>
4	$E \rightarrow$	<u>exp</u>



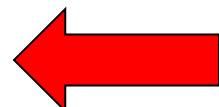
Not LL(1),
use left-factoring

1	$S \rightarrow$	<u>if E then S S'</u>
2		<u>other</u>
3	$S' \rightarrow$	<u>else S</u>
4		ϵ
5	$E \rightarrow$	<u>exp</u>



Compute the
FIRST and
FOLLOW
sets

Build the
LL(1)
Parse table



	OTHER	ELSE	EXP	IF	\$
S	$S \rightarrow \text{other}$			$S \rightarrow \text{if E then S S'}$	
S'		$S' \rightarrow \epsilon$			$S' \rightarrow \epsilon$
E		$S' \rightarrow \text{else S}$		$E \rightarrow \text{exp}$	

This grammar is not LL(1)
since there are multiple entries
(of course, the grammar was ambiguous)

Choose this production
and remove the other one
to resolve the ambiguity.
It gives the correct result.

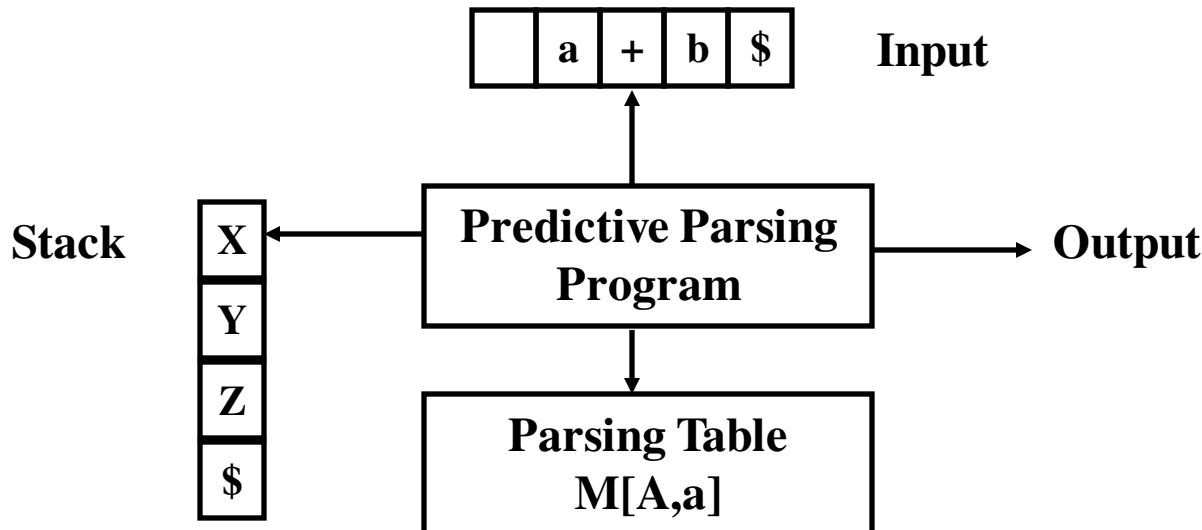
Symbol	FIRST	FOLLOW
S	{if, other}	{\$, else}
S'	{ ϵ , else}	{\$, else}
E	{exp}	{then}
if	{if}	
then	{then}	
else	{else}	
other	{other}	
exp	{exp}	

Basic Error Types

- **Lexical**
 - an error producing a wrong token (misspelling of an identifier, keyword or operator) e.g. WHILE – HWILE
- **Syntactic**
 - a program structure does not satisfy the CFG of the language (arithmetic expression with unbalanced parentheses, missing semicolon) ,
- **Semantic**
 - an error that needs additional context sensitive information, not included in CFG (operator applied to an incompatible operand, accessing an undeclared variable, problems with types, declarations, parameters)
- **Logical**
 - run-time errors, impossible to detect in programming stage (infinite recursion, accessing an array out of bounds, problems in pointer arithmetic, dividing by zero, memory allocation problems)

Syntax Error Recovery

When Do Errors Occur? Recall Predictive Parser Function:



1. If X is a terminal and it doesn't match input.
2. If $M[X, \text{Input}]$ is empty – No allowable actions

Panic Mode Recovery

- Panic mode recovery: discard input tokens until a token in a selected set of synchronizing tokens appear.
- until we reach a point where parsing can continue, otherwise terminate
- The synchronizing tokens should be chosen so the parser recovers quickly from the errors.
- One approach is to place all symbols in FOLLOW(A) into the synchronizing set for nonterminal A

Revised Parsing Table / Example

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	<u>synch</u>	<u>synch</u>
E'		$E' \rightarrow + TE'$			$E' \rightarrow \in$	$E' \rightarrow \in$
T	$T \rightarrow FT'$	<u>synch</u>		$T \rightarrow FT'$	<u>synch</u>	<u>synch</u>
T'		$T' \rightarrow \in$	$T' \rightarrow * FT'$		$T' \rightarrow \in$	$T' \rightarrow \in$
F	$F \rightarrow id$	<u>synch</u>	<u>synch</u>	$F \rightarrow (E)$	synch	synch

From Follow sets. Pop stack entry (Nonterminal)

Skip input symbol (Terminal)

$E \rightarrow TE'$
$E' \rightarrow + TE' \mid \in$
$T \rightarrow FT'$
$T' \rightarrow * FT' \mid \in$
$F \rightarrow (E) \mid id$

$First(E, F, T) = \{ (, id \}$
$First(E') = \{ +, \in \}$
$First(T') = \{ *, \in \}$

$Follow(E, E') = \{), \$ \}$
$Follow(F) = \{ *, +,), \$ \}$
$Follow(T, T') = \{ +,), \$ \}$

Panic Mode Recovery

- Let X be the top stack symbol (Terminal or nonterminal) and let a be the input symbol being read and b is the next input symbol
 - If $M[X, a]$ is blank and b is in $\text{First}(X)$:
 - skip input symbol a
 - If $M[X, a]$ is synch (a is in $\text{Follow}(X)$):
 - pop X from the stack
 - If X is a terminal (token) and does not match the input symbol a :
 - pop X (token) from the stack

Revised Parsing Table / Example(2)

STACK	INPUT	Remark
\$E) id * + id\$	error, skip)
\$E	id * + id\$	id is in First(E)
\$E'T	id * + id\$	
\$E'T'F	id * + id\$	
\$E'T'id	id * + id\$	
\$E'T'	* + id\$	
\$E'T'F*	* + id\$	
\$E'T'F	+ id\$	error, M[F,+] = synch
\$E'T'	+ id\$	F has been popped
\$E'	+ id\$	
\$E'T+	+ id\$	
\$E'T	id\$	
\$E'T'F	id\$	
\$E'T'id	id\$	
\$E'T'	\$	
\$E'	\$	
\$	\$	

How Would You Implement Table Derived Parser

- Key Issue – How is parsing table implemented ?

One approach: Assign unique IDS

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	synch	synch	$F \rightarrow (E)$	synch	synch

All rules have unique IDs

IDs for synch actions

Also for blanks which handle errors

Revised Parsing Table:

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	1	18	19	1	9	10
E'	20	2	21	22	3	3
T	4	11	23	4	12	13
T'	24	6	5	25	6	6
F	8	14	15	7	16	17

1 E → TE'

2 E' → +TE'

3 E' → ε

4 T → FT'

5 T' → *FT'

6 T' → ε

7 F → (E)

8 F → id

9 – 17 :

Sync

Actions

18 – 25 :

Error

Handlers

How is Parser Constructed ?

One large CASE statement:

```
state = M[ top(s), current_token ]
switch (state)
{
    case 1: proc_E_TE'();
              break ;
    ...
    case 8: proc_F_id();
              break ;
    case 9: proc_sync_9();
              break ;
    ...
    case 17: proc_sync_17();
              break ;
    case 18:
    ...
    case 25:
}
```



The code shows a large switch statement for handling different states. Cases 1 through 17 handle standard tokens or expressions. Cases 18 through 25 are grouped together and labeled as "Procs to handle errors".

Final Comments – Top-Down Parsing

- So far,
 - We've examined grammars and language theory and its relationship to parsing
 - Key concepts: Rewriting grammar into an acceptable form
 - Examined Top-Down parsing:
 - Transition diagrams & recursion
 - Elegant : Table driven
- We've identified its shortcomings:
 - Not all grammars can be made LL(1)
- Bottom-Up Parsing - Future

Compilers Construction

Chapter 5

Bottom-Up Parsing

Dr. Mohammed Salem Atoum

M.Atoum@inu.edu.jo

Bottom-up Parsing

- Given a grammar G and a string σ that one believes is a member of $L(G)$ there are two basic ways that one can attempt to find a derivation for σ .
- Top-down parsing
 - Begin with the start symbol S of G and repeatedly substitute the left hand side of a production for a non-terminal until the start symbol has been rewritten to match σ .
- Bottom-up parsing
 - Begin with σ and repeatedly "simplify" the string by replacing a sub-string that matches the right hand side of a production by the non-terminal on its left hand side until σ has been simplified (reduced) to S .

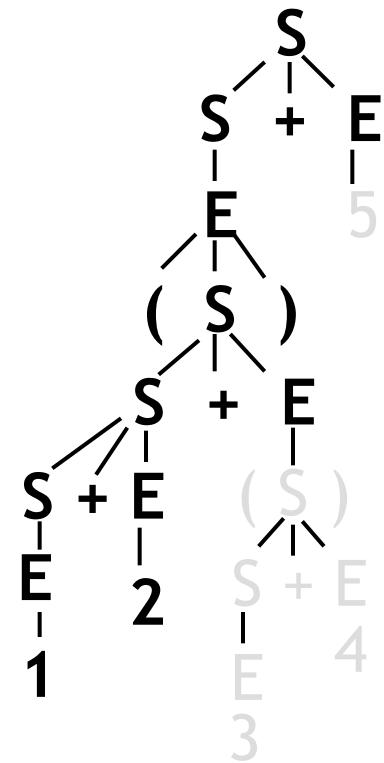
Top-down parsing

$S \rightarrow S+E \rightarrow E+E \rightarrow (S)+E \rightarrow (S+E)+E$
 $\rightarrow (S+E+E)+E \rightarrow (E+E+E)+E$
 $\rightarrow (1+E+E)+E \rightarrow (1+2+E)+E \dots$

Left-most derivation

$S \rightarrow S+E \mid E$
$E \rightarrow \text{number} \mid (S)$

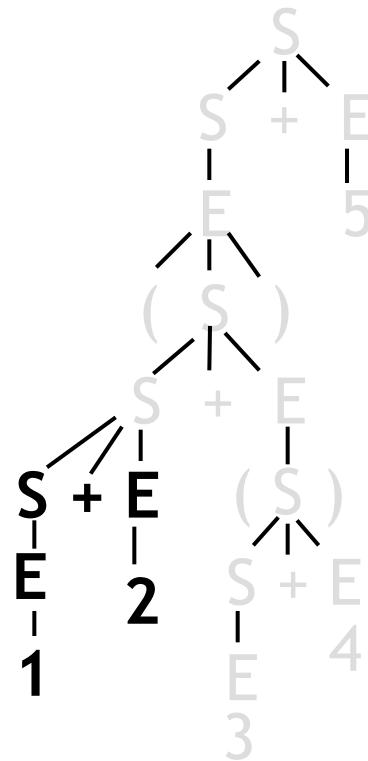
$(1+2+(3+4))+5$



Bottom-up parsing

- Right-most derivation-- backward
 - Start with the tokens
 - End with the start symbol
 - $(1+2+(3+4))+5 \leftarrow (\textcolor{blue}{E}+2+(3+4))+5 \leftarrow (\textcolor{blue}{S}+2+(3+4))+5 \leftarrow (S+\textcolor{blue}{E}+(3+4))+5 \dots$

$S \rightarrow S + E \mid E$
 $E \rightarrow \text{number} \mid (S)$



Bottom-up parsing

- **Idea:**
 - Read the input left to right
 - Whenever we've matched the **right hand side** of a production, **reduce** it to the appropriate **non-terminal** and add that non-terminal to the parse tree

Finding Reductions

- Consider the simple grammar and the input string
abbcde

- 1 $S \rightarrow aABe$
- 2 $A \rightarrow Abc$
- 3 $\quad \quad \rightarrow b$
- 4 $B \rightarrow d$

Sentential Form	Next Production	Reduction Position
abbcde	3	2
aAbcde	2	2
aAde	4	3
aABe	1	1
S	--	--

- $S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$
- The trick is scanning the input and finding the next reduction

Bottom-up Parsing

- Bottom-up parsing yields **right-most derivations** (*in reverse order*) by working from the input sentence back toward the start symbol S
 - $S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w \text{ sentence}$
- To derive the sentential form γ_{i-1} from γ_i , it matches $\gamma_i = \alpha\beta w$ against some β , then replaces β with its corresponding lhs, A (assuming $A \rightarrow \beta$)
 - $\gamma_{i-1} = \alpha A w$ and $|A| \leq |\beta|$
- the parser will first discover γ_{n-1} from γ_n , then γ_{n-2} from γ_{n-1} , etc.

How Do We Automate This?

- **Key:** given what we've already seen and the next input symbol, decide what to do
- Choices:
 - Perform a reduction
 - Look ahead further
- This is known as a *shift-reduce* parser
- Can reduce $A \rightarrow \beta$ if both of these hold:
 - $A \rightarrow \beta$ is a valid production
 - $A \rightarrow \beta$ is a step in this rightmost derivation
 - Since each replacement of β with A shrinks the input sentence, we call it a *reduction*
- A **sentential form** in a **rightmost derivation** is called a **right-sentential form** (similarly for leftmost and left-sentential)

Handles

- The handle of right sentential form γ is a production $A \rightarrow \beta$ and a position in γ , where β may be found and replaced by A to produce the *previous right sentential* form in the rightmost derivation of γ
-
- So, if $S \Rightarrow_{(rm)}^* \alpha A w \Rightarrow_{(rm)} \alpha \beta w$,
 - then $A \rightarrow \beta$ in the position following α is the handle of $\gamma = \alpha \beta w$
- The string w to the right of the handle contains **only** terminal symbols
- If G is **unambiguous**, then every right-sentential form has a **unique** handle

Example

- Consider the following grammar:
 - $E \rightarrow E + E$ $E \rightarrow E * E$
 - $E \rightarrow (E)$ $E \rightarrow id$
- And the rightmost derivation:
 - $E \xrightarrow{(rm)} \underline{E + E} \xrightarrow{(rm)} E + \underline{E * E} \xrightarrow{(rm)} E + E * \underline{id_3}$
 $\xrightarrow{(rm)} E + \underline{id_2} * \underline{id_3} \xrightarrow{(rm)} \underline{id_1} + \underline{id_2} * \underline{id_3}$
- The **handle** for each right-sentential form is **underlined**
- Because the grammar is ambiguous, there is another rightmost derivation of the same string
 - $E \xrightarrow{(rm)} \underline{E * E} \xrightarrow{(rm)} E * \underline{id_3} \xrightarrow{(rm)} \underline{E + E} * \underline{id_3}$
 $\xrightarrow{(rm)} E + \underline{id_2} * \underline{id_3} \xrightarrow{(rm)} \underline{id_1} + \underline{id_2} * \underline{id_3}$

Handle - Example

- Consider the grammar:

$$\begin{aligned}S &\rightarrow aB \mid bA \mid bc \\A &\rightarrow b \\B &\rightarrow b \mid c\end{aligned}$$

- In general, we can't say whether a 'b' or 'c' that appears in the input is a **handle** or not.
- After reading a 'b',
 - we know that if the following character is a 'b' then the 2nd 'b' is the **handle**,
 - but that if it is a 'c' then the pair 'bc' forms the **handle**.

Handle - Example

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

○ In the derivation

- $S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$
 - $abbcde$: is a right sentential form whose handle is $A \rightarrow b$ at position 2
 - $abbcde$: the second b at position 3 is not a handle even it is a rhs of production $A \rightarrow b$, since $abAcde$ is not a right sentential form
 - $aAbcde$ is a right sentential form whose handle is $A \rightarrow Abc$ at position 2

Handle-pruning

- The process of discovering a handle & reducing it to the appropriate left-hand side is called *handle pruning*
- Handle pruning forms the basis for a bottom-up parsing method
- To construct a rightmost derivation
 - $S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow w$
- Apply the following simple algorithm

for i ← n to 1 by -1

Find the handle $\langle A_i \rightarrow \beta_i, k_i \rangle$ in γ_i

Replace β_i with A_i to generate γ_{i-1}

Handle-pruning Example

- Consider the following grammar:
 - $E \rightarrow E + E \mid E * E \mid (E) \mid id$
- The sequence of reductions which reduces the input string $id + id * id$ to the start symbol:

Right-Sentential Form	Handle	Reducing Production
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

Shift-Reduce Parsing

- Two problems of parse by handle pruning:
 - to locate the handle to be reduced
 - to determine what production to choose in case there is more than one production with the same handle on the right hand side
 - Basic data structures for bottom-up parsing:
 - stack (to hold grammar symbols)
 - input buffer (to hold the input string w)
 - Initially: Stack: \$ Input: $w\$$
 - Finally: Stack: \$\$ Input: \$
 - handle can appear only on the top of the stack

Shift-reduce Parsing

- A shift-reduce parser has just four actions
 - Shift: next token is shifted onto the stack
 - Reduce: right end of handle is at top of stack
 - Locate left end of handle within the stack
 - Pop handle off stack & push appropriate lhs
 - Accept: stop parsing & report success
 - Error: call an error reporting/recovery routine
- **Accept** & **Error** are simple
- **Shift** is just a **push** and a call to the scanner
- **Reduce** takes $|rhs|$ pops & 1 push

Shift-reduce parsing

- **Shift** -- push head of input onto stack

stack	input	
(1+2+ (3+4) +5	<i>shift</i>
(1	+2+ (3+4) +5	

- **Reduce** -- replace symbols γ in top of stack with non-terminal symbol X , corresponding to production $X \rightarrow \gamma$ (pop γ , push X)

stack	input	
<u>(S+E</u>	+ (3+4) +5	<i>reduce S → S+E</i>
<u>(S</u>	+ (3+4) +5	

Shift-reduce parsing

```
push $  
S = Start Symbol  
token ← next_token( )  
repeat until (top of stack = S and token = $)  
    if the top of the stack is a handle A → β  
        then /* reduce β to A */  
            pop |β| symbols off the stack  
            push A onto the stack  
    else if (token ≠ $)  
        then /* shift */  
            push token  
            token ← next_token( )
```

Example $\underline{x} - \underline{2} * \underline{y}$

		Stack	Input	Handle	Action
1	<i>Goal</i>	\$	<u>id</u> - <u>num</u> * <u>id</u>	<i>none</i>	shift
2	<i>Expr</i>	\$ <u>id</u>	- <u>num</u> * <u>id</u>	9,1	red. 9
3		\$ <i>Factor</i>	- <u>num</u> * <u>id</u>	7,1	red. 7
4		\$ <i>Term</i>	- <u>num</u> * <u>id</u>	4,1	red. 4
5		\$ <i>Expr</i>	- <u>num</u> * <u>id</u>	<i>none</i>	shift
6		\$ <i>Expr</i> -	<u>num</u> * <u>id</u>	<i>none</i>	shift
7		\$ <i>Expr</i> - <u>num</u>	* <u>id</u>	8,3	red. 8
8		\$ <i>Expr</i> - <u>Factor</u>	* <u>id</u>	7,3	red. 7
9	<i>Term</i>	\$ <i>Expr</i> - <u>Term</u>	* <u>id</u>	<i>none</i>	shift
		\$ <i>Expr</i> - <u>Term</u> *	<u>id</u>	<i>none</i>	shift
		\$ <i>Expr</i> - <u>Term</u> * <u>id</u>	9,5	red. 9	
	<i>Factor</i>	\$ <i>Expr</i> - <u>Term</u> * <u>Factor</u>	5,5	red. 5	
		\$ <i>Expr</i> - <u>Term</u>	3,3	red. 3	
		\$ <i>Expr</i>	1,1	red. 1	
		\$ <i>Goal</i>	<i>none</i>	accept	

1. Shift until top-of-stack is the right end of a handle
2. Find the left end of the handle & reduce

5 shifts +
9 reduces +
1 accept

Example, Corresponding Parse Tree

Stack	Input	Action
\$		
\$ <u>id</u>	<u>id</u> - num * id	shift
\$ Factor	- num * id	red. 9
\$ Term	- num * id	red. 7
\$ Expr	- num * id	red. 4
\$ Expr -	- num * id	shift
\$ Expr - num	num * id	shift
\$ Expr - Factor	* id	red. 8
\$ Expr - Term	* id	red. 7
\$ Expr - Term *	* id	shift
\$ Expr - Term * id	id	shift
\$ Expr - Term * Factor		red. 9
\$ Expr - Term		red. 5
\$ Expr		red. 3
\$ Goal		red. 1
		accept

```

graph TD
    Goal --> Expr1[Expr]
    Goal --> Expr2[Expr]
    Expr1 --> Expr3[Expr]
    Expr1 --> Term1[Term]
    Expr2 --> Term2[Term]
    Expr2 --> Fact1[Fact.]
    Term1 --> Fact2[Fact.]
    Term2 --> Fact3[Fact.]
    Fact1 --- Minus["-"]
    Fact3 --- Star["*"]
    Fact2 --- IDx["<id,x>"]
    Fact3 --- Num2["<num,2>"]
    Fact4["<id,y>"] --- Fact5[Fact.]
  
```

Parsing Conflicts

- **shift-reduce** conflict
 - $\text{stmt} \rightarrow \text{if expr then stmt}$
 | $\text{if expr then stmt else stmt}$ | other
 - Handle is **if expr then stmt** and input symbol is **else**
 - Do we Shift? Do we Reduce?
 - We can not tell
- **reduce-reduce** conflict
 - $A \rightarrow C$
 - $B \rightarrow C$
 - If the top of stack is **C** which reduction should be performed?

reduce-reduce conflict

- Let us have a grammar with rules:

stmt	→ id(parameter_list)	(1)
stmt	→ expr := expr	
parameter_list	→ parameter_list, parameter	
parameter_list	→ parameter	
parameter	→ id	(5)
expr	→ id (expr_list)	
expr	→ id	(7)
expr_list	→ expr_list, expr	
expr_list	→ expr	

reduce-reduce conflict

- input:

A(I, J)

- after lexical analysis:

id(id, id)

- parsing: STACK

INPUT

...id(id

,id)...

5

7

if A is a procedure

if A is an array

- Solution:

- additional information from declaration part (symbol table) is required
- redesign the grammar, e.g. change **id** to **procid** in (1). Then we obtain unique grammar forms, e.g.

➤ stackprocid(id

,id).... input

LR Parsing

- LR grammars -- more power than LL
 - can handle left-recursive grammars, virtually all programming languages
- LR(1) parsers are table-driven, shift-reduce parsers
 - L: for left-to-right scanning of the input
 - R: for constructing a rightmost derivation in reverse
 - 1: symbol lookahead

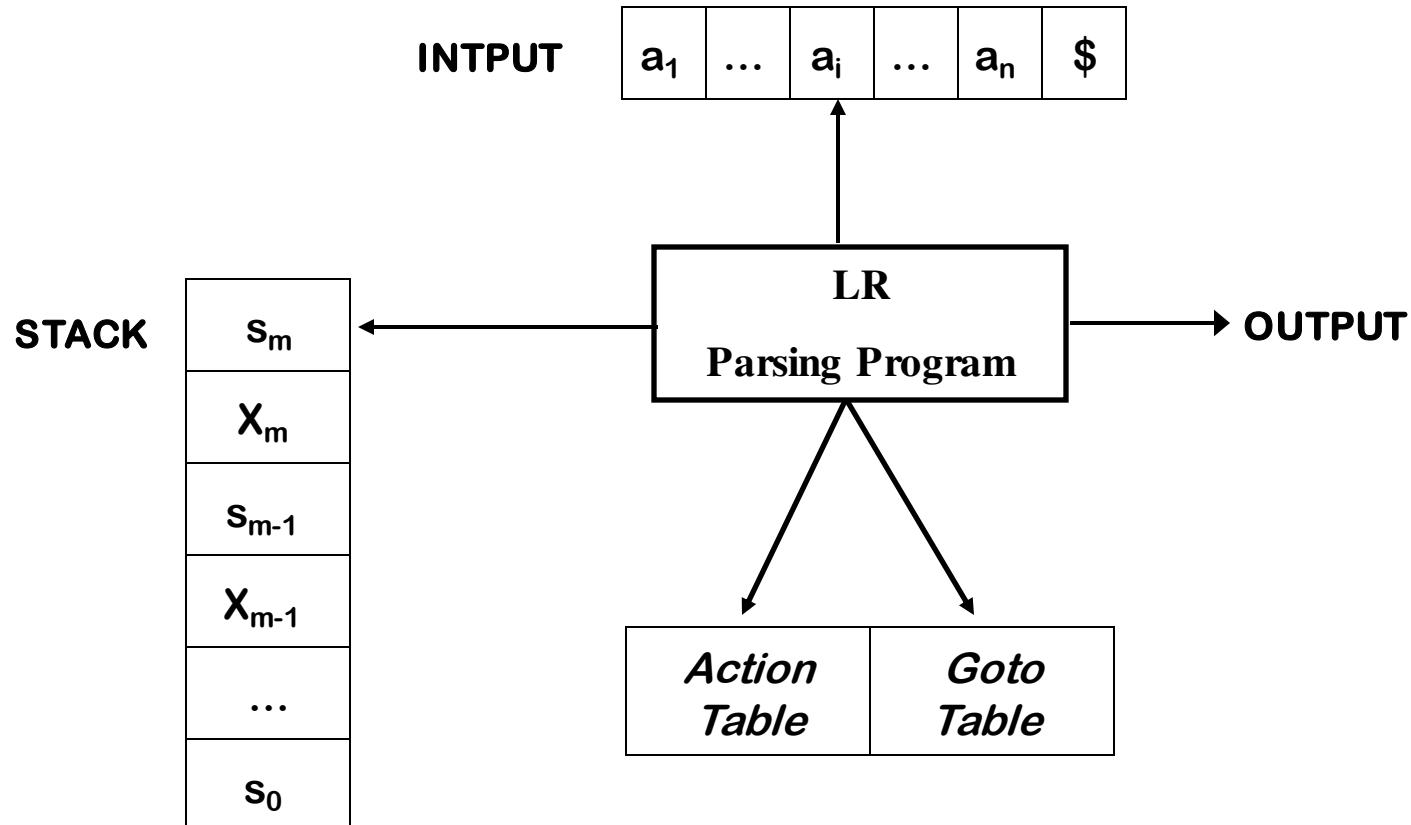
LR Parsers

- Simple LR (SLR)
 - Easy
 - Least powerful
- Canonical LR:
 - Expensive
 - Most powerful
- Lookahead LR (LALR)
 - Intermediate in power and cost

LR Parser

- It consists of:
 - Input stream
 - Output stream
 - Stack
 - Driver program (same for all LR parsers)
 - Reads characters from the input buffer one at a time
 - Parsing tables (changes from one parser to another)
 - Action Table
 - Goto Table

LR Parser



LR Parser

- The program uses the stack to store a string of the form
 - $s_0X_1s_1X_2s_2 \dots X_ms_m$
 - s_m is on top
 - Each X_i is a grammar symbol
 - Each s_i is a symbol called a state
 - Each state symbol summarized the information contained in the stack below it
- The combination of the state symbol on the top of the stack and the current input symbol are used to index the parsing table and determine the shift-reduce parsing decision

Action Table

○ Action Table

State	a_1	...	a_i	a_{i+1}	...	a_n	\$
s_0							
s_1							
...							
s_m				v			
...							
s_k							

Action Table

- $\text{action}[s_m, a_i] = v$ is the parsing action table entry for state s_m and input a_i
- v can have one of four values
 - shift s : shift the input symbol a_i and state s into the stack (i.e., shift and move to state s)
 - reduce j : reduce using grammar production j
 - The production number tells us how many <symbol, state> pairs to pop off the stack
 - accept
 - blank : no transition – syntax error

Goto Table

○ Goto Table

- $goto(s_m, A_i) = s$
- It takes
 - a state s_m and
 - a nonterminal A_i
- as arguments and produces
 - a state s

State	A_1	...	A_i	...	A_l
s_0					
s_1					
...					
s_m			s		
...					
s_k					

LR Parsing Algorithm

- A *configuration* of an LR parser is a **pair** whose
 - first component is the **stack** contents and whose
 - second component is the **unexpanded input**
 - $(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$
 - This configuration represent the **right-sentential form**
 - $X_1 X_2 \dots X_m a_i a_{i+1} \dots a_n$

LR Parser behaves as follows:

- It determines:
 - s_m : the state currently on top of the stack
 - a_i : the current input symbol
- It consults the *action* table entry $\text{action}[s_m, a_i]$
- If $\text{action}[s_m, a_i] = \text{shift } s$,
 - The parser execute a *shift* move, entering the configuration
 - $(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m a_i s, a_{i+1} \dots a_n \$)$
 - push a_i and s onto the stack
 - a_{i+1} becomes the current input symbol

LR Parser behaves as follows:

- If $\text{action}[s_m, a_i] = \text{reduce } A \rightarrow \beta$,
 - The parser execute a *reduce* move, entering the configuration
 - $(s_0 X_1 s_1 X_2 s_2 \dots X_{m-r} s_{m-r} A s, a_i a_{i+1} \dots a_n \$)$
 - pop $2r$ symbols off the stack
 - r state symbols + r grammar symbols
 - r is the length of β
 - $\beta = X_{m-r+1} \dots X_m$
 - s_{m-r} becomes top of the stack
 - push onto the stack both
 - A : the lhs of the production
 - s : where $s = \text{goto}[s_{m-r}, A]$

LR Parser behaves as follows:

- If $\text{action}[s_m, a_i] = \text{accept}$,
 - parsing is completed
- If $\text{action}[s_m, a_i] = \text{error}$,
 - call an error recovery routine
- Initially:
 - the stack has s_0 on it
 - $w\$$ is in the input buffer, where w is the input string
 - Output is right-most derivation of w or an error

LR Shift-Reduce Parsers

```
push $  
push  $s_0$   
token  $\leftarrow$  next_token()  
repeat forever  
    s  $\leftarrow$  top of stack  
    if ACTION[s,token] = reduce  $\alpha \rightarrow \beta$   
        then  
            pop  $2 * |\beta|$  symbols  
            s  $\leftarrow$  top of stack  
            push  $\alpha$   
            push GOTO[s, $\alpha$ ]  
    else if ACTION[s,token] = shift  $s_i$   
        then  
            push token ; push  $s_i$   
            token  $\leftarrow$  next_token()  
    else if ACTION[s,token] = accept  
        and token = $  
        then break;  
    else report a syntax error  
report success
```

1	E	$\rightarrow E + T$
2	E	$\rightarrow T$
3	T	$\rightarrow T * F$
4	T	$\rightarrow F$
5	F	$\rightarrow (E)$
6	F	$\rightarrow \text{id}$

Example Parses

<i>si</i>	shift and push state i
<i>rj</i>	reduce a production # j
<i>acc</i>	accept
<i>blank</i>	error

State	id	action					goto		
		+	*	()	\$	E	T	F
0	s5				s4		1	2	3
1		s6				acc			
2		r2	s7			r2	r2		
3		r4	r4			r4	r4		
4	s5				s4		8	2	3
5		r6	r6			r6	r6		
6	s5				s4			9	3
7	s5				s4				10
8		s6				s11			
9		r1	s7			r1	r1		
10		r3	r3			r3	r3		
11		r5	r5			r5	r5		

1	E	$\rightarrow E + T$
2	E	$\rightarrow T$
3	T	$\rightarrow T * F$
4	T	$\rightarrow F$
5	F	$\rightarrow (E)$
6	F	$\rightarrow \text{id}$

Example Parses

Stack	Input	Action
(1) 0	id * id + id \$	shift 5
(2) 0 id 5	* id + id \$	reduce 6
(3) 0 F 3	* id + id \$	reduce 4
(4) 0 T 2	* id + id \$	shift 7
(5) 0 T 2 * 7	id + id \$	shift 5
(6) 0 T 2 * 7 id 5	+ id \$	reduce 6
(7) 0 T 2 * 7 F 10	+ id \$	reduce 3
(8) 0 T 2	+ id \$	reduce 2
(9) 0 E 1	+ id \$	shift 6
(10) 0 E 1 + 6	id \$	shift 5
(11) 0 E 1 + 6 id 5	\$	reduce 6
(12) 0 E 1 + 6 F 3	\$	reduce 4
(13) 0 E 1 + 6 T 9	\$	reduce 1
(14) 0 E 1	\$	accept

LR Grammars

- We have discussed how a shift-reduce parser works, now it is time to learn how to build one
- The essential task for such a parser is to recognize the "**handle**" of a sentential form
- In particular, a shift-reduce parser **should never** read beyond the **end** of the **handle** of a sentential form without performing a reduction.
- $S \xrightarrow{\text{rm}}^* \alpha A w \xrightarrow{\text{rm}}^* \alpha \beta w$, then $A \rightarrow \beta$ in the position α is a **handle** of $\alpha \beta w$

Constructing SLR Parsing Tables

- SLR:
 - Simple Left-to-right scanning, Right-most derivation
 - Weakest in terms of the number of grammars for which it succeeds
 - Easiest to implement

LR(0) Items

- An (LR(0)) *item* is a **production** with a “•” somewhere in the **RHS**
- The production $A \rightarrow XYZ$ yields the 4 items:
 - $A \rightarrow \bullet XYZ$ $A \rightarrow X\bullet YZ$
 - $A \rightarrow XY\bullet Z$ $A \rightarrow XYZ\bullet$
- $A \rightarrow \epsilon$ has only one item $A \rightarrow \bullet$
- $A \rightarrow \bullet XYZ$:
 - we hope to see a string derivable from **XYZ** next on the input

LR(0) Items

- $A \rightarrow X \bullet YZ$:
 - we have just seen on the input a string derivable from X and we hope to see a string derivable from YZ next on the input
- Stuff before “ \bullet ” already on stack
- Stuff after “ \bullet ”: what we might see next
- A *state* is a set of *items*
- **Item sets**: all items that can potentially be seen at some given *state* in the parse
 - These items will always have the same grammar symbol preceding the \bullet

Augmented grammar

- If G is a grammar with start symbol S , then G' , the **augmented grammar** for G , is G with a new start symbol S' and production $S' \rightarrow S$

- The purpose of this new starting production is to indicate to the parser when it should stop parsing
 - That is, acceptance occurs when and only when the parser is about to reduce $S' \rightarrow S$

Computing Closures

- For a set of items I , $\text{Closure}(I)$ adds all the items implied by items already in I
 - Any item in I is in $\text{Closure}(I)$
 - If item $[A \rightarrow \alpha \bullet B\beta]$ is in $\text{Closure}(I)$ and $B \rightarrow \gamma$ is a production, then add $B \rightarrow \bullet\gamma$ to I
- The algorithm

```
Closure(I)
J = I;
while ( J is still changing )
    for each item [A → α • Bβ] ∈ J
        for each production B → γ ∈ P
            add [B → •γ] to J
Return J;
```

Closure example

- Consider the augmented grammar (add a new start symbol)

$E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

Closure of $\{ [E' \rightarrow \bullet E] \} = \{$

$E' \rightarrow \bullet E$
 $E \rightarrow \bullet E + T$
 $E \rightarrow \bullet T$
 $T \rightarrow \bullet T * F$
 $T \rightarrow \bullet F$
 $F \rightarrow \bullet (E)$
 $F \rightarrow \bullet id \}$

The Goto Function

- $\text{Goto}(I, X)$ for a set of items I and a grammar symbol X computes the state that the parser would reach if it recognized an X while in a state corresponding to I
- Given a set of items I for a grammar G , we define
 - $\text{goto}(I, X) = \{\text{closure}(A \rightarrow \alpha X \bullet \beta) \mid [A \rightarrow \alpha \bullet X \beta] \in I\}$

The Goto Function

○ Example

```
E' → E
E → E + T | T
T → T * F | F
F → (E) | id
```

```
Goto{ ([E' → E•] , [E → E•+T]) , + } =
closure(E → E+•T) =
{
    E → E + • T
    T → • T * F
    T → • F
    F → • ( E )
    F → • id
}
```

The Goto Function

- We computed $\text{goto}(I, +)$ by examining I for items with $+$ immediately to the right of the dot
 - $E' \xrightarrow{} E\bullet$ is not such an item, but
 - $E \xrightarrow{} E\bullet+T$ is
 - We moved the dot over the $+$ to get
 - $[E \xrightarrow{} E+\bullet T]$
 - and then took the closure of $[E \xrightarrow{} E+\bullet T]$

E'	$\xrightarrow{} E$
E	$\xrightarrow{} E + T \mid T$
T	$\xrightarrow{} T * F \mid F$
F	$\xrightarrow{} (E) \mid \text{id}$

$\text{Goto}\{ ([E' \xrightarrow{} E\bullet], [E \xrightarrow{} E\bullet+T]) , + \} =$

$$\begin{aligned} E &\xrightarrow{} E + \bullet T \\ T &\xrightarrow{} \bullet T * F \\ T &\xrightarrow{} \bullet F \\ F &\xrightarrow{} \bullet (E) \\ F &\xrightarrow{} \bullet \text{id} \end{aligned}$$

Sets-of-Items Construction

Items(G') (G' is the augmented grammar)
begin

$C = \{\text{Closure}(\{[S' \rightarrow \bullet S]\})\}$

while (C is still changing)

for each set of items I in C and each
grammar symbol X such that $\text{goto}(I, X)$ is
not empty and not already in C

add $\text{goto}(I, X)$ to C

end

Sets of LR(0) Items: Example

$I_0:$

$$\begin{aligned} E' &\rightarrow \bullet E \\ E &\rightarrow \bullet E + T \\ E &\rightarrow \bullet T \\ T &\rightarrow \bullet T^* F \\ T &\rightarrow \bullet F \\ F &\rightarrow \bullet (E) \\ F &\rightarrow \bullet id \end{aligned}$$

$I_1:$

$$\begin{aligned} E' &\rightarrow E \bullet \\ E &\rightarrow E \bullet + T \end{aligned}$$

$I_2:$

$$\begin{aligned} E &\rightarrow T \bullet \\ T &\rightarrow T \bullet * F \end{aligned}$$

$I_3:$

$$T \rightarrow F \bullet$$

$I_4:$

$$\begin{aligned} F &\rightarrow (\bullet E) \\ E &\rightarrow \bullet E + T \\ E &\rightarrow \bullet T \\ T &\rightarrow \bullet T^* F \\ T &\rightarrow \bullet F \\ F &\rightarrow \bullet (E) \\ F &\rightarrow \bullet id \end{aligned}$$

$I_5:$

$$F \rightarrow id \bullet$$

$I_6:$

$$\begin{aligned} E &\rightarrow E + \bullet T \\ T &\rightarrow \bullet T^* F \\ T &\rightarrow \bullet F \\ F &\rightarrow \bullet (E) \\ F &\rightarrow \bullet id \end{aligned}$$

$I_7:$

$$\begin{aligned} T &\rightarrow T^* \bullet F \\ F &\rightarrow \bullet (E) \\ F &\rightarrow \bullet id \end{aligned}$$

$I_8:$

$$\begin{aligned} F &\rightarrow (E \bullet) \\ E &\rightarrow E \bullet + T \end{aligned}$$

$I_9:$

$$\begin{aligned} E &\rightarrow E + T \bullet \\ T &\rightarrow T \bullet * F \end{aligned}$$

$I_{10}:$

$$T \rightarrow T^* F \bullet$$

$I_{11}:$

$$F \rightarrow (E) \bullet$$

$C = \{\text{Closure}(\{[S' \rightarrow \bullet S]\})\}$

while (C is still changing)

for each set of items I in C
and each grammar symbol X such that $\text{goto}(I, X)$ is not empty and not already in C
add $\text{goto}(I, X)$ to C

E'	\rightarrow	E		
E	\rightarrow	$E + T$	$ $	T
T	\rightarrow	$T^* F$	$ $	F
F	\rightarrow	(E)	$ $	id

Sets of LR(0) Items: Example

○ $I_0 = \text{closure}(\{[E' \rightarrow \bullet E]\})$

- $E' \rightarrow \bullet E,$
- $E \rightarrow \bullet E + T,$
- $E \rightarrow \bullet T,$
- $T \rightarrow \bullet T^* F,$
- $T \rightarrow \bullet F,$
- $F \rightarrow \bullet (E),$
- $F \rightarrow \bullet \text{id}$

C = {Closure({[S' → •S]})}
while (C is still changing)
for each set of items I in C
and each grammar symbol
X such that $\text{goto}(I, X)$ is not
empty and not already in C
add $\text{goto}(I, X)$ to C

E'	\rightarrow	E
E	\rightarrow	$E + T \quad \quad T$
T	\rightarrow	$T^* F \quad \quad F$
F	\rightarrow	$(E) \quad \quad \text{id}$

Sets of LR(0) Items: Example

- $I_1 = goto(I_0, E) = closure(\{[E' \rightarrow E^\bullet], [E \rightarrow E^\bullet + T]\})$
= $\{[E' \rightarrow E^\bullet], [E \rightarrow E^\bullet + T]\}$

$I_0:$
 $E' \rightarrow \bullet E$
 $E \rightarrow \bullet E + T$
 $E \rightarrow \bullet T$
 $T \rightarrow \bullet T^* F$
 $T \rightarrow \bullet F$
 $F \rightarrow \bullet (E)$
 $F \rightarrow \bullet id$

$C = \{\text{Closure}(\{[S' \rightarrow \bullet S]\})\}$
while (C is still changing)
for each set of items I in C
and each grammar symbol
X such that $goto(I, X)$ is not
empty and not already in C
add $goto(I, X)$ to C

E'	\rightarrow	E		
E	\rightarrow	$E + T$	$ $	T
T	\rightarrow	$T^* F$	$ $	F
F	\rightarrow	(E)	$ $	id

Sets of LR(0) Items: Example

- $I_2 = goto(I_0, T) = \text{closure}(\{[E \rightarrow T \bullet], [T \rightarrow T \bullet *F]\})$
= $\{[E \rightarrow T \bullet], [T \rightarrow T \bullet *T]\}$

$I_0:$
 $E' \rightarrow \bullet E$
 $E \rightarrow \bullet E + T$
 $E \rightarrow \bullet T$
 $T \rightarrow \bullet T^* F$
 $T \rightarrow \bullet F$
 $F \rightarrow \bullet (E)$
 $F \rightarrow \bullet id$

$C = \{\text{Closure}(\{[S' \rightarrow \bullet S]\})\}$
while (C is still changing)
for each set of items I in C
and each grammar symbol
X such that $goto(I, X)$ is not
empty and not already in C
add $goto(I, X)$ to C

E'	\rightarrow	E		
E	\rightarrow	$E + T$	$ $	T
T	\rightarrow	$T^* F$	$ $	F
F	\rightarrow	(E)	$ $	id

Sets of LR(0) Items: Example

○ $I_3 = goto(I_0, F) = \text{closure}(\{[T \rightarrow F \bullet]\})$
 $= \{[T \rightarrow F \bullet]\}$

$I_0:$
 $E' \rightarrow \bullet E$
 $E \rightarrow \bullet E + T$
 $E \rightarrow \bullet T$
 $T \rightarrow \bullet T^* F$
 $T \rightarrow \bullet F$
 $F \rightarrow \bullet (E)$
 $F \rightarrow \bullet id$

$C = \{\text{Closure}(\{[S' \rightarrow \bullet S]\})\}$
while (C is still changing)
for each set of items I in C
and each grammar symbol
X such that $goto(I, X)$ is not
empty and not already in C
add $goto(I, X)$ to C

E'	\rightarrow	E		
E	\rightarrow	$E + T$	$ $	T
T	\rightarrow	$T^* F$	$ $	F
F	\rightarrow	(E)	$ $	id

Sets of LR(0) Items: Example

- $I_4 = goto(I_0, () = \text{closure} (\{[F \rightarrow (\bullet E)]\})$
 $= \{ [F \rightarrow (\bullet E)],$
 $[E \rightarrow \bullet E+T],$
 $[E \rightarrow \bullet T],$
 $[T \rightarrow \bullet T^*F],$
 $[T \rightarrow \bullet F],$
 $[F \rightarrow \bullet (E)],$
 $[F \rightarrow \bullet id]\}$

$I_0:$

$E' \rightarrow \bullet E$
 $E \rightarrow \bullet E+T$
 $E \rightarrow \bullet T$
 $T \rightarrow \bullet T^*F$
 $T \rightarrow \bullet F$
 $F \rightarrow \bullet (E)$
 $F \rightarrow \bullet id$

$C = \{\text{Closure}(\{[S' \rightarrow \bullet S]\})\}$
while (C is still changing)
for each set of items I in C
and each grammar symbol
X such that $goto(I, X)$ is not
empty and not already in C
add $goto(I, X)$ to C

E'	\rightarrow	E		
E	\rightarrow	$E + T$	$ $	T
T	\rightarrow	T^*F	$ $	F
F	\rightarrow	(E)	$ $	id

Sets of LR(0) Items: Example

○ $I_5 = goto(I_0, id)$ = closure ($\{[F \rightarrow id^\bullet]\}$)
= $\{[F \rightarrow id^\bullet]\}$

$I_0:$
 $E' \rightarrow \bullet E$
 $E \rightarrow \bullet E + T$
 $E \rightarrow \bullet T$
 $T \rightarrow \bullet T^* F$
 $T \rightarrow \bullet F$
 $F \rightarrow \bullet (E)$
 $F \rightarrow \bullet id$

$C = \{\text{Closure}(\{[S' \rightarrow \bullet S]\})\}$
while (C is still changing)
for each set of items I in C
and each grammar symbol
X such that $goto(I, X)$ is not
empty and not already in C
add $goto(I, X)$ to C

E'	\rightarrow	E		
E	\rightarrow	$E + T$	$ $	T
T	\rightarrow	$T^* F$	$ $	F
F	\rightarrow	(E)	$ $	id

Sets of LR(0) Items: Example

- $I_6 = goto(I_1, +)$ = closure ($\{[E \rightarrow E + \bullet T]\}$)
= { $[E \rightarrow E + \bullet T]$,
 $[T \rightarrow \bullet T^* F]$,
 $[T \rightarrow \bullet F]$,
 $[F \rightarrow \bullet (E)]$,
 $[F \rightarrow \bullet id]$ }

$I_1:$
 $E' \rightarrow E \bullet$
 $E \rightarrow E \bullet + T$

$C = \{\text{Closure}(\{[S' \rightarrow \bullet S]\})\}$
while (C is still changing)
for each set of items I in C
and each grammar symbol X such that $goto(I, X)$ is not
empty and not already in C
add $goto(I, X)$ to C

$E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T^* F \mid F$
$F \rightarrow (E) \mid id$

Sets of LR(0) Items: Example

- $I_7 = goto(I_2, *) = closure (\{[T \rightarrow T^* \bullet F]\})$
 $= \{ [T \rightarrow T^* \bullet F],$
 $[F \rightarrow \bullet (E)],$
 $[F \rightarrow \bullet id]\}$

$I_2:$
 $E \rightarrow T \bullet$
 $T \rightarrow T \bullet * F$

$C = \{\text{Closure}(\{[S' \rightarrow \bullet S]\})\}$
while (C is still changing)
for each set of items I in C
and each grammar symbol X such that $goto(I, X)$ is not
empty and not already in C
add $goto(I, X)$ to C

$E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T^* F \mid F$
$F \rightarrow (E) \mid id$

Sets of LR(0) Items: Example

- $I_8 = goto(I_4, E) = closure(\{[F \rightarrow (E \bullet)], [E \rightarrow E \bullet + T]\})$
 $= \{ [F \rightarrow (E \bullet)], [E \rightarrow E \bullet + T]\}$

I_4 :

$F \rightarrow (\bullet E)$
 $E \rightarrow \bullet E + T$
 $E \rightarrow \bullet T$
 $T \rightarrow \bullet T^* F$
 $T \rightarrow \bullet F$
 $F \rightarrow \bullet (E)$
 $F \rightarrow \bullet id$

$C = \{\text{Closure}(\{[S' \rightarrow \bullet S]\})\}$
while (C is still changing)
for each set of items I in C
and each grammar symbol X such that $goto(I, X)$ is not
empty and not already in C
add $goto(I, X)$ to C

$E' \rightarrow E$
$E \rightarrow E + T \quad \quad T$
$T \rightarrow T^* F \quad \quad F$
$F \rightarrow (E) \quad \quad id$

Sets of LR(0) Items: Example

- $I_9 = goto(I_6, T) = \text{closure}(\{[E \rightarrow E + T \bullet], [T \rightarrow T \bullet *F]\})$
 $= \{ [E \rightarrow E + T \bullet], [T \rightarrow T \bullet *F] \}$

$I_6:$

$E \rightarrow E + \bullet T$
 $T \rightarrow \bullet T^* F$
 $T \rightarrow \bullet F$
 $F \rightarrow \bullet (E)$
 $F \rightarrow \bullet id$

$C = \{\text{Closure}(\{[S' \rightarrow \bullet S]\})\}$
while (C is still changing)
for each set of items I in C
and each grammar symbol X such that $goto(I, X)$ is not
empty and not already in C
add $goto(I, X)$ to C

E'	\rightarrow	E		
E	\rightarrow	$E + T$	$ $	T
T	\rightarrow	$T^* F$	$ $	F
F	\rightarrow	(E)	$ $	id

Sets of LR(0) Items: Example

○ $I_{10} = goto(I_7, F) = closure(\{[T \rightarrow T^*F^\bullet]\})$
 $= \{[T \rightarrow T^*F^\bullet]\}$

$I_7:$

$$T \rightarrow T^*\bullet F$$

$$F \rightarrow \bullet (E)$$

$$F \rightarrow \bullet id$$

$C = \{\text{Closure}(\{[S' \rightarrow \bullet S]\})\}$
while (C is still changing)
for each set of items I in C
and each grammar symbol X such that $goto(I, X)$ is not
empty and not already in C
add $goto(I, X)$ to C

$$\begin{array}{lcl} E' & \rightarrow & E \\ E & \rightarrow & E + T \quad | \quad T \\ T & \rightarrow & T^* F \quad | \quad F \\ F & \rightarrow & (E) \quad | \quad id \end{array}$$

Sets of LR(0) Items: Example

○ $I_{11} = goto(I_8,) = \text{closure}(\{[F \rightarrow (E)^\bullet]\})$
 $= \{[F \rightarrow (E)^\bullet]\}$

$I_8:$

$F \rightarrow (E^\bullet)$

$E \rightarrow E^\bullet + T$

$C = \{\text{Closure}(\{[S' \rightarrow \bullet S]\})\}$
while (C is still changing)
for each set of items I in C
and each grammar symbol
X such that $goto(I, X)$ is not
empty and not already in C
add $goto(I, X)$ to C

$E' \rightarrow E$
$E \rightarrow E + T \quad \quad T$
$T \rightarrow T * F \quad \quad F$
$F \rightarrow (E) \quad \quad \text{id}$

Sets of LR(0) Items: Example

$I_0:$

$$\begin{aligned} E' &\rightarrow \bullet E \\ E &\rightarrow \bullet E + T \\ E &\rightarrow \bullet T \\ T &\rightarrow \bullet T^* F \\ T &\rightarrow \bullet F \\ F &\rightarrow \bullet (E) \\ F &\rightarrow \bullet id \end{aligned}$$

$I_1:$

$$\begin{aligned} E' &\rightarrow E \bullet \\ E &\rightarrow E \bullet + T \end{aligned}$$

$I_2:$

$$\begin{aligned} E &\rightarrow T \bullet \\ T &\rightarrow T \bullet * F \end{aligned}$$

$I_3:$

$$T \rightarrow F \bullet$$

$I_4:$

$$\begin{aligned} F &\rightarrow (\bullet E) \\ E &\rightarrow \bullet E + T \\ E &\rightarrow \bullet T \\ T &\rightarrow \bullet T^* F \\ T &\rightarrow \bullet F \\ F &\rightarrow \bullet (E) \\ F &\rightarrow \bullet id \end{aligned}$$

$I_5:$

$$F \rightarrow id \bullet$$

$I_6:$

$$\begin{aligned} E &\rightarrow E + \bullet T \\ T &\rightarrow \bullet T^* F \\ T &\rightarrow \bullet F \\ F &\rightarrow \bullet (E) \\ F &\rightarrow \bullet id \end{aligned}$$

$I_7:$

$$\begin{aligned} T &\rightarrow T^* \bullet F \\ F &\rightarrow \bullet (E) \\ F &\rightarrow \bullet id \end{aligned}$$

$I_8:$

$$\begin{aligned} F &\rightarrow (E \bullet) \\ E &\rightarrow E \bullet + T \end{aligned}$$

$I_9:$

$$\begin{aligned} E &\rightarrow E + T \bullet \\ T &\rightarrow T \bullet * F \end{aligned}$$

$I_{10}:$

$$T \rightarrow T^* F \bullet$$

$I_{11}:$

$$F \rightarrow (E) \bullet$$

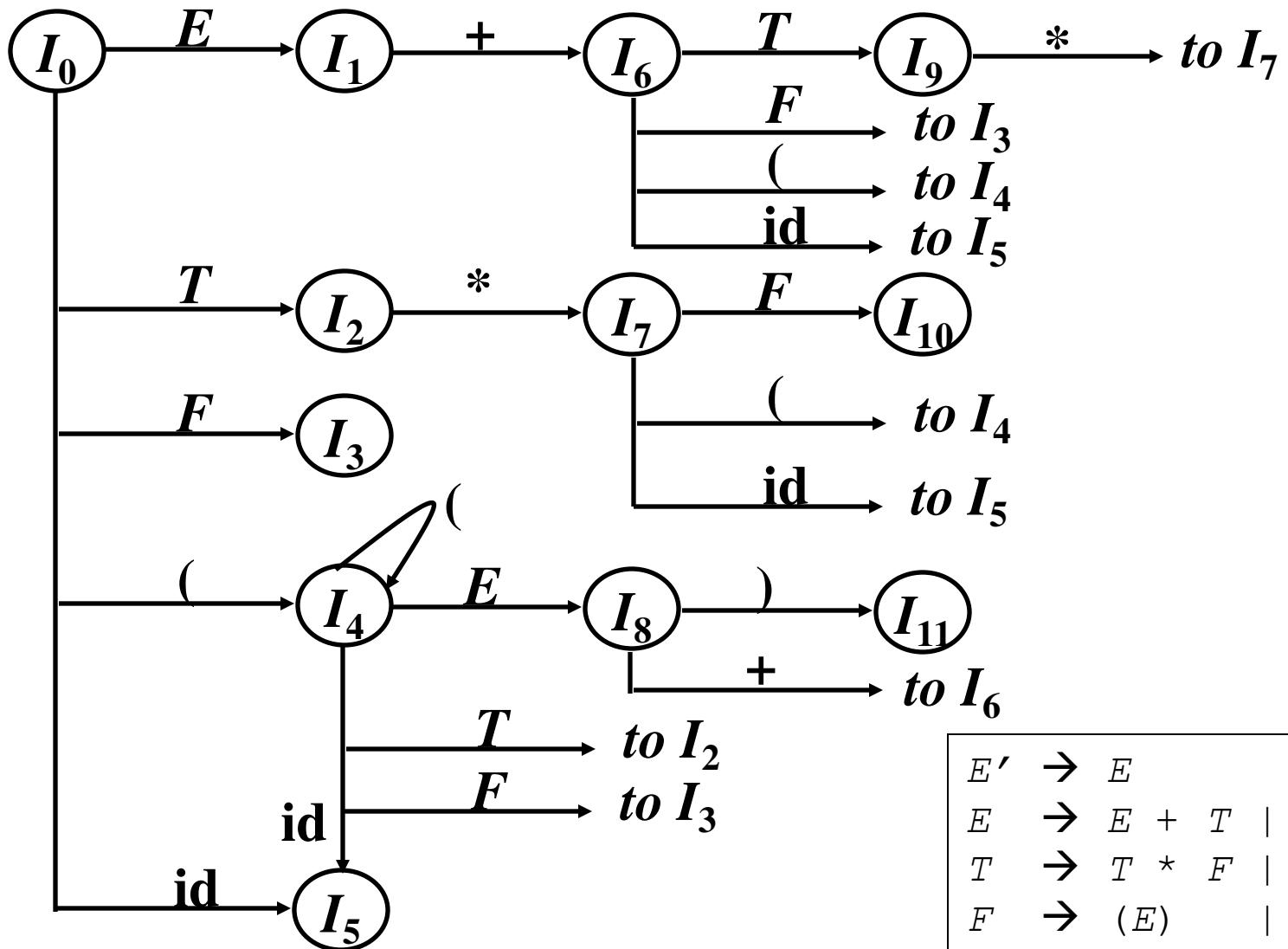
$C = \{\text{Closure}(\{[S' \rightarrow \bullet S]\})\}$

while (C is still changing)

for each set of items I in C
and each grammar symbol X such that $\text{goto}(I, X)$ is not empty and not already in C
add $\text{goto}(I, X)$ to C

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E + T \quad | \quad T \\ T &\rightarrow T^* F \quad | \quad F \\ F &\rightarrow (E) \quad | \quad id \end{aligned}$$

Transition Diagram for LR(0)



$E' \rightarrow E$	$E \rightarrow E + T$	$T \rightarrow T * F$	$F \rightarrow (E)$	$ $	id
--------------------	-----------------------	-----------------------	---------------------	-----	-------------

goto Function

goto

I	id	+	*	()	E	T	F
0	5			4		1	2	3
1		6						
2			7					
3								
4	5			4		8	2	3
5								
6	5			4			9	3
7	5			4				10
8		6			11			
9			7					
10								
11								

SLR Table Algorithm

1. Construct C , the collection of sets of LR(0) items
2. State i corresponds to I_i
 - a) If $[A \rightarrow \alpha \cdot a\beta]$ is in I_i , and $\text{goto}(I_i, a) = I_j$ then
 $\text{action}[i, a] = \text{"shift } j\text{"}$ for terminal a
 - b) If $[A \rightarrow \alpha \cdot]$ is in I_i , then $\text{action}[i, a] = \text{"reduce } A \rightarrow \alpha\text{"}$ for all a in $\text{FOLLOW}(A)$; except for $A=S'$
 - c) If $[S' \rightarrow S \cdot]$ is in I_i , then $\text{action}[i, \$] = \text{"accept"}$
3. if $\text{goto}(I_i, A) = I_j$, then $\text{goto}[i, A] = j$ for nonterminals A
4. Initial state corresponds to I_0 .
5. Undefined entries are errors

Example

$I_0:$

$$\begin{aligned} E' &\rightarrow \bullet E \\ E &\rightarrow \bullet E + T \\ E &\rightarrow \bullet T \\ T &\rightarrow \bullet T^* F \\ T &\rightarrow \bullet F \\ F &\rightarrow \bullet (E) \\ F &\rightarrow \bullet id \end{aligned}$$

$I_1:$

$$\begin{aligned} E' &\rightarrow E \bullet \\ E &\rightarrow E \bullet + T \end{aligned}$$

$I_2:$

$$\begin{aligned} E &\rightarrow T \bullet \\ T &\rightarrow T \bullet * F \end{aligned}$$

$I_3:$

$$T \rightarrow F \bullet$$

$F \rightarrow \bullet (E)$	implies $\text{action}[0, ()] = \text{shift } 4$
$F \rightarrow \bullet id$	implies $\text{action}[0, id] = \text{shift } 5$
$E' \rightarrow E \bullet$	implies $\text{action}[1, \$] = \text{accept}$
$E \rightarrow E \bullet + T$	implies $\text{action}[1, +] = \text{shift } 6$
$E \rightarrow T \bullet$	implies $\text{action}[2, \$] = \text{action}[2, +] = \text{action}[2, ()] = \text{reduce } E \rightarrow T$ since $\text{Follow}(E) = \{\$, +, ()\}$
$T \rightarrow T \bullet * F$	implies $\text{action}[2, *] = \text{shift } 7$
$T \rightarrow F \bullet$	implies $\text{action}[3, \$] = \text{action}[3, +] = \text{action}[3, ()] = \text{action}[3, *] = \text{reduce } T \rightarrow F$ since $\text{Follow}(T) = \{\$, +, *, ()\}$

1. Construct C , the canonical collection
 1. If $[A \rightarrow \beta \bullet a\delta]$ is in li , then $\text{action}[i, a] = \text{"shift } j\text{"}$ where $\text{goto}(li, a) = l_j$ for terminal a
 2. If $[A \rightarrow \beta \bullet]$ is in li , then $\text{action}[i, a] = \text{"reduce } A \rightarrow \beta\text{"}$ for all a in $\text{FOLLOW}(A)$; except for $A=S$
 3. If $[S' \rightarrow S \bullet]$ is in li , then $\text{action}[i, \$] = \text{"accept"}$
2. State I corresponds to l_i
3. $\text{Goto}(li, A) = l_j$ implies $\text{goto}[l_i, A] = j$ for nonterminals A

goto Function

goto

I	id	+	*	()	E	T	F
0	5			4		1	2	3
1		6						
2			7					
3								
4	5			4		8	2	3
5								
6	5			4			9	3
7	5			4				10
8		6			11			
9			7					
10								
11								

1	E	$\rightarrow E + T$
2	E	$\rightarrow T$
3	T	$\rightarrow T * F$
4	T	$\rightarrow F$
5	F	$\rightarrow (E)$
6	F	$\rightarrow \text{id}$

Example Parses

<i>si</i>	shift and push state <i>i</i>
<i>rj</i>	reduce a production # <i>j</i>
<i>acc</i>	accept
<i>blank</i>	error

action

goto

State	id	+	*	()	\$	E	T	F
0	s5				s4		1	2	3
1		s6				acc			
2		r2	s7			r2	r2		
3		r4	r4			r4	r4		
4	s5				s4		8	2	3
5		r6	r6			r6	r6		
6	s5				s4			9	3
7	s5				s4				10
8		s6				s11			
9		r1	s7			r1	r1		
10		r3	r3			r3	r3		
11		r5	r5			r5	r5		

SLR(1) Grammars

- If parsing table contains conflict actions, then we say the grammar is not SLR(1)
- A grammar having an SLR(1) parsing table is said to be SLR(1)
- Every SLR(1) grammar is unambiguous
- But there are many unambiguous grammars that are not SLR(1)
- Read Example 4.39

SLR(1) Grammars

- A grammar is SLR(1) if and only if, for any state s , the following two conditions are satisfied
 - For any item $A \rightarrow \alpha \cdot a\beta$ in s with $a \in T$ (terminal), there is **no** complete item $B \rightarrow \gamma \cdot$ in s with $a \in \text{Follow}(B)$
 - For any two complete items
 - $A \rightarrow \alpha \cdot$ and $B \rightarrow \gamma \cdot$ in s ,
 - $\text{Follow}(A) \cap \text{Follow}(B) = \emptyset$

LR(1) items

- An LR(1) item is a pair $[A \rightarrow \alpha \cdot \beta, a]$, where
 - $A \rightarrow \alpha \beta$ is a production, and
 - a is a lookahead character (terminal symbol or $\$$)
- Examples:
 - $[A \rightarrow \cdot \beta \gamma \delta, a] \quad [A \rightarrow \beta \cdot \gamma \delta, a]$
 - $[A \rightarrow \beta \gamma \cdot \delta, a] \quad [A \rightarrow \beta \gamma \delta \cdot, a]$
- The \cdot in an item indicates the position of the top of the stack
- LR(0) items $[A \rightarrow \beta \cdot \gamma \delta]$ (no lookahead symbol)
- LR(1) items $[A \rightarrow \beta \cdot \gamma \delta, a]$ (one token lookahead)

Computing Closure of LR(1) items

Closure(I)

while (I is still changing)

for each item $[A \rightarrow \alpha \cdot B\beta, a]$ in I

for each production $B \rightarrow \gamma$

for each terminal $b \in \text{FIRST}(\beta a)$

if $[B \rightarrow \cdot \gamma, b] \notin I$

then add $[B \rightarrow \cdot \gamma, b]$ to I

Closure(I)

J = I;

while (J is still changing)

for each item $[A \rightarrow \alpha \cdot B\beta] \in J$

for each production $B \rightarrow \gamma \in P$

add $[B \rightarrow \cdot \gamma]$ to J

Return J;

**Computing closure of
LR(0) items:**

LR(1) Parsers: Closure

- Consider

- $S' \rightarrow S$
- $S \rightarrow CC$
- $C \rightarrow cC \mid d$

- Closure of $\{[S' \rightarrow \bullet S, \$]\}$

- $[A \rightarrow \alpha \bullet B\beta, a]$

- $A = S'$, $\alpha = \epsilon$, $B = S$, $\beta = \epsilon$, and $a = \$$

- Add $[B \rightarrow \bullet \gamma, b]$ for each production $B \rightarrow \gamma$, and terminal b in $FIRST(\beta a)$

- $B \rightarrow \gamma$ must be $S \rightarrow CC$

- Since β is ϵ and a is $\$$, b may only be $\$$

- Add $[S \rightarrow \bullet CC, \$]$

Closure(I)

```
while ( I is still changing )
    for each item [A → α•Bβ, a] in I
        for each production B → γ in G'
            for each terminal b ∈ FIRST(βa)
                if [B → •γ , b] ∉ I
                    then add [B → •γ , b] to I
```

LR(1) Parsers: Closure

- Consider
 - $S' \rightarrow S$
 - $S \rightarrow CC$
 - $C \rightarrow cC \mid d$

- Closure of $\{[S \rightarrow \bullet CC, \$]\}$

- $[A \rightarrow \alpha \cdot B\beta, a]$

- $A = S, \alpha = \epsilon, B = C, \beta = C$, and $a = \$$
 - Add $[B \rightarrow \bullet \gamma, b]$ for each production $B \rightarrow \gamma$, and terminal b in $\text{FIRST}(\beta a)$
 - $B \rightarrow \gamma$ must be $C \rightarrow cC$ and $C \rightarrow d$
 - Since C is not ϵ and a is $\$$, then $\text{FIRST}(C\$) = \text{FIRST}(C) = \{c, d\}$
 - Add $[C \rightarrow \bullet cC, c]$, $[C \rightarrow \bullet cC, d]$, $[C \rightarrow \bullet d, c]$, and $[C \rightarrow \bullet d, d]$

Closure(I)

```
while ( I is still changing )
    for each item [A → α·Bβ, a] in I
        for each production B → γ in G'
            for each terminal b ∈ FIRST(βa)
                if [B → •γ , b] ∉ I
                    then add [B → •γ , b] to I
```

LR(1) Parsers: Closure

- None of the new items has a nonterminal immediately to the right of the dot Consider, so
- $I_0 = \{[S' \rightarrow \bullet S, \$], [S \rightarrow \bullet CC, \$], [C \rightarrow \bullet cC, c], [C \rightarrow \bullet cC, d], [C \rightarrow \bullet d, c], [C \rightarrow \bullet d, d]\}$

1	$S \rightarrow CC$
2	$C \rightarrow cC$
3	$C \rightarrow d$

Closure(I)

```
while (  $I$  is still changing )
    for each item  $[A \rightarrow \alpha \cdot B\beta, a]$  in  $I$ 
        for each production  $B \rightarrow \gamma$  in  $G'$ 
            for each terminal  $b \in \text{FIRST}(\beta a)$ 
                if  $[B \rightarrow \bullet \gamma, b] \notin I$ 
                    then add  $[B \rightarrow \bullet \gamma, b]$  to  $I$ 
```

Algorithms for LR(1) Parsers

Computing closure of set of LR(1) items:

Closure(I)

```
while ( I is still changing )
  for each item [A → α • Bβ, a] in I
    for each production B → γ ∈ P
      for each terminal b ∈ FIRST(βa)
        if [B → • γ , b] ∉ I
          then add [B → • γ , b] to I
```

Computing goto for set of LR(1) items:

Goto(I, X)

new ← \emptyset

```
for each item [A → α • X β , a] in I
  new ← new ∪ [A → α X • β , a]
return closure(new)
```

Constructing canonical collection of LR(1) items:

```
C ← closure( [S' → • S , $] )
while ( C is still changing )
  for each set of items I ∈ C and for each X ∈ (T ∪ NT)
    C ← C ∪ goto(I, X), if it is non-empty and not already in C
```

Construction of the sets of LR(1) items

- The initial set of items is

- I_0 :

- $S' \rightarrow \bullet S, \quad \$$
 - $S \rightarrow \bullet CC, \quad \$$
 - $C \rightarrow \bullet cC, \quad c/d$
 - $C \rightarrow \bullet d, \quad c/d$

1	$S \rightarrow CC$
2	$C \rightarrow cC$
3	$C \rightarrow d$

- $[C \rightarrow \bullet cC, c/d]$ is a shorthand for the two items $[C \rightarrow \bullet cC, c]$ and $[C \rightarrow \bullet cC, d]$
 - Now we compute $\text{goto}(I_0, X)$ for the various values of X

- | | |
|---|--------------------|
| 1 | $S \rightarrow CC$ |
| 2 | $C \rightarrow cC$ |
| 3 | $C \rightarrow d$ |

Construction of the sets of LR(1) items

- $I_1 = \text{goto}(I_0, S) = \text{closure}(\{[S' \rightarrow S\bullet, \$]\}) =$
 - $\{[S' \rightarrow S\bullet, \$]\}$
 - No additional closure is possible, since the dot is at the right end
 - $I_2 = \text{goto}(I_0, C) = \text{closure}(\{[S \rightarrow C\bullet C, \$]\}) =$
 - $\{[S \rightarrow C\bullet C, \$]\}$
 - $[C \rightarrow \bullet cC, \$]$
 - $[C \rightarrow \bullet d, \$]$
 - $I_3 = \text{goto}(I_0, c) = \text{closure}(\{[C \rightarrow c\bullet C, c/d]\}) =$
 - $\{[C \rightarrow c\bullet C, c/d]\}$
 - $[C \rightarrow \bullet cC, c/d]$
 - $[C \rightarrow \bullet d, c/d]$
- $I_0:$
- | | | | | |
|------|---------------|-----------|-------|-------|
| S' | \rightarrow | \bullet | $S,$ | $\$$ |
| S | \rightarrow | \bullet | $CC,$ | $\$$ |
| C | \rightarrow | \bullet | $cC,$ | c/d |
| C | \rightarrow | \bullet | $d,$ | c/d |

1	$S \rightarrow CC$
2	$C \rightarrow cC$
3	$C \rightarrow d$

Construction of the sets of LR(1) items

- $I_4 = \text{goto}(I_0, d) = \text{closure}(\{[C \rightarrow d\bullet, c/d]\}) =$
 - $\{[C \rightarrow d\bullet, c/d]\}$
 - We have finished considering *goto* on I_0
- For I_1 , we got no new states
- $I_5 = \text{goto}(I_2, C) = \text{closure}(\{[S \rightarrow CC\bullet, \$]\}) =$
 - $\{[S \rightarrow CC\bullet, \$]\}$
- $I_6 = \text{goto}(I_2, c) = \text{closure}(\{[C \rightarrow c\bullet C, \$]\}) =$
 - $\{[C \rightarrow c\bullet C, \$]\}$
 - $\{[C \rightarrow \bullet c C, \$]\}$
 - $\{[C \rightarrow \bullet d, \$]\}$
- $I_7 = \text{goto}(I_2, d) =$
 - $\text{closure}(\{[C \rightarrow d\bullet, \$]\}) =$
 - $\{[C \rightarrow d\bullet, \$]\}$

$I_2 :$

$S \rightarrow C\bullet C , \$$
$C \rightarrow \bullet C C , \$$
$C \rightarrow \bullet d , \$$

Construction of the sets of LR(1) items

- $I_8 = \text{goto}(I_3, C) = \text{closure}(\{[C \rightarrow cC\bullet, c/d]\}) =$
 - $\{[C \rightarrow cC\bullet, c/d]\}$
- $\text{goto}(I_3, c) = I_3$
- $\text{goto}(I_3, d) = I_4$
- I_4 and I_5 have no *goto*'s
- $\text{goto}(I_6, c) = I_6$
- $\text{goto}(I_6, d) = I_7$
- $I_9 = \text{goto}(I_6, C) = \text{closure}(\{[C \rightarrow cC\bullet, \$]\}) =$
 - $\{[C \rightarrow cC\bullet, \$]\}$
- Remaining sets of items yield no *goto*'s, so we done.

1	$S \rightarrow CC$
2	$C \rightarrow cC$
3	$C \rightarrow d$

$I_3:$

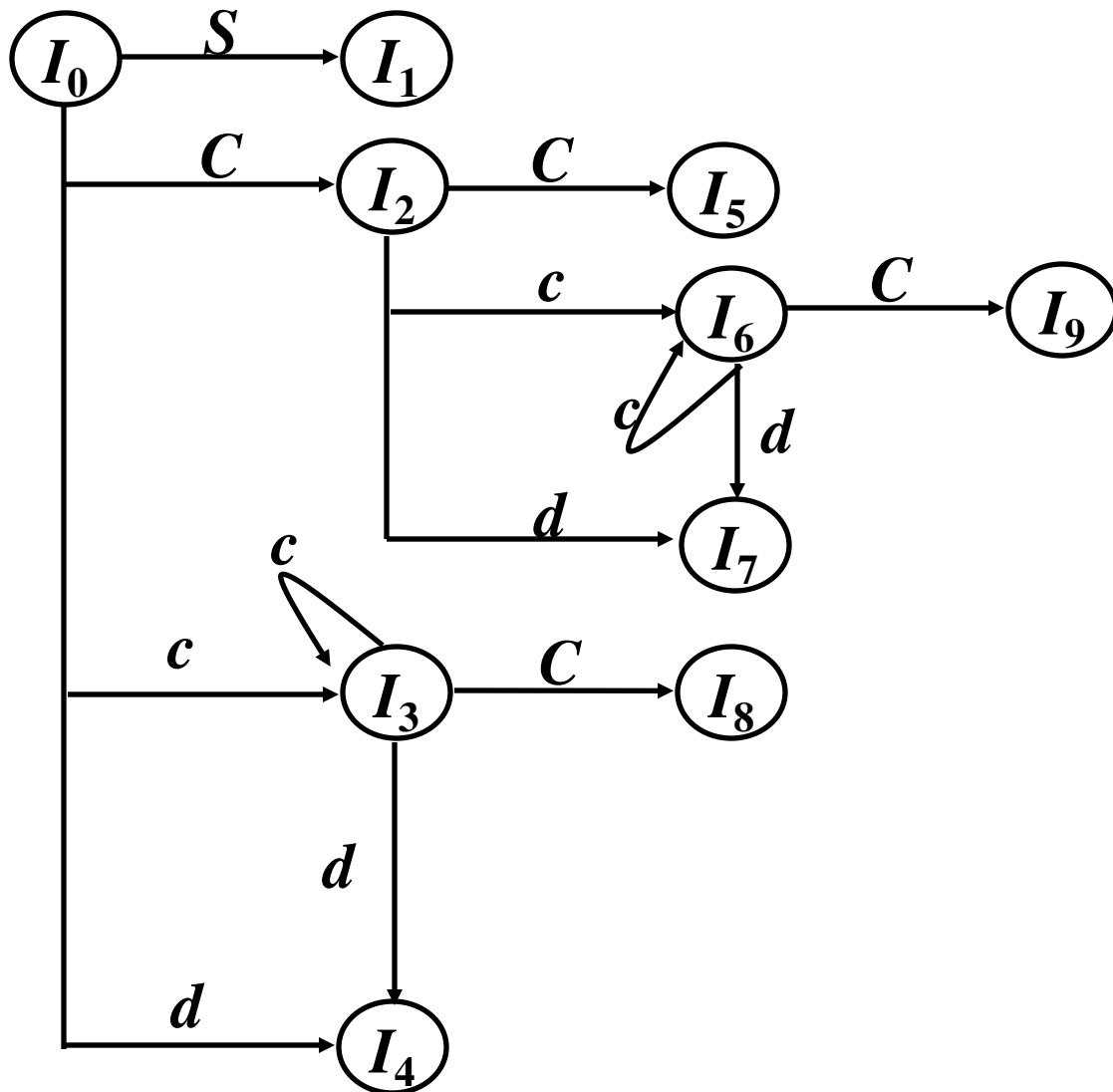
$C \rightarrow c\bullet C, c/d$
 $C \rightarrow \bullet cC, c/d$
 $C \rightarrow \bullet d, c/d$

$I_6:$

$C \rightarrow c\bullet C, \$$
 $C \rightarrow \bullet cC, \$$
 $C \rightarrow \bullet d, \$$

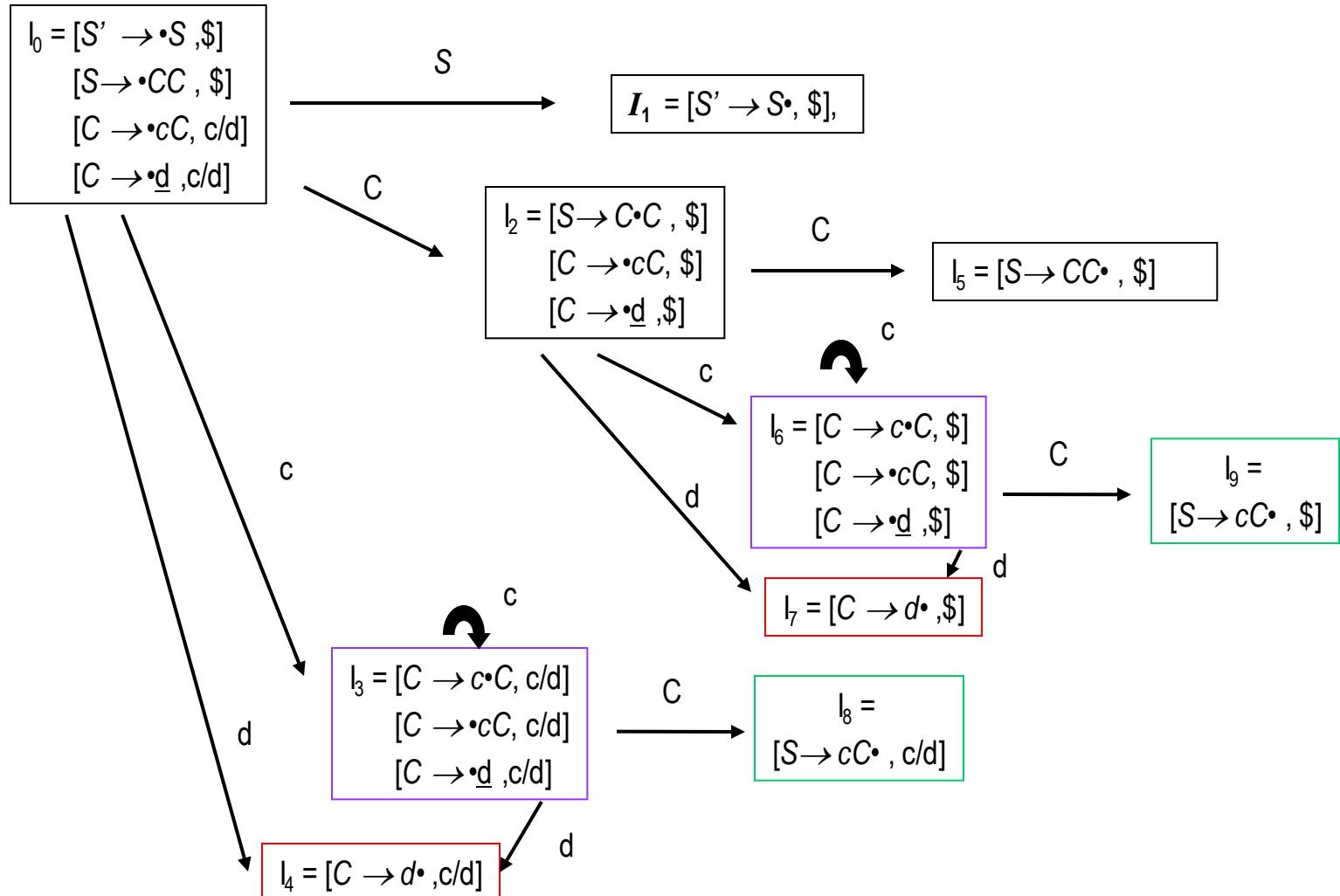
The *goto* graph

1	$S \rightarrow CC$
2	$C \rightarrow cC$
3	$C \rightarrow d$



Example

Start with the LR(1) construction



Constructing the ACTION and GOTO Tables of LR1 Parser

1. Construct $C = \{I_0, I_1, \dots, I_n\}$, the canonical LR(1) collection
2. State i corresponds to I_i
 1. If $[A \rightarrow \alpha \cdot a\beta, x]$ is in I_i , then $\text{action}[i,a] = \text{"shift } j"$ where $\text{goto}(I_i, a) = I_j$, for terminal a
 2. If $[A \rightarrow \alpha \cdot, a]$ is in I_i , then $\text{action}[i,a] = \text{"reduce } A \rightarrow \alpha"$, except for $A=S$
 3. If $[S' \rightarrow S \cdot, \$]$ is in I_i , then $\text{action}[i,\$] = \text{"accept"}$
 - $\text{goto}(I_i, A) = I_j$ implies $\text{goto}[i, A] = j$ for nonterminals A
 - Initial state corresponds to I_0 .
 - Undefined entries are **error**.

1	$S \rightarrow CC$
2	$C \rightarrow cC$
3	$C \rightarrow d$

Example

$I_0:$	$C \rightarrow \bullet cC, c/d$	implies action[0,c] = shift 3
	$C \rightarrow \bullet d, c/d$	implies action[0,d] = shift 4
	$S' \rightarrow S \bullet, \$$	implies action[1,\$] = accept
	$C \rightarrow \bullet cC, \$$	implies action[2,c] = shift 6
	$C \rightarrow \bullet d, \$$	implies action[2,d] = shift 7
	$C \rightarrow \bullet cC, c/d$	implies action[3,c] = shift 3
	$C \rightarrow \bullet d, c/d$	implies action[3,d] = shift 4
$I_1:$	$C \rightarrow \bullet cC, c/d$	implies action[4,c] = action[4,d] = reduce $C \rightarrow d$
	$C \rightarrow \bullet d, c/d$	
$I_2:$	$C \rightarrow d \bullet, c/d$	
	$S \rightarrow C \bullet C, \$$	
	$C \rightarrow \bullet cC, \$$	
	$C \rightarrow \bullet d, \$$	
$I_3:$		
	$C \rightarrow c \bullet C, c/d$	1. Construct $C = \{I_0, I_1, \dots, I_n\}$, the canonical LR(1) collection
	$C \rightarrow \bullet cC, c/d$	2. State i corresponds to I_i
	$C \rightarrow \bullet d, c/d$	1. If $[A \rightarrow \alpha \bullet a\beta, x]$ is in I_i , then $\text{action}[i,a] = \text{"shift } j \text{"}$ where $\text{goto}(I_i, a) = I_j$, for terminal a
$I_4:$		2. If $[A \rightarrow \alpha \bullet, a]$ is in I_i , then $\text{action}[i,a] = \text{"reduce } A \rightarrow \alpha \text{"}$, except for $A=S$
	$C \rightarrow d \bullet, c/d$	3. If $[S' \rightarrow S \bullet, \$]$ is in I_i , then $\text{action}[i,\$] = \text{"accept"}$
		• $\text{goto}(I_i, A) = I_j$ implies $\text{goto}[i,A] = j$ for nonterminals A
		• Initial state corresponds to I_0 .
		• Undefined entries are error.

Example Parses

1 $S \rightarrow CC$
 2 $C \rightarrow cC$
 3 $C \rightarrow d$

si shift and push state *i*
rj reduce a production # *j*
acc accept
blank error

State	action			goto	
	c	d	\$	s	c
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

What can go wrong in LR(1) parsing?

- What if set I contains $[\alpha \rightarrow \beta \bullet \underline{a}\gamma, b]$ and $[\alpha \rightarrow \beta \bullet, \underline{a}]$?
 - First item $[\alpha \rightarrow \beta \bullet \underline{a}\gamma, b]$ generates “shift”,
 - Second item $[\alpha \rightarrow \beta \bullet, \underline{a}]$ generates “reduce”
 - Action[s_i, a] = {shift, reduce}
 - This is called a *shift/reduce conflict*
 - Modify the grammar to eliminate it (*if-then-else*)
 - Shifting will often resolve it correctly
- What if set I contains $[\alpha \rightarrow \gamma \bullet, \underline{a}]$ and $[\beta \rightarrow \gamma \bullet, \underline{a}]$?
 - Each generates “reduce”, but with a different production
 - Action[s_i, a] = {reduce, reduce}
 - This is called a *reduce/reduce conflict*
 - Modify the grammar to eliminate it
- In either case, the grammar is not LR(1)

LALR parsing

- We have two bottom-up parsing methods:
 - SLR and LR
- Which do we choose? Neither
 - SLR is not powerful enough
 - Smallest class of grammars
 - LR parsing tables are too big
 - 1000's of states vs. 100's for SLR parser
- In practice, use LALR
 - A compromise between SLR and LR
 - Stands for “Look-Ahead LR”
 - Both SLR & LALR have the same number of states

Core

- The **core** of a set of LR(1) items is
 - the set of first components
 - the set of LR(0) items derived by ignoring the lookahead symbols
- The core of
 - $\{[X \rightarrow \alpha\bullet, b], [X \rightarrow \alpha\bullet, c], [X \rightarrow \beta\bullet, d]\}$ is
 - $\{[X \rightarrow \alpha\bullet], [X \rightarrow \beta\bullet]\}$
- The two sets
 - $\{[A \rightarrow \alpha\bullet\beta, a], [A \rightarrow \alpha\bullet\beta, b]\}$
 - $\{[A \rightarrow \alpha\bullet\beta, c], [A \rightarrow \alpha\bullet\beta, d]\}$
 - $\{[A \rightarrow \alpha\bullet\beta], [A \rightarrow \alpha\bullet\beta]\}$

have the same core, even if the lookahead are different

Key Idea

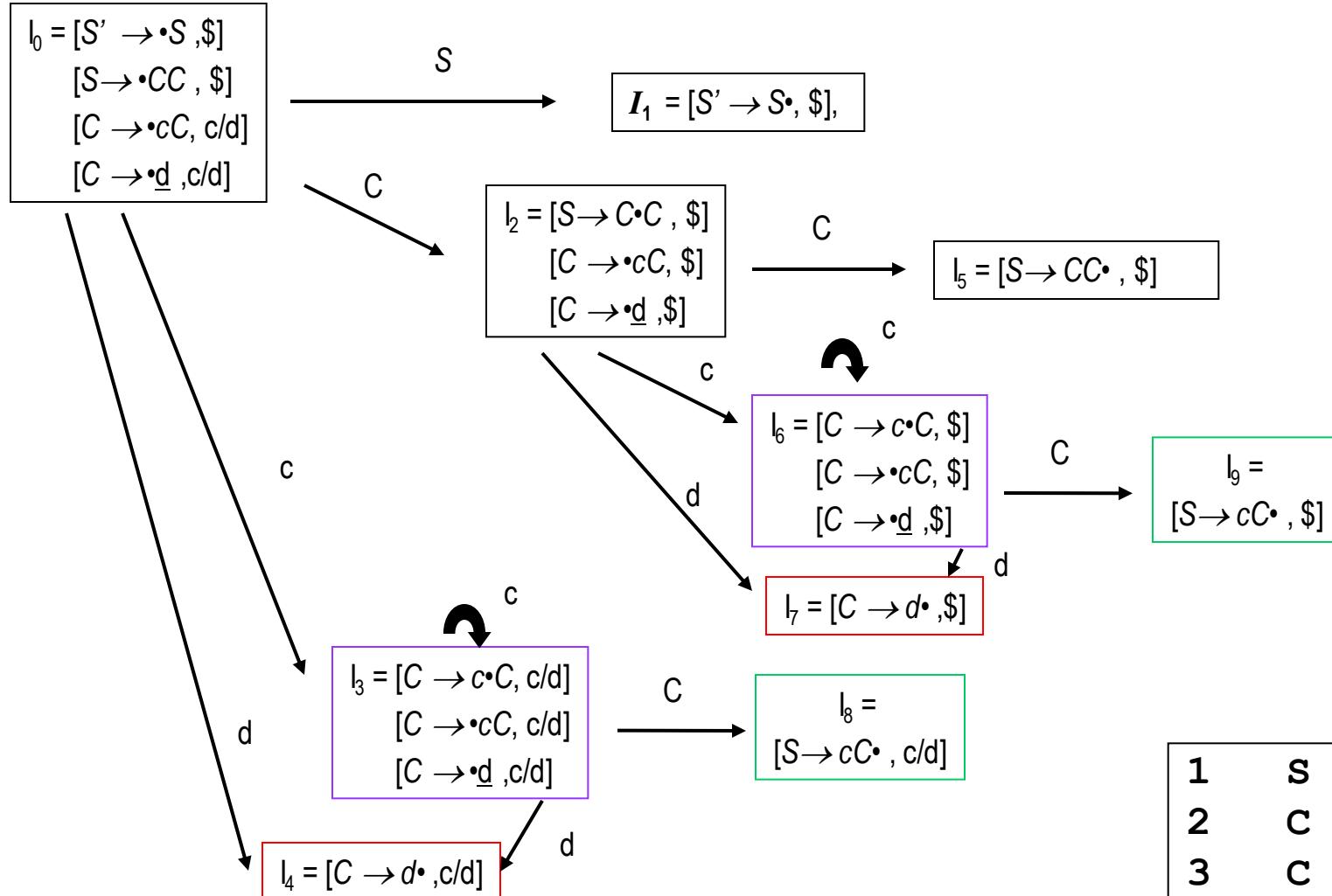
- If two states I_i and I_j , have the **same** core we can **merge** those states in the *action* and *goto* tables
- A LALR(1) can be constructed from an LR(1) as follows:
 - Repeat until all states have distinct cores
 - Choose any two distinct states in LR(1) with the same core
 - Merge the states by creating one state with the union of two items
 - Predecessors of the old states must be modified to point to the new state
 - The new state points to all the successors of the old states

Constructing the ACTION and GOTO Tables of LALR Parser

- Construct $C = \{I_0, I_1, \dots, I_n\}$, the canonical LR(1) collection
- For each **core** present among the set of LR(1) items, find all sets having that core and replace these sets by their union
- The parsing actions are constructed in the same manner as LR(1)
- Update the **goto** function to reflect the replacement sets

Example

Start with the LR(1) construction



Example

- Replace I_3 and I_6 by I_{36}
 - $I_3 = \{[C \rightarrow c \cdot C, c/d], [C \rightarrow \cdot cC, c/d], [C \rightarrow \cdot d, c/d]\}$
 - $I_6 = \{[C \rightarrow c \cdot C, \$], [C \rightarrow \cdot cC, \$], [C \rightarrow \cdot d, \$]\}$
 - $I_{36} = \{[C \rightarrow c \cdot C, c/d/\$], [C \rightarrow \cdot cC, c/d/\$], [C \rightarrow \cdot d, c/d/\$]\}$
- Replace I_4 and I_7 by I_{47}
 - $I_4 = \{[C \rightarrow d \cdot, c/d]\}$
 - $I_7 = \{[C \rightarrow d \cdot, \$]\}$
 - $I_{47} = \{[C \rightarrow d \cdot, c/d/\$]\}$
- Replace I_8 and I_9 by I_{89}
 - $I_8 = \{[C \rightarrow cC \cdot, c/d]\}$
 - $I_9 = \{[C \rightarrow cC \cdot, \$]\}$
 - $I_{89} = \{[C \rightarrow cC \cdot, c/d/\$]\}$

1	s	→	cc
2	c	→	cc
3	c	→	d

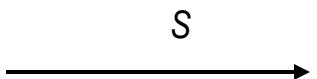
Example

1	s	→	cc
2	c	→	cc
3	c	→	d

- $\text{goto}(I_0, c)$:
 - In LR(1), $\text{goto}(I_0, c) = I_3$, and I_3 is part of I_{36} in LALR so,
 - $\text{goto}(I_0, c) = I_{36}$, in LALR
- $\text{goto}(I_2, c)$:
 - In LR(1), $\text{goto}(I_2, c) = I_6$, and I_6 is part of I_{36} in LALR so,
 - $\text{goto}(I_2, c) = I_{36}$, in LALR
- $\text{goto}(I_2, d)$:
 - In LR(1), $\text{goto}(I_2, d) = I_7$, and I_7 is part of I_{47} in LALR so,
 - $\text{goto}(I_2, d) = I_{47}$, in LALR
- $\text{goto}(I_{36}, C)$:
 - $\text{goto}(I_3, C) = I_8$, and I_8 is part of I_{89} , and
 - $\text{goto}(I_6, C) = I_9$, and I_9 is part of I_{89} , so
 - $\text{goto}(I_{36}, C) = I_{89}$

Example

Start with the LR(1) construction

$$I_0 = [S' \rightarrow \bullet S, \$]
[S \rightarrow \bullet CC, \$]
[C \rightarrow \bullet cC, c/d]
[C \rightarrow \bullet d, c/d]$$


$$I_1 = [S' \rightarrow S \bullet, \$],$$

$$I_2 = [S \rightarrow C \bullet C, \$]
[C \rightarrow \bullet cC, \$]
[C \rightarrow \bullet d, \$]$$

C

$$I_5 = [S \rightarrow CC \bullet, \$]$$

c

$$I_6 = [C \rightarrow c \bullet C, \$]
[C \rightarrow \bullet cC, \$]
[C \rightarrow \bullet d, \$]$$

C

$$I_9 = [S \rightarrow cC \bullet, \$]$$

c

$$I_3 = [C \rightarrow c \bullet C, c/d]
[C \rightarrow \bullet cC, c/d]
[C \rightarrow \bullet d, c/d]$$

C

$$I_8 = [S \rightarrow cC \bullet, c/d]$$

d

$$I_4 = [C \rightarrow d \bullet, c/d]$$

5 . 95

	Action			Go To
	c	d	\$	S C
0	s36	s47		1 2
1			acc	
2	s36	s47		5
36	s36	s47		89
47	r3	r3	r3	
5			r1	
89	r2	r2	r2	

1	S	\rightarrow	CC
2	C	\rightarrow	cC
3	C	\rightarrow	d

LR(1) & LALR(1)

State	<i>action</i>			<i>goto</i>	
	c	d	\$	s	c
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

LR

1 $S \rightarrow CC$
 2 $C \rightarrow cC$
 3 $C \rightarrow d$

	Action			Go To	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

LALR

$L(G) = c^*dc^*d$

LR verses LALR

- Both LR & LALR make exactly the same sequence of shifts and reductions on correct input
- However, when parsing an incorrect input:
 - LALR parser may proceed to do some reductions after the LR parser has declared an error
 - LALR will never shift another symbol after the LR parser declares an error
- Example: on input ccd\$
 - LR parser will put 0 c 3 c 3 d 4 on the stack
 - In state 4 will discover an error on \$

LR verses LALR

- Example: on input ccd\$
 - LALR stack will have 0 c 36 c 36 d 47
 - State 47 on input \$ has action reduce C → d
 - Stack will have 0 c 36 c 36 C 89
 - State 89 on input \$ has action reduce C → cC
 - Stack will have 0 c 36 C 89
 - State 89 on input \$ has action reduce C → cC
 - Stack will have 0 C 2
 - Finally state 2 has action error on input \$

Conflicts in LALR Parser

- The **core** of $\text{goto}(I, X)$ depends only on core I , not on X , so just merge the **goto** function for merged states
 - Assume states s_1 and s_2 have the same core and
 - $\text{goto}(s_1, X) = s_i$ and $\text{goto}(s_2, X) = s_j$
- Now
 - $\text{goto}(\text{Merge}(s_1, s_2), X) = s_i$
 - $\text{goto}(\text{Merge}(s_1, s_2), X) = s_j$
 - Is this a conflict?
- No. If s_1 and s_2 have the same core, then s_i and s_j will have the same core and will be merged

shift/reduce Conflict in LALR Parser

- shift action depends only on core I
 - $[A \rightarrow \alpha \bullet a \beta, b]$, both LR and LALR will shift on a
- reduce action depends on both: core and lookahead
 - $[A \rightarrow \alpha \bullet, b]$
- Merging states as above does not introduce shift-reduce conflict unless there was one before
- Proof:

shift/reduce Conflict in LALR Parser

- Proof: Assume there is a **shift-reduce** conflict in some state of the LALR parser with items
 - $\{[A \rightarrow \alpha\bullet, a], [A \rightarrow \gamma\bullet a\beta, b]\}$
 - Action on **a** for this item $[A \rightarrow \alpha\bullet, a]$ is reduce
 - Action on **a** for this item $[A \rightarrow \gamma\bullet a\beta, b]$ is shift
- Then **some** set of items from which the union was formed has item $[A \rightarrow \alpha\bullet, a]$, and since the cores of all these states are the same, it must have an item $[A \rightarrow \gamma\bullet a\beta, c]$ for some **c**
- But then this state has the same **shift/reduce** conflict on **a**, and the grammar was not LR(1) as we assumed

Conflicts in LALR Parser

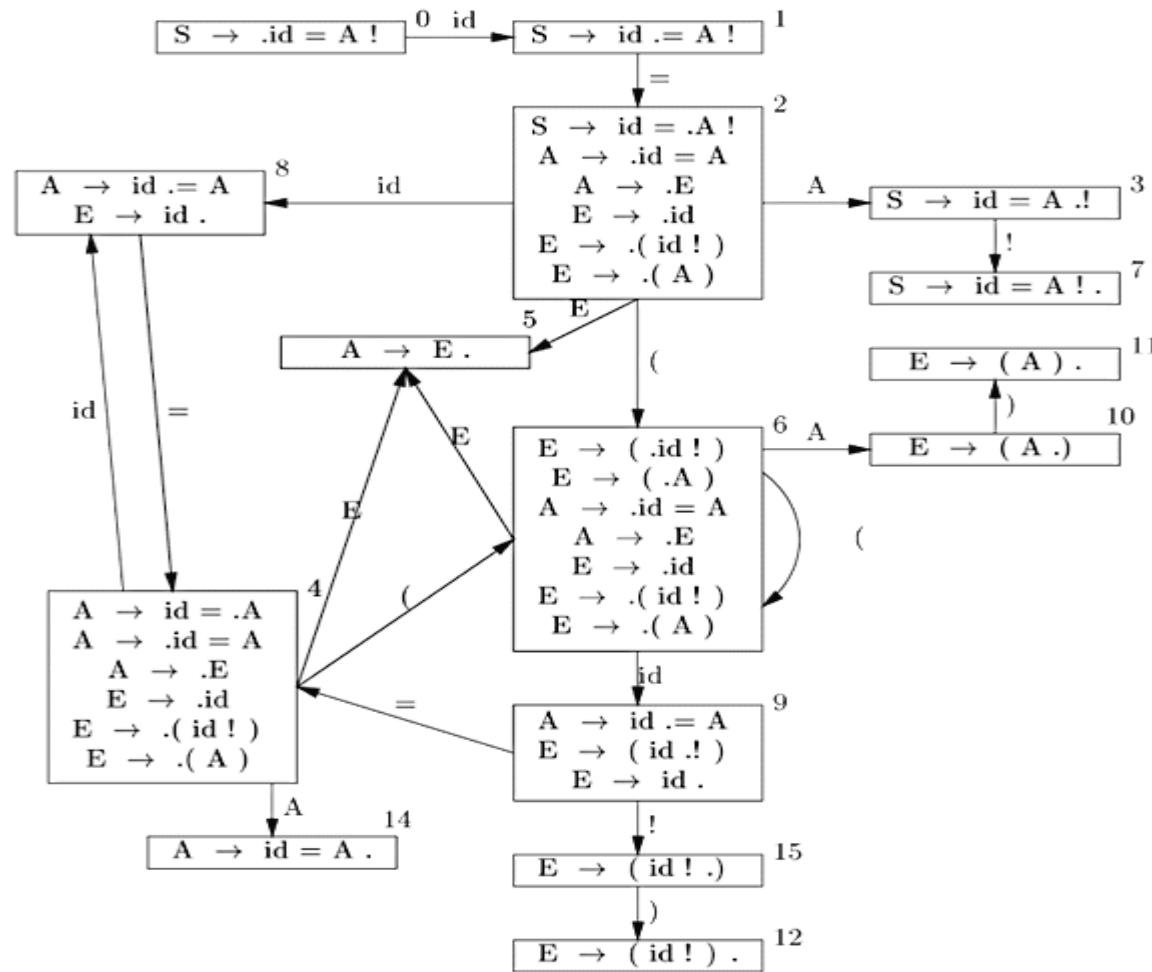
- New **reduce-reduce** conflict are possible
- Consider the LR(1) grammar G , $L(G) = \{acd, ace, bcd, bce\}$
 - $S' \rightarrow S$ $S \rightarrow aAd \mid bBd \mid aBe \mid bAe$
 - $A \rightarrow c$ $B \rightarrow c$
- The following two states are in LR(1)
 - $s_1 = \{[A \rightarrow c\bullet,d], [B \rightarrow c\bullet,e]\}$
 - $s_2 = \{[A \rightarrow c\bullet,e], [B \rightarrow c\bullet,d]\}$
 - Neither of these generates a conflict, and their cores are the same
- Merging them generates a reduce/reduce conflict:
 - $\{[A \rightarrow c\bullet,d/e], [B \rightarrow c\bullet,d/e]\}$
 - Since reductions by both $A \rightarrow c$ and $B \rightarrow c$ are called for on inputs d and e

Example

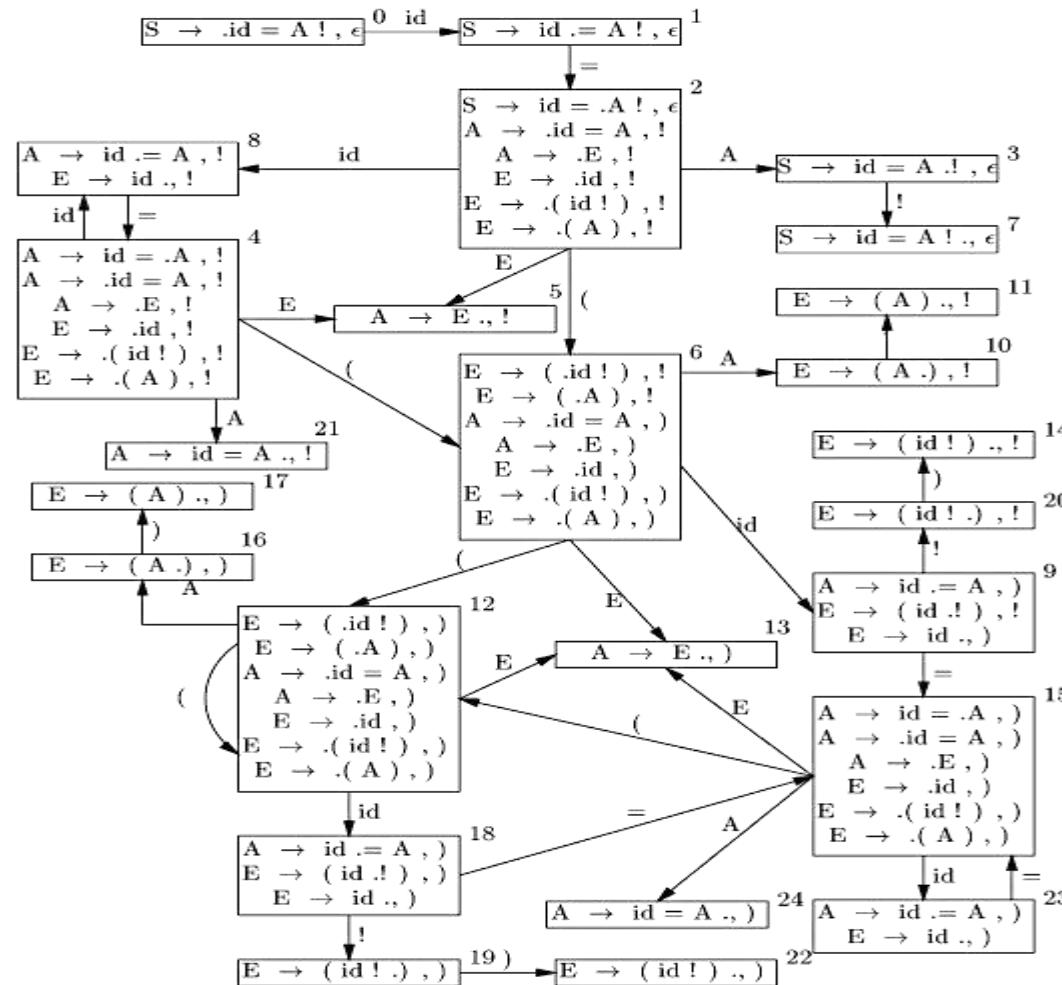
- Consider the following grammer:
 - $S \rightarrow id = A !$ 1
 - $A \rightarrow id = A$ 2
 - $A \rightarrow E$ 3
 - $E \rightarrow id$ 4
 - $E \rightarrow (id !)$ 5
 - $E \rightarrow (A)$ 6

- Nonterminlsls = { S, A, E }
- Terminals = { $id, =, !, (,)$ }

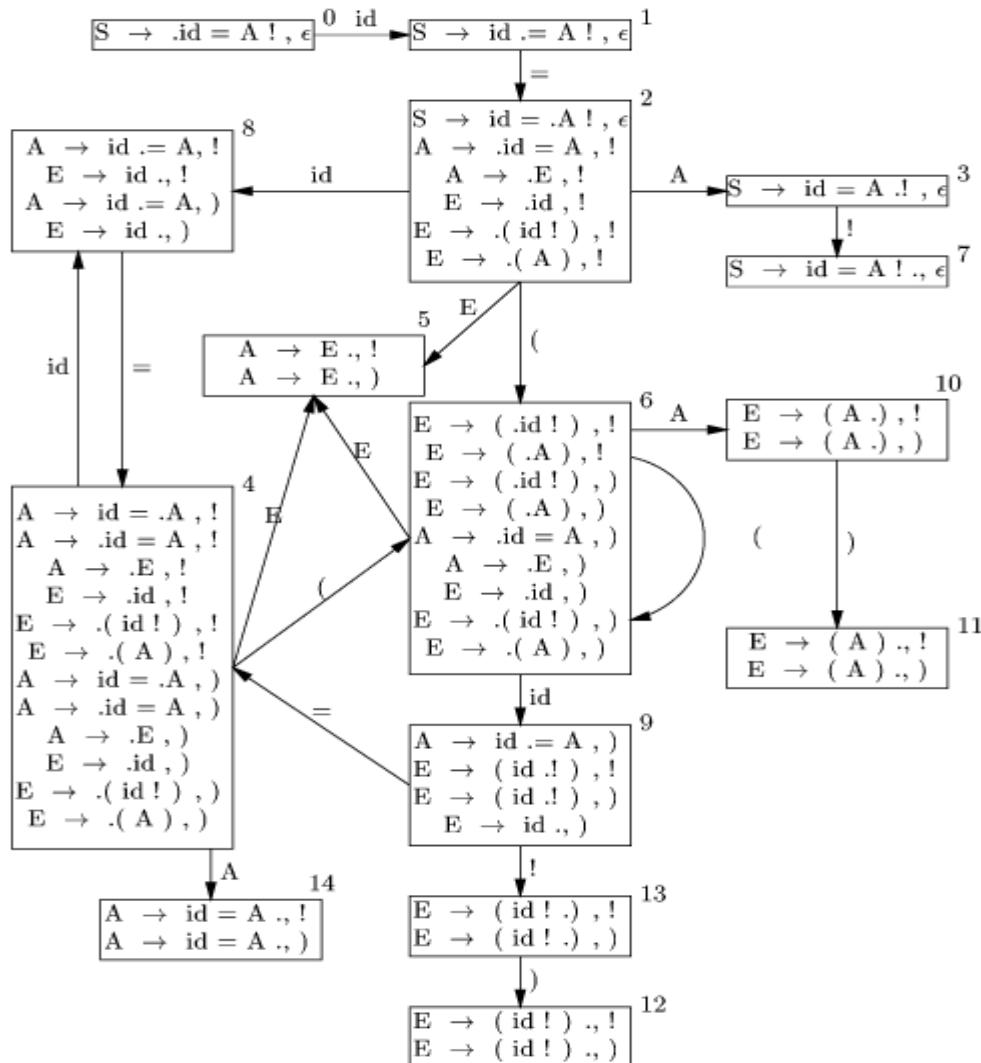
SLR(1)



LR(1)



LALR(1)



Compilers Construction

Chapter 6

Semantic Analysis

Dr. Mohammed Salem Atoum

M.Atoum@inu.edu.jo

Motivation and Definition

- Syntax-directed definitions extend context-free grammars by including attributes for each grammar symbol and semantic rules that specify semantic actions to be performed during parsing
 - Each production rule of the CFG has a set of semantic rules
 - Each grammar symbol has a set of attributes (such as data type, value, memory location, etc) defined by semantic rules

Attributes

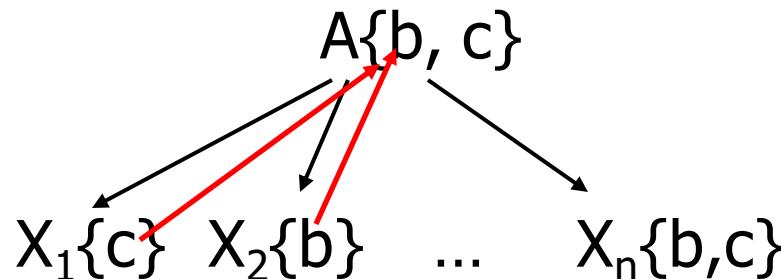
- Each grammar symbol $X \in T \cup NT$ has a set of **attributes**
 - $\text{Attr}(X) = \{X.a, X.b, \dots\}$
 - $\text{Attr}(X_1 \dots X_n) = \text{Attr}(X_1) \cup \dots \cup \text{Attr}(X_n)$
- Each production $A \rightarrow \alpha$ has a set of **semantic rules**
 - $a_0 = f(a_1, \dots, a_m)$ where $a_i \in \text{Attr}(A\alpha)$
- Attributes can either be **inherited** or **synthesized**
 - Synthesized: $a_0 \in \text{Attr}(A)$
 - Inherited: $a_0 \in \text{Attr}(\alpha)$

Form of a Syntax-Directed Definition

- Each grammar production $A \rightarrow \alpha$ has associated with it a set of semantic rules
- Each semantic rule has the form, $b = f(c_1, \dots, c_k)$, where f is a function, and either:
 - b is a synthesized attribute of A and c_1, \dots, c_k are attributes belonging to the grammar symbols of the production, or
 - b is an inherited attribute of one of the rhs grammar symbols and c_1, \dots, c_k are attributes belonging to the grammar symbols of the production

Synthesized Attributes

- An **attribute** of a grammar symbol is **synthesized** if it is computed from the attributes of its **children** (rhs of production rule) in the parse tree



- $A \rightarrow X_1 X_2 \dots X_n$ is a production
- Attribute **A.b** is **synthesized** since it is computed from children
 - $A.b = X_1.c + X_2.b$

Synthesized Attributes

- This can be expressed by the following psuedocode for a recursive postorder attribute evaluator:

```
procedure PostEval (T: treenode)
begin
    for each child C of T do
        PostEval(C);
    compute all synthesized attributes of T;
end ;
```

Synthesized Attributes

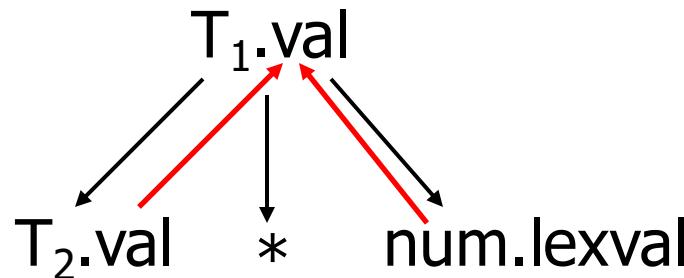
- Example:

- Production

□ $T_1 \rightarrow T_2 * \text{num}$

- Semantic Rules

$T_1.\text{val} = T_2.\text{val} * \text{num.lexval}$



- $T_1.\text{val}$ is a **synthesized** attribute
- num.lexval is a **synthesized** attribute whose value is assumed to be supplied by lexical analyzer (scanner)

Synthesized Attributes – Example2

Productions	Semantic Rules
$L \rightarrow E \text{ return}$	$\text{print}(E.\text{val})$
$E \rightarrow E + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} = T.\text{val}$
$T \rightarrow T * F$	$T.\text{val} = T_1.\text{val} * F.\text{val}$
$T \rightarrow F$	$T.\text{val} = F.\text{val}$
$F \rightarrow (E)$	$F.\text{val} = E.\text{val}$
$F \rightarrow \text{num}$	$F.\text{val} = \text{num}.\text{lexvalue}$

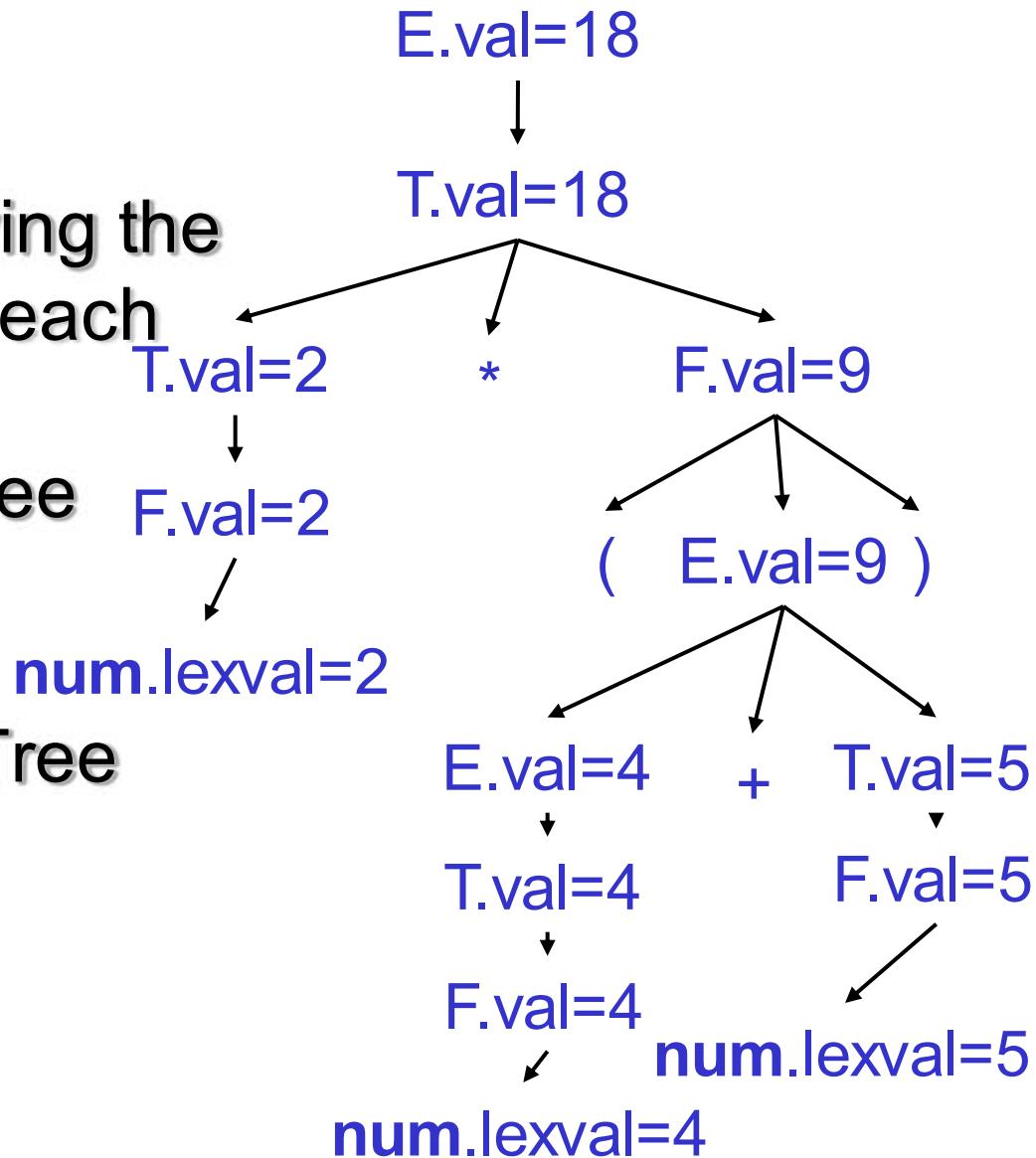
- $E.\text{val}$, $T.\text{val}$, $F.\text{val}$, and $\text{num}.\text{lexval}$ are synthesized attributes

Annotated Parse Tree

Input: $2 * (4 + 5)$

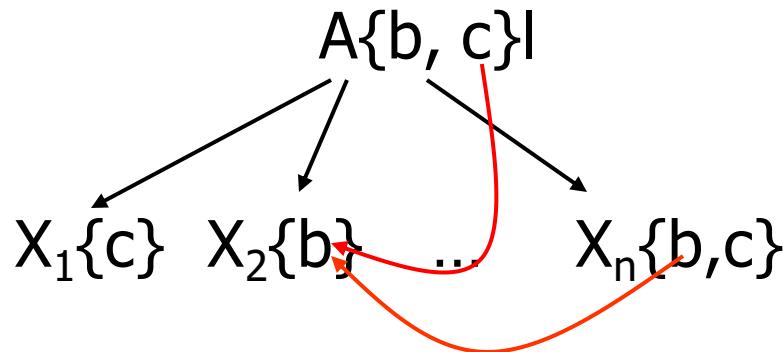
A Parse tree showing the attribute values at each node is called an **annotated** parse tree

Annotated Parse Tree



Inherited Attributes

- An attribute of a grammar symbol is **inherited**, if it is computed from the attributes of its **parents** (lhs of production rule) and/or **siblings**



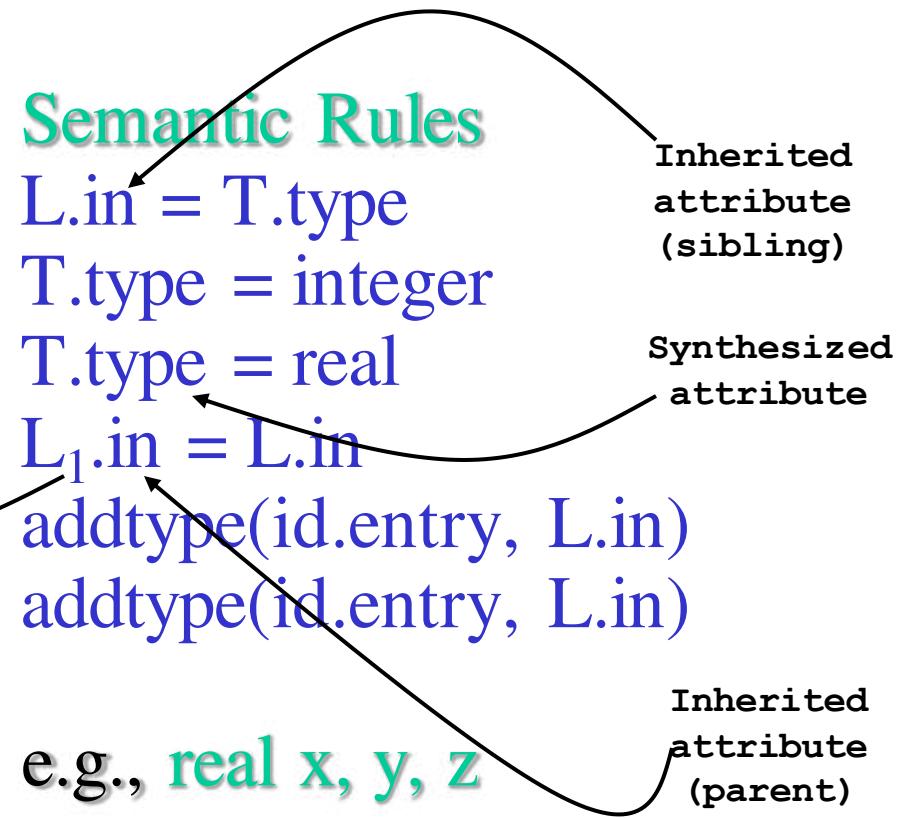
- Attribute $X_2.b$ is **inherited** since it is computed from both parent and siblings
 - $X_2.b = \text{conc}(A.c, X_n.b)$

Inherited Attributes

```
procedure PreEval (T :treenode)
begin
    for each child C of T do
        compute all inherited attributes of C;
        PreEval(C);
end;
```

Inherited Attributes - Example

- Example
- Production
 - $D \rightarrow T \ L$
 - $T \rightarrow \text{int}$
 - $T \rightarrow \text{real}$
 - $L \rightarrow L, \text{id}$
 - $L \rightarrow \text{id}$

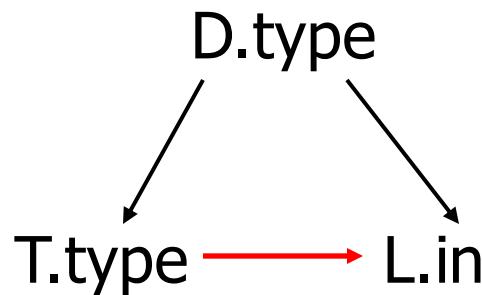


- Variable Declaration, e.g., **real x, y, z**
- **T.type** attribute is synthesized
- **L.in** attribute is inherited

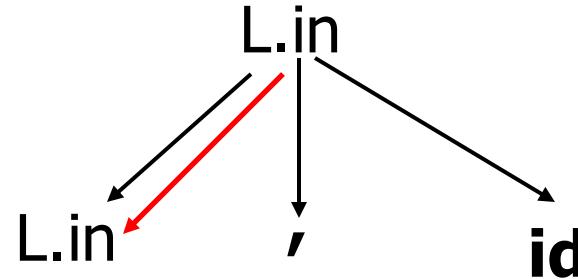
Inherited attributes

- $D \rightarrow T \ L$
- $L \rightarrow L, id$

$$\begin{aligned}L.in &= T.type \\L1.in &= L.in\end{aligned}$$



Inherited from sibling

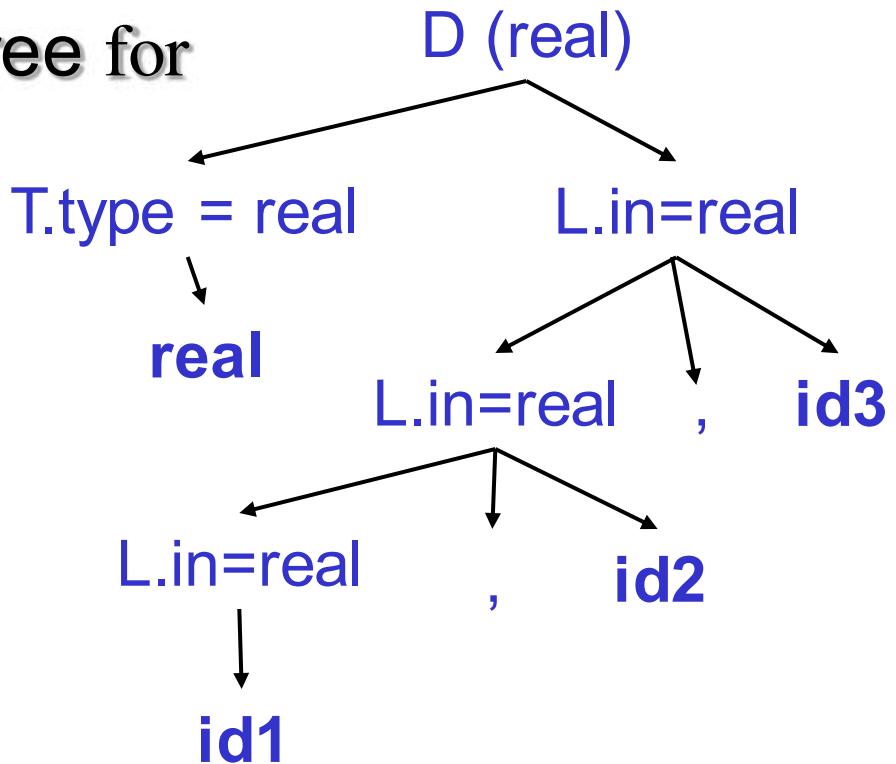


Inherited from parent

Annotated Parse Tree

- annotated parse tree for
 - real id1, id2, id3

- $D \rightarrow T\ L$
- $T \rightarrow \text{int}$
- $T \rightarrow \text{real}$
- $L \rightarrow L, \text{id}$
- $L \rightarrow \text{id}$



Attributes - Contd

- **Terminals** have attributes only (**no** semantic rules)
- **Terminals** are assumed to have **no** inherited attributes
- The **synthesized attribute** values for terminals are provided by the **lexical analyzer**.
- **Nonterminals** have both **attributes** and **semantic rules**

Attributes - Contd

- S-Attributed Definition
 - An S-attributed definition is a **syntax directed definition** that contains only **Synthesized** attributes
- L-Attributed Definition
 - Inherited attributes depends only on attributed to the **Left**

Example 2: Compute the type of an expression

$E \rightarrow E + E$

if ((E_2 .trans == INT) and (E_3 .trans == INT)
then E_1 .trans = INT

$E \rightarrow E \text{ and } E$

if ((E_2 .trans == BOOL) and (E_3 .trans == BOOL)
then E_1 .trans = BOOL
else E_1 .trans = ERROR

$E \rightarrow E == E$

if (E_2 .trans == E_3 .trans)
then E_1 .trans = BOOL
else E_1 .trans = ERROR

$E \rightarrow \text{true}$

E .trans = BOOL

$E \rightarrow \text{false}$

E .trans = BOOL

$E \rightarrow \text{int}$

E .trans = INT

$E \rightarrow (E)$

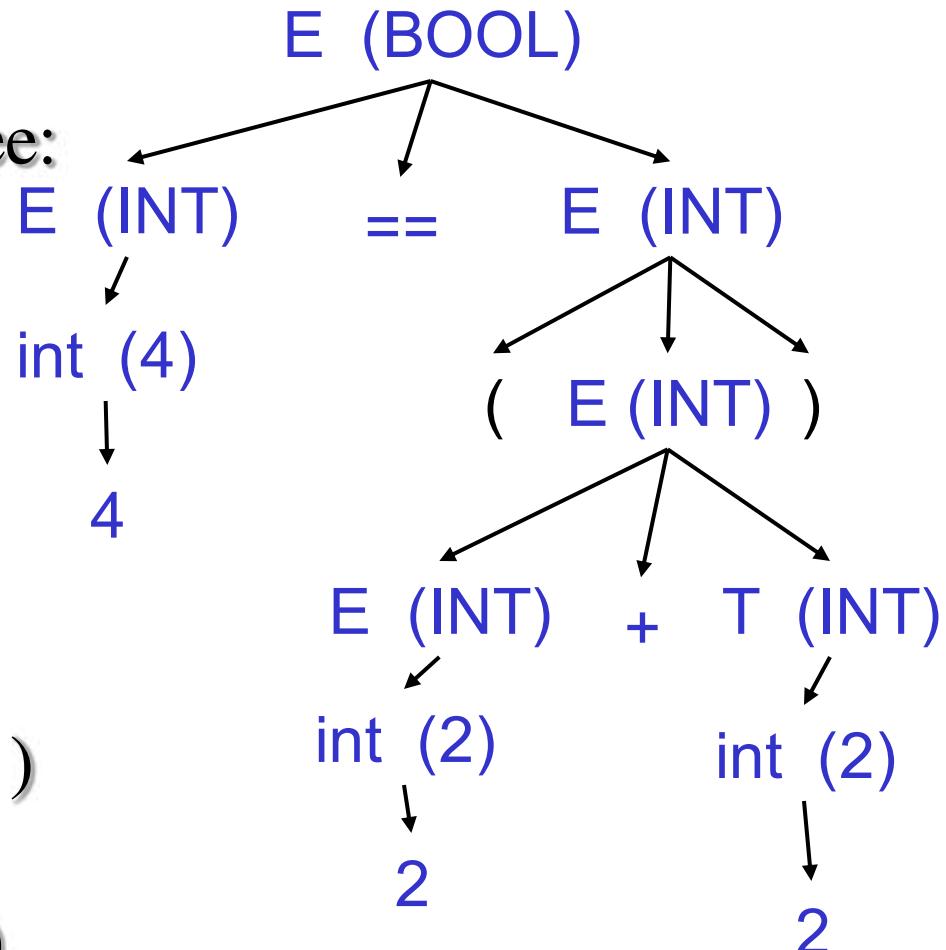
E_1 .trans = E_2 .trans

E .trans is synthesized attribute

Example 2 (cont)

- Input: $4 == (2+2)$
- Annotation parse tree:

$\begin{array}{l} \text{E} \rightarrow E == E \\ \rightarrow \text{int} == E \\ \rightarrow 4 == E \\ \rightarrow 4 == (\text{E}) \\ \rightarrow 4 == (\text{E+E}) \\ \rightarrow 4 == (\text{int + E}) \\ \rightarrow 4 == (2 + \text{E}) \\ \rightarrow 4 == (2 + \text{int}) \\ \rightarrow 4 == (2+2) \end{array}$



TEST YOURSELF

- A CFG for the language of binary numbers:

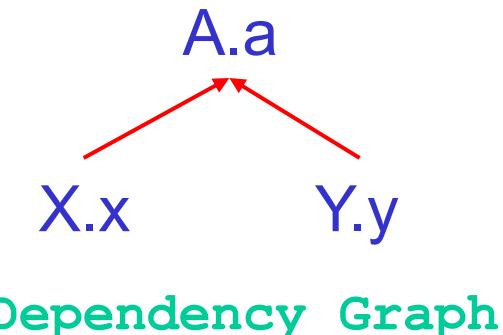
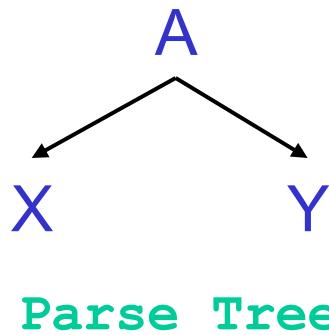
$B \rightarrow 0$	$B.\text{val} = 0$
$B \rightarrow 1$	$B.\text{val} = 1$
$B \rightarrow B\ 0$???
$B \rightarrow B\ 1$???
- Define a syntax-directed translation (definition) so that the translation of a binary number is its base-10 value
- Draw the parse tree for 1001 and **annotate** each nonterminal with its translation

Evaluation Order

- Strategy:
 - Evaluate only rules for which all arguments are already evaluated
- Dependency Graph:
 - Create a Node for each Attribute of each Parse Tree Node
 - Insert Edge $c_i \rightarrow b$ for all rules $b = f(\dots, c_i, \dots)$
- Topological Sort
 - If an attribute b at a node in a parse tree depends on an attribute c , then the semantic rule of b at that node must be evaluated after the rule that defines c
 - Ordering of Nodes (i.e. Attributes): a_1, \dots, a_n
 - Evaluate rule for a_1 , then a_2, \dots

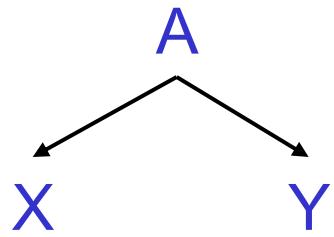
Dependency Graph

- Dependences between attributes:
 - Synthesized attributes: computed from children
 - Inherited attributes: computed from parent and/or siblings
- Suppose $A.a = f(X.x, Y.y)$ is a semantic rule for the production $A \rightarrow XY$
 - $A.a$ is synthesized attribute
 - Dependency Graph is

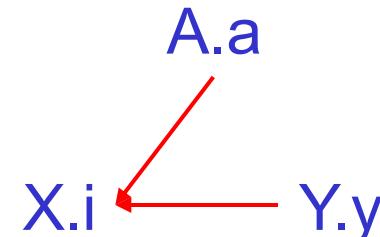


Dependency Graph

- Suppose $X.i = g(A.a, Y.y)$ is a semantic rule for the production $A \rightarrow XY$
 - $X.i$ is inherited attribute
 - Dependency Graph is



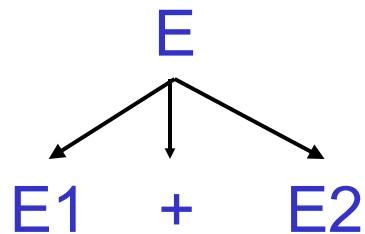
Parse Tree



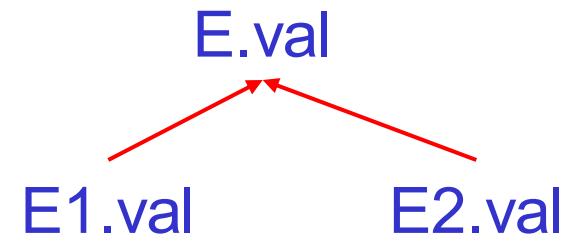
Dependency Graph

Dependency Graph: Example1

- Production Semantic Rule
 - $E \rightarrow E_1 + E_2$ $E.\text{val} := E_1.\text{val} + E_2.\text{val}$



Parse Tree



Dependency Graph

Dependency Graph: Example2

○ Production

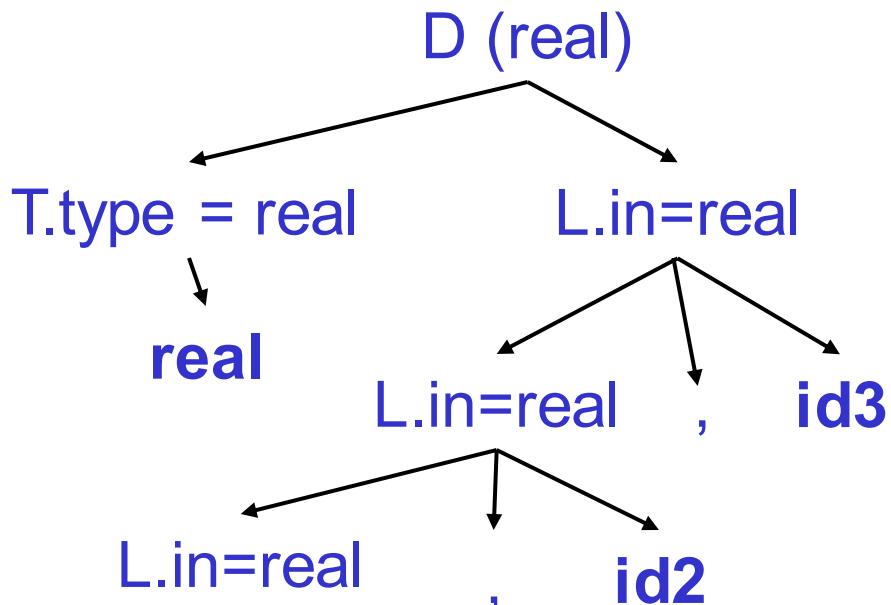
- $D \rightarrow T\ L$
- $T \rightarrow \text{int}$
- $T \rightarrow \text{real}$
- $L \rightarrow L, \text{id}$
- $L \rightarrow \text{id}$

Semantic Rules

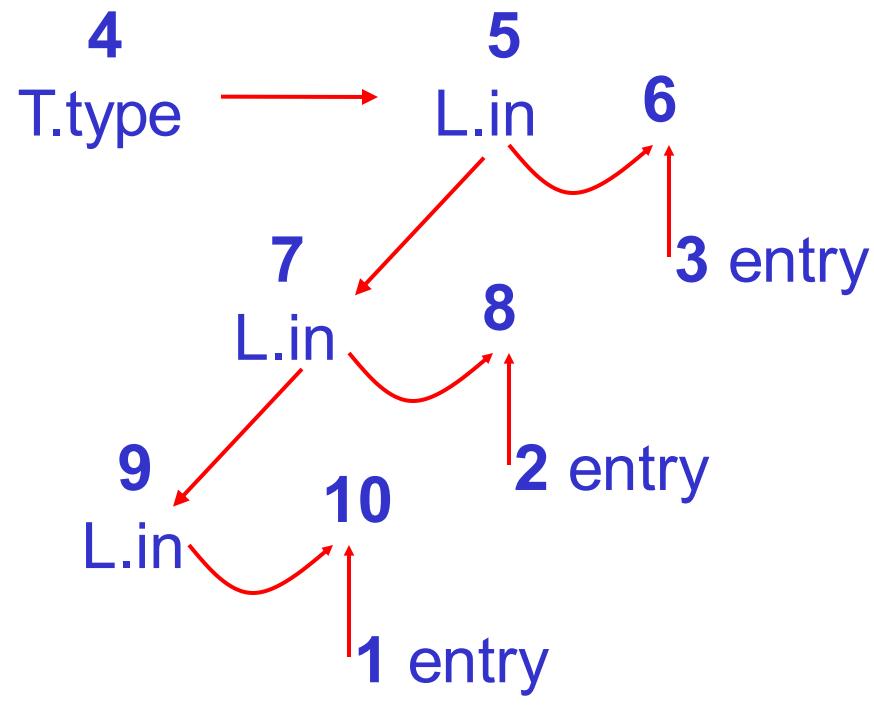
- $L.\text{in} = T.\text{type}$
- $T.\text{type} = \text{integer}$
- $T.\text{type} = \text{real}$
- $L_1.\text{in} = L.\text{in}$
- $\text{addtype(id.entry, L.in)}$
- $\text{addtype(id.entry, L.in)}$

Dependency Graph: Example2

- real id1, id2, id3



Parse Tree



Dependency Graph

Dependency Graph: Example2

- Note the following regarding this graph:
 - There is an edge for $L.in$ from node 4 to node 5 for $T.type$. This is because the inherited attribute $L.in$ depends on the attribute $T.type$ according to the semantic rule $L.in = T.type$ for $D \rightarrow T L$
 - The two downward edges into nodes 7 and 9 arise because $L1.in$ depends on $L.in$ according to the semantic rule $L1.in = L.in$ for the production $L \rightarrow L1, id$
 - Each of the semantic rules $addtype(id.entry, L,in)$ associated with the L -productions leads to the creation of a dummy attribute. Node 6, 8, 10 are constructed for these dummy attributes

Evaluation Order

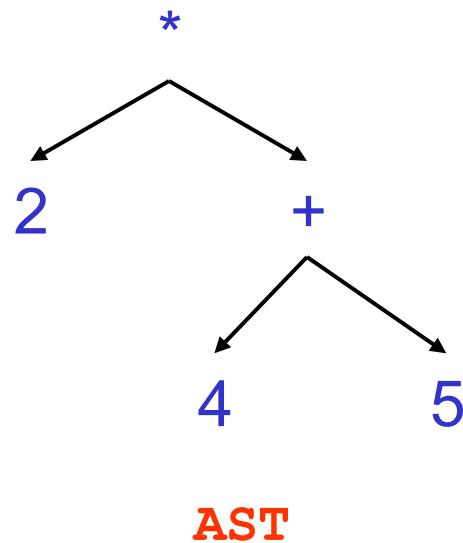
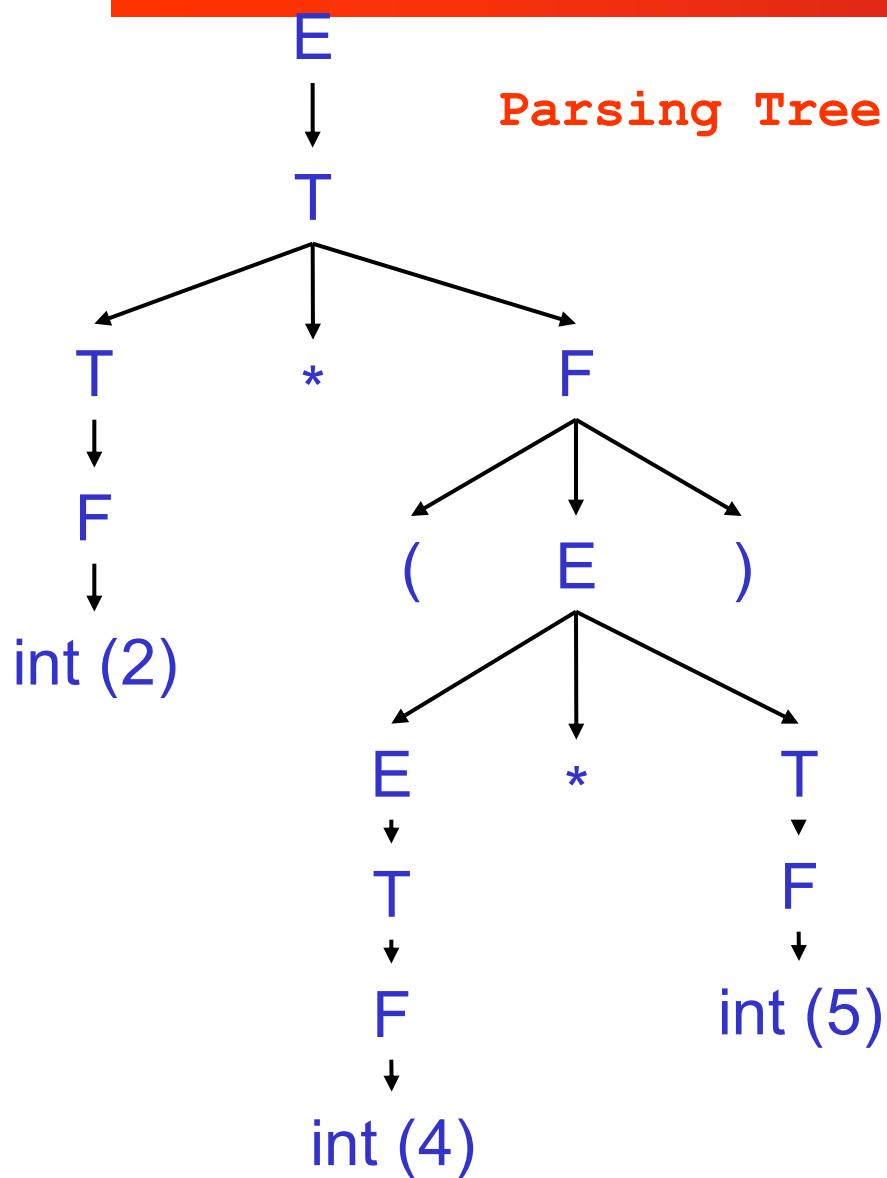
- A **topological sort** of a directed acyclic graph is any ordering m_1, m_2, \dots, m_k of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes; that is, if $m_i \rightarrow m_j$ is an edge from m_i to m_j , then m_i appears before m_j in the ordering
- Here is the evaluation order of the dependency graph in the previous slide (a_n is an attribute associated with the node numbered n in the dependency graph)
 - $a4 := \text{real};$
 - $a5 := a4;$
 - $\text{Addtype}(id_3.\text{entry}, a5)$
 - $a7 := a5;$
 - $\text{Addtype}(id_2.\text{entry}, a7)$
 - $a9 := a7;$
 - $\text{Addtype}(id_1.\text{entry}, a9)$

AST vs Parse Tree

- An (Abstract) Syntax Tree (AST) is condensed form of parse tree
 - operators and keywords do not appear as leaves, but rather are associated with the *internal* nodes,
 - Syntactic details are omitted
 - e.g., parentheses, commas, semi-colons
- AST is a better structure for later compiler stages
 - omits details having to do with the source language
 - only contains information about the essential structure of the program

Example: $2 * (4 + 5)$

parse tree vs AST



Constructing Syntax Tree for Expressions

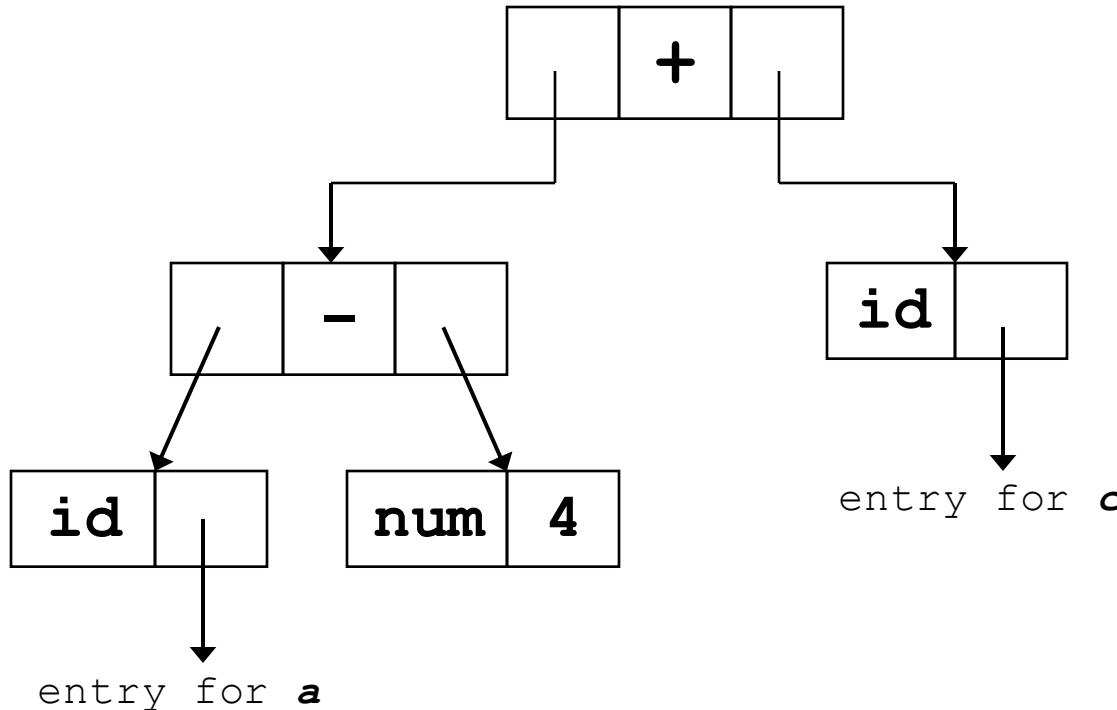
- Each node in a syntax tree can be implemented as a record with several fields
- In the node for an operator, one field identifies the operator, and the remaining fields contain pointers to the nodes for the operands
- The operator is often called the *label* of the node

Constructing Syntax Tree for Expressions

- Functions used to create the nodes of a syntax tree in a syntax-directed definition, are:
 - `mknode(op, left, right)`: create an operator node with label *op* and two fields containing pointers to *left* and *right*
 - `mkleaf(id, entry)`: creates an identifier node with label *id* and a field containing *entry*, a pointer to the symbol-table entry for the identifier
 - `mkleaf(num, val)`: creates a number node with label *num* and a field containing *val*, the value of the number

Constructing Syntax Tree for Expressions

- The syntax tree for $a-4+c$ is:
 - The tree is constructed button-up



Syntax tree for $a-4+c$

Constructing Syntax Tree for Expressions

- The sequence of function calls for creating the syntax tree for **a-4+c** is:
 - $p1 := \text{mkleaf}(\text{id}, \text{entrya})$
 - $p2 := \text{mkleaf}(\text{num}, 4)$
 - $p3 := \text{mknod}(\text{'-'}, p1, p2)$
 - $p4 := \text{mkleaf}(\text{id}, \text{entryc})$
 - $p5 := \text{mknod}(\text{'+'}, p3, p4)$

A Syntax-Directed Definition for Constructing Syntax Trees

Production

$E \rightarrow E_1 + T$

$E \rightarrow E_1 - T$

$E \rightarrow T$

$T \rightarrow (E)$

$T \rightarrow id$

$T \rightarrow int$

Semantic Rule

$E.nptr = mknode('+', E_1.nptr, T.nptr)$

$E.nptr = mknode('-', E_1.nptr, T.nptr)$

$E.nptr = T.nptr$

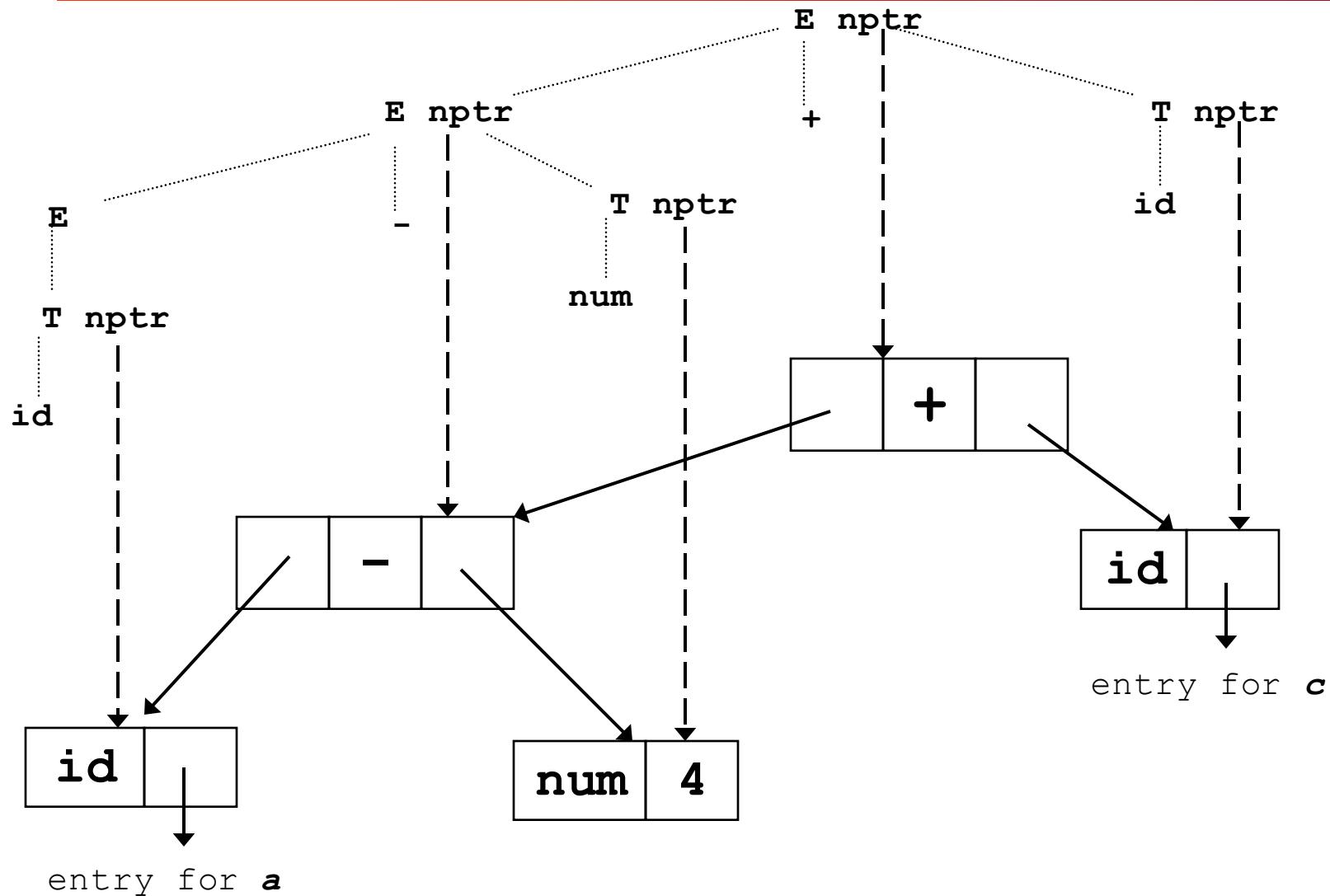
$T.nptr = E.nptr$

$T.nptr = mkleaf(id, id.entry)$

$T.nptr = mkleaf(num, num.val)$

- Attributes **id.entry** and **num.val** are the lexical values assumed to be returned by the lexical analyzer

Construction of a Syntax tree for $a-4+c$

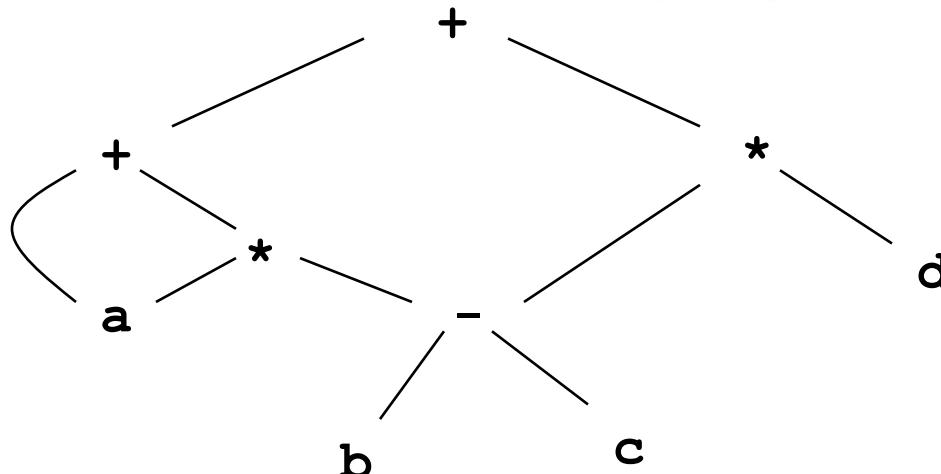


Directed Acyclic Graph for Expressions

- Like a syntax tree, a directed acyclic graph (**dag**) has a node for every subexpression of the expression
- An interior node represents an operator and its children represent its operands
- The **difference** is that a node in a dag representing a common subexpression **has more than one “parent”** in a syntax tree
- The common subexpression would be represented as a duplicated subtree

Directed Acyclic Graph for Expressions

- The following Figure contains a dag for the expression: $a + a * (b - c) + (b - c) * d$



- The leaf for a has b two parents because a is common to the b two subexpression a and $a*(b-c)$.
- Likewise, both occurrences of the common subexpression $b-c$ are represented by the same node, which also has two parents

Directed Acyclic Graph for Expressions

- The following syntax-directed definition can be modified to construct a dag instead of a syntax tree
- **Production** **Semantic Rule**

$E \rightarrow E_1 + T$

$E.\text{nptr} = \text{mknnode}('+' , E_1.\text{nptr}, T.\text{nptr})$

$E \rightarrow E_1 - T$

$E.\text{nptr} = \text{mknnode}('-' , E_1.\text{nptr}, T.\text{nptr})$

$E \rightarrow T$

$E.\text{nptr} = T.\text{nptr}$

$T \rightarrow (E)$

$T.\text{nptr} = E.\text{nptr}$

$T \rightarrow \text{id}$

$T.\text{nptr} = \text{mkleaf}(\text{id}, \text{id}.entry)$

$T \rightarrow \text{int}$

$T.\text{nptr} = \text{mkleaf}(\text{num}, \text{num}.val)$

- A dag is obtained if the function (**mknnode** or **mkleaf**) constructing a node first checks to see whether an identical node already exists
- If so, the function can return a pointer to the previously constructed one

Directed Acyclic Graph for Expressions

- The following are the sequence of instructions that construct the dag of $a + a * (b - c) + (b - c) * d$

$p_1 := \text{mkleaf}(\text{id}, a)$

$p_2 := \text{mkleaf}(\text{id}, a)$

$p_3 := \text{mkleaf}(\text{id}, b)$

$p_4 := \text{mkleaf}(\text{id}, c)$

$p_5 := \text{mknnode}(' - ', p_3, p_4)$

$p_6 := \text{mknnode}(' * ', p_2, p_5)$

$p_7 := \text{mknnode}(' + ', p_1, p_6)$

$p_8 := \text{mkleaf}(\text{id}, b)$

$p_9 := \text{mkleaf}(\text{id}, c)$

$p_{10} := \text{mknnode}(' - ', p_8, p_9)$

$p_{11} := \text{mkleaf}(\text{id}, d)$

$p_{12} := \text{mknnode}(' * ', p_{10}, p_{11})$

$p_{13} := \text{mknnode}(' - ', p_7, p_{12})$

- $p_1 = p_2$

$$p_3 = p_8$$

- $p_4 = p_9$

$$p_5 = p_{10}$$

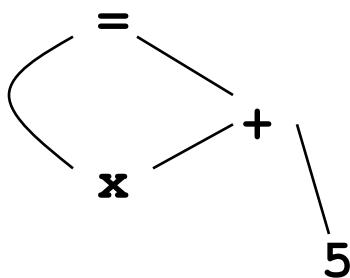
Directed Acyclic Graph for Expressions

- Often nodes in a tree are implemented as records stored in an array

Assignment

$x = x + 5$

DAG



Representation

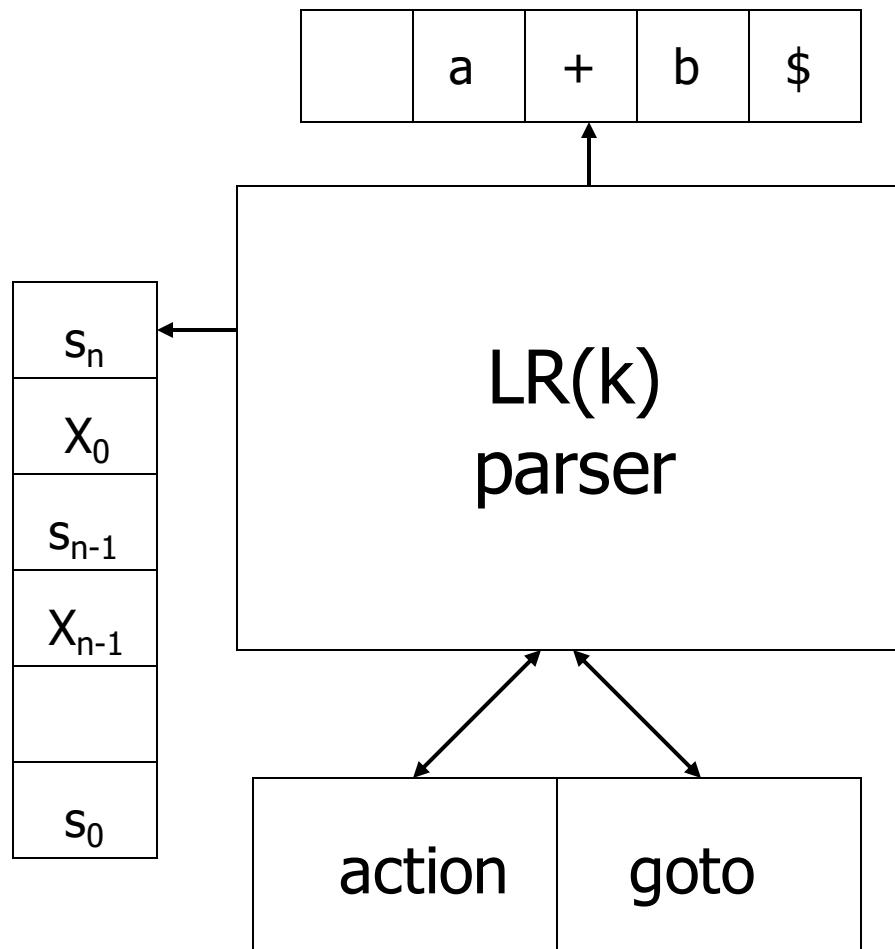
1	id			entry for x	
2	num	5			
3	+	1	2		
4	=	1	3		
5		...			

- We refer to a node by its **index** in the array
- The integer **index** of a node is often called a **value number** for historical reasons

Bottom-Up Evaluation of S-Attributed Definitions

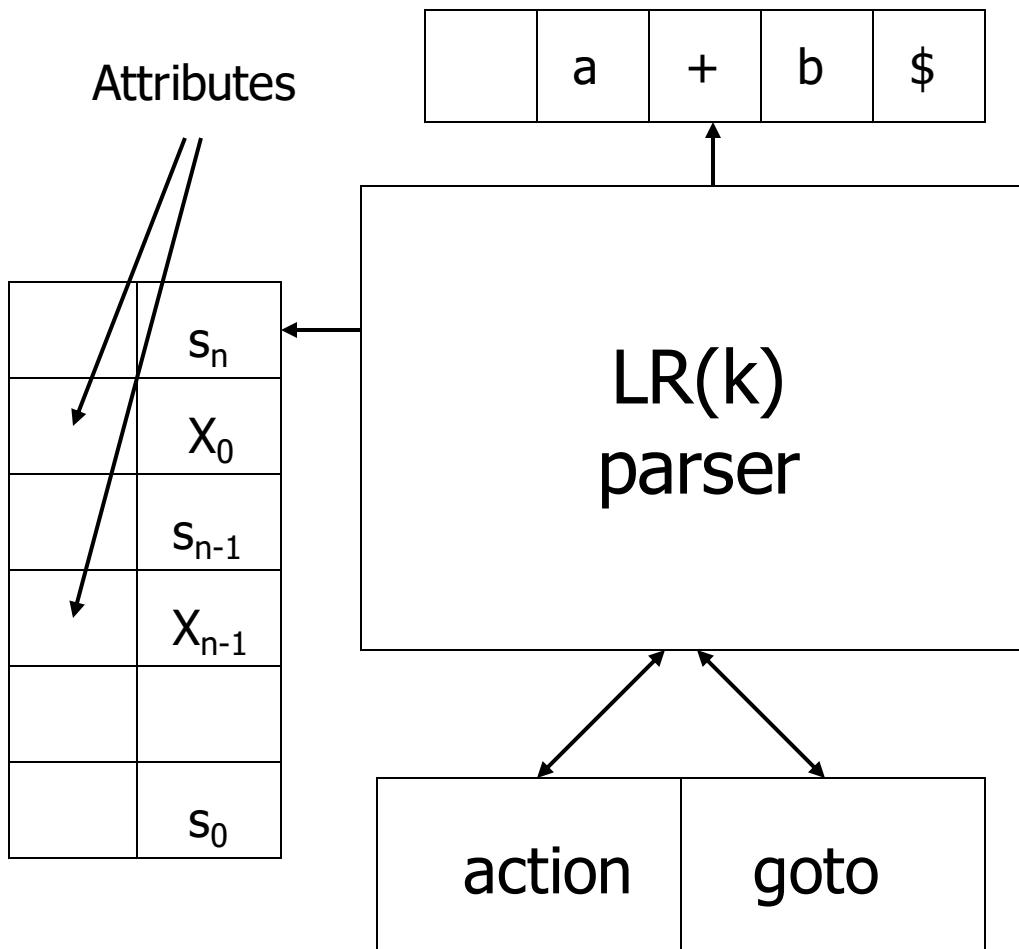
- S-Attributed grammar:
 - contains only synthesized attributes
 - can be evaluated by bottom-up parser during the parsing of input (bottom-up traversal)
 - Parser evaluates attributes as it parses the input
 - use stack information processing
 - Extra fields are added to the LR stack to hold the values of synthesized attributes

LR parser



- Given the current state on top and current token, consult the action table.
- Either, shift, i.e., read a new token, put in stack, and push new state, or
- Reduce, i.e., remove some elements from stack, and given the newly exposed top of stack and current token, to be put on top of stack, consult the goto table about new state on top of stack.

LR parser adapted for S-Attributed Grammars



Same as before, plus:

- Whenever reduce step, **execute the action associated with grammar rule.**
- Can put record of attributes, associated with a grammar symbol, on stack.

Bottom-Up Evaluation of S-Attributed Definitions

- If the i -th state symbol is A , then $val[i]$ will hold the value of the attribute associated with A
- The current top of the stack is indicated by top
- We assume that **synthesized attributes** are evaluated just before each reduction

top →

state	val
Z	Z . z
Y	Y . y
X	X . x
...	...

Bottom-Up Evaluation of S-Attributed Definitions

- Suppose the semantic rule $A.a = f(X.x, Y.y, Z.z)$ is associated with the production $A \rightarrow XYZ$
- Before XYZ is reduced to A :
 - the value of the attribute $Z.z$ is in $\text{val}[\text{top}]$
 - the value of the attribute $Y.y$ is in $\text{val}[\text{top-1}]$
 - the value of the attribute $X.x$ is in $\text{val}[\text{top-2}]$
- After reduction:
 - $\text{top} = \text{top} - 2$
 - $\text{state}[\text{top}] = A$
 - $\text{val}[\text{top}] = A.a$

top →

state	val
Z	Z . z
Y	Y . y
X	X . x
...	...

Bottom-Up Evaluation of S-Attributed Definitions

	state	val
top→	Z	Z . z
	Y	Y . y
	X	X . x

before reduction

	state	val
top→	A	A . a

after reduction

Example

Consider the following Syntax-directed definition

Production	Semantic Rule	Code Fragment
$L \rightarrow E\ n$	$\text{print}(E.\text{val})$	$\text{print}(\text{val}[top])$
$E \rightarrow E + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$	$\text{val}[ntop] = \text{val}[top-2] + \text{val}[top]$
$E \rightarrow T$	$E.\text{val} = T.\text{val}$	
$T \rightarrow T * F$	$T.\text{val} = T_1.\text{val} * F.\text{val}$	$\text{val}[ntop] = \text{val}[top-2] * \text{val}[top]$
$T \rightarrow F$	$T.\text{val} = F.\text{val}$	
$F \rightarrow (E)$	$F.\text{val} = E.\text{val}$	$\text{val}[ntop] = \text{val}[top-1]$
$F \rightarrow \text{digit}$	$F.\text{val} = \text{digit}.\text{lexvalue}$	

oval and lexval are synthesized attributes

Example:

$3*5+4n$

<i>Input</i>	<i>state</i>	<i>val</i>	<i>Prod. Rule</i>
$3*5+4n$	-	-	
$*5+4n$	3	3	
$*5+4n$	F	3	$F \rightarrow \text{digit}$
$*5+4n$	T	3	$T \rightarrow F$
$5+4n$	T^*	3 -	
$+4n$	T^*5	3 - 5	
$+4n$	T^*F	3 - 5	$F \rightarrow \text{digit}$
$+4n$	T	15	$F \rightarrow T^*F$
$+4n$	E	15	$E \rightarrow T$
$4n$	$E+$	15 -	
n	$E+4$	15 - 4	
n	$E+F$	15 - 4	$F \rightarrow \text{digit}$
n	$E+T$	15 - 4	$T \rightarrow F$
n	E	19	$E \rightarrow E+T$
	E n	19 -	
	L	19	$L \rightarrow E \ n$

L-Attributed Definition

- contain both **synthesized** and **inherited** attributes
- all **inherited** attributes in production are functions only of symbols to their **left** in the production
- For example for general production
 - $X \rightarrow Y_1 Y_2 \dots Y_n$
 - the inherited attributes of Y_k depends only on the attributes of X and $Y_1 Y_2 \dots Y_{k-1}$ can be evaluated by a **left to right** traversal of the parse tree

L-Attributed Definition

- L-Attributed Definition
 - A syntax-directed definition is L-Attributed if each inherited attribute of X_j , $1 \leq j \leq n$, on the right side of $A \rightarrow X_1 X_2 \dots X_n$, depends only on
 - the attribute of the symbols X_1, X_2, \dots, X_{j-1} to the left of X_j in the production, and
 - the inherited attributes of A
- Every S-attributed definition is L-attributed, since the above rules apply only to *inherited* attributes

Example

Production	Semantic Rule
$A \rightarrow L M$	$L.i = l(A.i)$
	$M.i = m(L.s)$
	$A.s = f(M.s)$
$A \rightarrow Q R$	$R.i = r(A.i)$
	$Q.i = q(R.s)$
	$A.s = f(Q.s)$

- i is inherited attribute, while s is synthesized
- The above syntax-directed definition is **NOT L-attributed**
 - The **inherited** attribute $Q.i$ in the rule $Q.i = q(R.s)$ depends on the attribute $R.s$ of R to its **right**

Translation Schema

- Is a CFG in which
 - attributes are associated with the grammar symbols and
 - semantic actions enclosed between braces {} are inserted within the right side of production
- Translation Schemas may have both synthesized and inherited attributes

Example

- Consider the following translation schema to map *infix* expressions with only + and – into corresponding *postfix* expressions

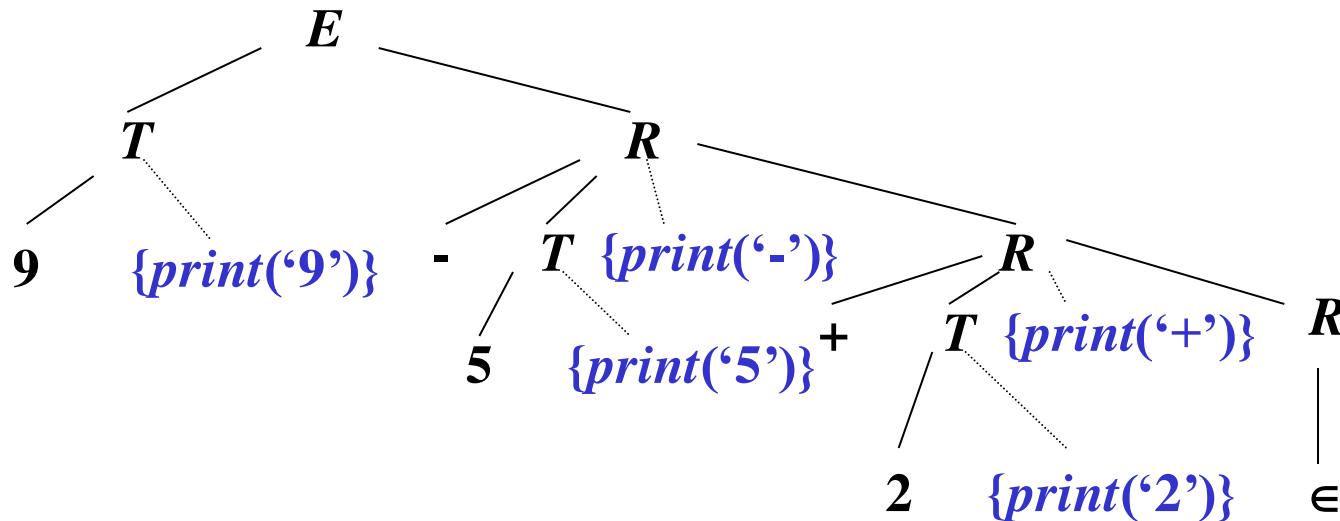
$E \rightarrow T\ R$

$R \rightarrow \text{addop}\ T\ \{\text{print}(\text{addop.lexeme})\}\ R\ |\ \in$
 $T \rightarrow \text{num}\ \{\text{print}(\text{num.val})\}$

- Note the following:
 - The non-terminal **addop** represents the terminals + or –
 - We treat the **semantic actions** (i.e., the print statements above) as though they are terminal symbols

Parse Tree for 9-5+2 showing actions

- When performed in **depth-first** order the **actions** print out the translated statements 95-2+



Restrictions when designing a Translation Schema

- Ensure that an attribute value is available when an action refers to it (i.e., the action does not refer to an attribute that has not yet been computed)
- The easiest case is when **only synthesized attributes** are needed. For this case, the **semantic action** will be the same as **semantic rule**, and placing this **action** at the end of the **right side** of the production.
- Example:

Production	Semantic Rule	Translation Scheme
$T \rightarrow T_1 * F$	$T.\text{val} \rightarrow T_1.\text{val} * F.\text{val}$	$T \rightarrow T_1 * F \{ T.\text{val} \rightarrow T_1.\text{val} * F.\text{val} \}$

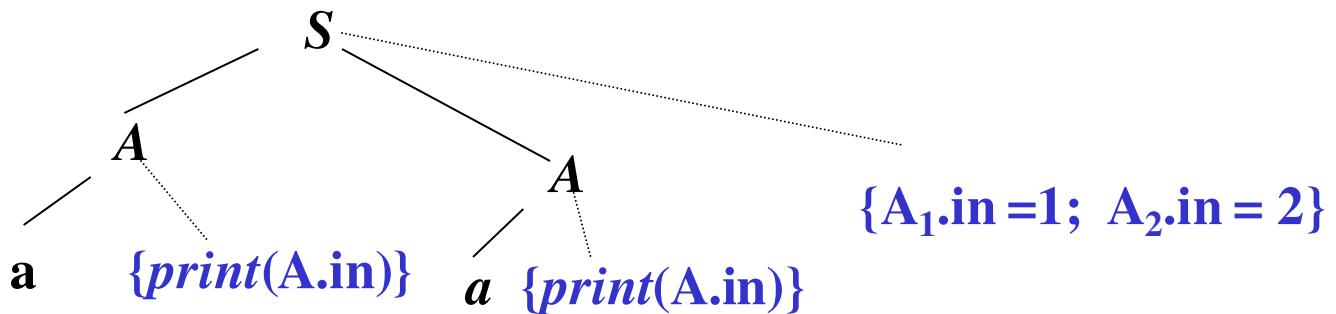
Restrictions when designing a Translation Schema

- If we have both **inherited** and **synthesized** attributes, we must be more careful
- 1. An **inherited attribute** for a **symbol** on the **right side** of a production must be computed in **an action before that symbol**
- 2. An **action** must **not** refer to a **synthesized attribute** of a symbol to the **right** of the **action**
- 3. A **synthesized attribute** for the **nonterminal on the left** can only be computed after all attributes it references have been computed. The action computing such attributes can usually be placed at the end of the right side of the production

Example of a Translation scheme that does not satisfy the first requirement

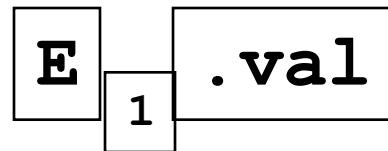
$S \rightarrow A_1 \ A_2 \ \{A_1.in = 1; \ A_2.in = 2\}$
 $A \rightarrow a \ \{print(A.in)\}$

- The **inherited** attribute $A.in$ in the **2nd production** is **not** defined when an attempt is made to print its value during a **depth-first** traversal of the parse tree for the input string ***aa***



Example

- We use an example from EQN language
- It is box language
- Examples:
 - $E \text{ sub } 1 . \text{val}$ results in $E_1.\text{val}$



- $a \text{ sub } \{i \text{ sup } 2\}$ results in a_{i^2}

Example: Syntax-directed Definition

<u>Production</u>	<u>Semantic Rules</u>
$S \rightarrow B$	$B.ps = 10$
$B \rightarrow B_1 B_2$	$S.ht = B.ht$ $B_1.ps = B.ps$ $B_2.ps = B.ps$ $B.ht = \max(B_1.ht, B_2.ht)$
$B \rightarrow B_1 \text{ sub } B_2$	$B_1.ps = B.ps$ $B_2.ps = \text{shrink}(B.ps)$ $B.ht = \text{disp}(B_1.ht, B_2.ht)$
$B \rightarrow \text{text}$	$B.ht = \text{text}.h * B.ps$

- **text** and **sub** are terminals
- We assume that the attribute **h** of **text** is obtained by table lookup, given the character represented by the token **text**

Example

- The non-terminal B (for Box) represents a math formula
- $\textcolor{teal}{B} \rightarrow \textcolor{teal}{B}_1 \textcolor{teal}{B}_2$ represents the juxtaposition of two boxes, and when it is applied:
 - $\textcolor{teal}{B}_1$ and $\textcolor{teal}{B}_2$ inherit the point size from B
 - The height of $\textcolor{teal}{B}$, represented by the synthesized attribute ht , is the maximum of the heights of $\textcolor{teal}{B}_1$ and $\textcolor{teal}{B}_2$

Example

- $B \rightarrow B_1 \sub B_2$ represents the placement of the 2nd subscript box (i.e., B_2) in a smaller size than the 1st box (i.e., B_1), and when it is applied:
 - The function *shrink* lowers the point size of B_2 by 30%
 - The function *disp* allows for downward displacement of the box B_2 as it computes the height of B
- The inherited attribute *ps* (point size) affects the height of the formula
- The production $B \rightarrow \text{text}$ causes the normalized height of the text to be multiplied by the point size to get the actual height of the text

Example: Translation Schema

$S \rightarrow \{B.ps = 10\}$
 $B \{S.ht = B.ht\}$

$B \rightarrow \{B_1.ps = B.ps\}$
 $B_1 \{B_2.ps = B.ps\}$
 $B_2 \{B.ht = \max(B_1.ht, B_2.ht)\}$

$B \rightarrow \{B_1.ps = B.ps\}$
 B_1
sub $\{B_2.ps = \text{shrink}(B.ps)\}$
 $B_2 \{B.ht = \text{disp}(B_1.ht, B_2.ht)\}$

$B \rightarrow \text{text} \{B.ht = \text{text}.h * B.ps\}$

Example:

Translation Schema

$S \rightarrow \{B.ps = 10\} B \{S.ht = B.ht\}$

$B \rightarrow \{B_1.ps = B.ps\} B_1 \{B_2.ps = B.ps\} B_2 \{B.ht = \max(B_1.ht, B_2.ht)\}$

$B \rightarrow \{B_1.ps = B.ps\} B_1 \text{ sub } \{B_2.ps = \text{shrink}(B.ps)\}$
 $B_2 \{B.ht = \text{disp}(B_1.ht, B_2.ht)\}$

$B \rightarrow \text{text } \{B.ht = \text{text}.h * B.ps\}$

Example

- Note that the Syntax-directed definition is L-attributed:
 - The only inherited attribute is *ps* of the nonterminal **B**
 - Each semantic rule defines *ps* only in terms of the inherited attribute of the nonterminal on the left of the production
- The Translation Schema satisfied the three requirements given above

Compilers Construction

Chapter 6

Semantic Analysis

Dr. Mohammed Salem Atoum

M.Atoum@inu.edu.jo

Symbol Table

- The symbol table involved with the parser and the scanner, either of which may need to enter information directly into the symbol table or consult it to resolve ambiguities.
- Also involve with semantic analysis phase.

What are the entries in the symbol table?

- variable names
- procedure and function names
- defined constants
- labels
- keywords may also be entered if the lexical analyzer does not take care of them

What information should be kept?

- textual name
 - data type
 - for arrays dimension, upper and lower bounds for each dimension
 - for records or structures list of fields and their information
 - for functions and procedures number and type of arguments
 - storage information (memory location: base address and offset)
 - scope
-
- Entries are not uniform, different entries may require different information
 - Information is entered as it becomes available during different phases of the compilation

Basic symbol table interface

- Insert names (identifiers) and their attributes to the symbol table
 - *Insert(name, record)*
- Lookup names and their attributes
 - *LookUp(name)*
- The delete operation is needed to remove the information provided by a declaration when that declaration no longer applies.
 - Open/close scopes

The structure of the symbol table

- The symbol table in a compiler is a typical dictionary data structure.
- Typical implementations of dictionary structures include
 - linear lists
 - various search tree structures (binary, search trees, AVL trees, B trees)
 - hash tables

Linear lists

- A simple (not very efficient) implementation:
Unordered list
 - use an array or linked list
 - enter the names in the order they are encountered
 - Insert O(1) complexity if multiple entries are allowed. If we have to check that an entry was not entered before, then insert will be O(N)
 - LookUp O(N) time
 - Simple and compact but too slow in practice

Search tree structures

- are somewhat less useful for the symbol table, partially because they do not provide best-case efficiency, but also because of the complexity of the delete operation

Hash Tables

- Distribute the entries to S buckets using a hash function on the names
- Hash function should be efficient and should distribute the entries uniformly
- Open hashing: Create linked lists for the buckets which have collisions
 - Cost for lookup should be $O(N / S)$ assuming that computing the hash function takes constant time
 - If $S \approx N$, then cost of lookup is $O(1)$
- Open addressing: rehash when there is a collision
 - Requires more space since it stores every entry in the table, however rehash chains are short and the lookup is faster if the hash functions have good distributions

Lexically-scoped Symbol Tables

- The problem
 - The compiler needs a distinct record for each declaration
 - Nested lexical scopes admit duplicate declarations
 - Solution: store the scope information in the symbol table

Lexically-scoped Symbol Tables

- The interface

- Insert(name, level) – creates record for name at level
- Lookup(name, level) – returns pointer or index
- InitializeScope() – increments the current level and creates a new symbol table for that level
- FinalizeScope() – changes current level pointer so that it points at the table for the scope surrounding the current level, and then decrements the current level

Example in C

```
static int w;                  /* level 0 */
int x;
void example(a, b);
    int a, b                  /* level 1 */
{
    int c;
    {
        int b, z;            /* level 2a */
        ...
    }
    {
        int a, x;            /* level 2b */
        ...
        {
            int c, x;        /* level 3 */
            b = a + b + c + x;
        }
    }
}
```

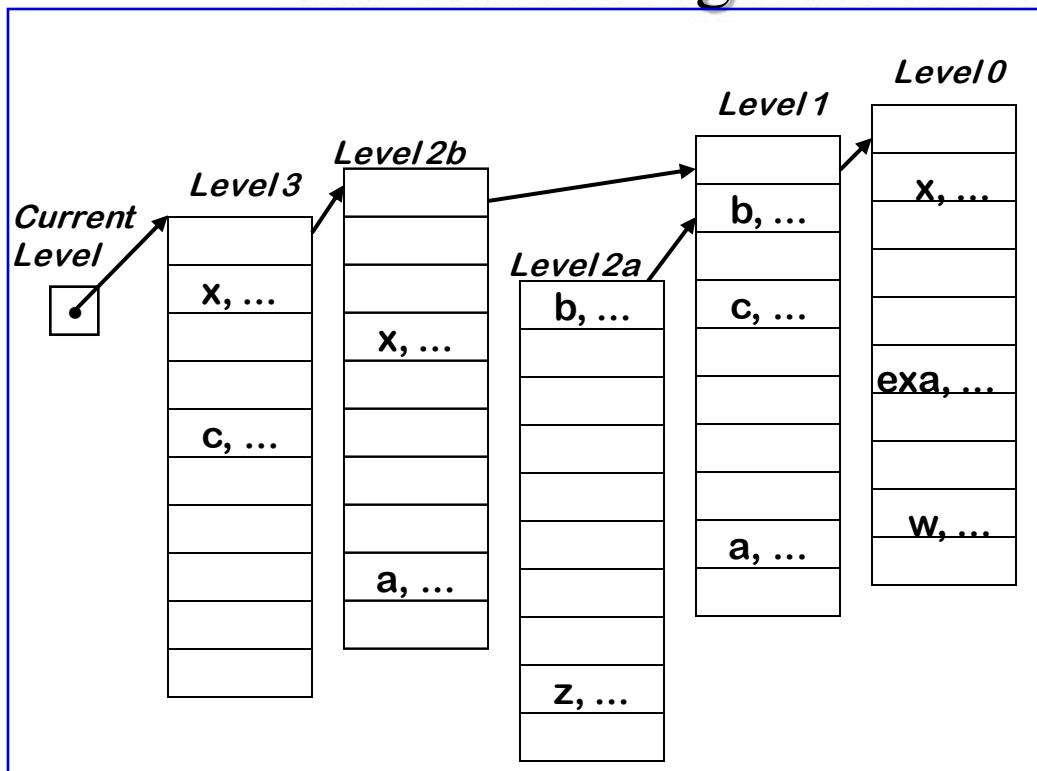
Generated sequence of calls:

- InitializeScope()
- Insert(a)
- Insert(b)
- Insert(c)
- InitializeScope()
- Insert(b)
- Insert(z)
- FinalizeScope()
- InitializeScope()
- Insert(a)
- Insert(x)
- InitializeScope()
- Insert(c)
- Insert(x)
- Lookup(b)
- LookUp(a)
- LookUp(b)
- LookUp(c)
- LookUp(x)
- FinalizeScope()
- FinalizeScope()
- FinalizeScope()

Lexically-scoped Symbol Tables

High-level idea

- Create a new table for each scope
- Chain them together for lookup



- *Insert()* inserts at the current level
- *LookUp()* walks chain of tables & returns first occurrence of name
- *InitializeScope()* creates a new table, connects it to the current level and updates the current level to point to the new table
- *FinalizeScope()* removes the table which is the top table in the chain

Lexically-scoped Symbol Tables

A simpler implementation

- Store a number which indicates the lexical level (scope) with each name in the symbol table
- InitializeScope() simply increments an internal counter for the current lexical level
- FinalizeScope() must find each record with the current lexical level and remove them before decrementing the current lexical level

Hash tables

- use a table (array) of size m to store n elements from a set of much larger size
- given a key k , use a function h to compute the slot $h(k)$ for that key.
 - h is a hash function
 - k hashes to slot $h(k)$
 - the hash value of k is $h(k)$
 - collision: when two keys have the same hash value
 - $\lambda=n/m$ is the load factor of the hash table

Hash tables

- We must choose a "good" hash function: easy to compute, distributes keys uniformly
 - truncation
 - folding
 - division ($h(k)=k \bmod m$, make m : prime)
 - multiplication
 - $h(k)=\lfloor m * (k * c - \lfloor k * c \rfloor) \rfloor, 0 < c < 1$
 - good $c: (\sqrt{5}-1)/2$
 - see Data Structures textbooks for more

Hash tables

- We must determine a way to handle collisions (two keys hash to the same slot)
 - chaining
 - put all elements that hash to the same slot in a doubly linked list (chain).
The slot contains a pointer to the head of the list.
 - Insert : O(1)
 - Delete : O(1)
 - Search : O(1+ λ)
 - open addressing
 - probe the table until an empty slot is found

Back to symbol tables

- The symbol table holds **all** identifiers. What if the same name is used in different scopes to refer to different objects?
 - Use a different symbol table for each scope
 - To access non locals look at the symbol tables at enclosing scopes.
 - Use the same symbol table but
 - keep track of what is "visible" at each time.
 - use unique "labels" to identify similarly named objects.
 - Example: our class project

Outline

- How to build symbol tables
- How to use them to find
 - multiply-declared and
 - undeclared variables.
- How to perform type checking

The Compiler So Far

- Lexical analysis
 - Detects inputs with illegal tokens
 - e.g.: main\$ ();
- Parsing
 - Detects inputs with ill-formed parse trees
 - e.g.: missing semicolons
- Semantic analysis
 - Last “front end” phase
 - Catches all remaining errors

Introduction

- typical semantic errors:
 - multiple declarations: a variable should be declared (in the same scope) at most once
 - undeclared variable: a variable should not be used before being declared.
 - type mismatch: type of the left-hand side of an assignment should match the type of the right-hand side.
 - wrong arguments: methods should be called with the right number and types of arguments.

An sample semantic analyzer

- works in two phases
 - i.e., it traverses the AST created by the parser:
 1. For each scope in the program:
 - **process the declarations =**
 - add new entries to the symbol table and
 - report any variables that are multiply declared
 - **process the statements =**
 - find uses of undeclared variables, and
 - update the "ID" nodes of the AST to point to the appropriate symbol-table entry.
 2. Process all of the statements in the program again,
 - use the symbol-table information to determine the type of each expression, and to find type errors.

Symbol Table = set of entries

- purpose:
 - keep track of names declared in the program
 - names of
 - variables, classes, fields, methods,
- symbol table entry:
 - associates a name with a set of attributes, e.g.:
 - kind of name (variable, class, field, method, etc)
 - type (int, float, etc)
 - nesting level
 - memory location (i.e., where will it be found at runtime).

Scoping

- symbol table design influenced by what kind of scoping is used by the compiled language
- In most languages, the same name can be declared multiple times
 - if its declarations occur in different scopes, and/or
 - involve different kinds of names.

Scoping: example

- Java: can use same name for
 - a class,
 - field of the class,
 - a method of the class, and
 - a local variable of the method
- *legal Java program:*

```
class Test {  
    int Test;  
    void Test( ) { double Test; }  
}
```

Scoping: overloading

- Java and C++ (but not in Pascal or C):
 - can use the same name for more than one method
 - as long as the number and/or types of parameters are unique.

```
int add(int a, int b);  
float add(float a, float b);
```

Scoping: general rules

- The scope rules of a language:
 - determine which declaration of a named object corresponds to each use of the object.
 - i.e., scoping rules map uses of objects to their declarations.

- C++ and Java use *static scoping*:
 - mapping from uses to declarations is made at compile time.
 - C++ uses the "most closely nested" rule
 - a use of variable x matches the declaration in the most closely enclosing scope
 - such that the declaration precedes the use.

Scope levels

- Each function has two or more scopes:
 - one for the parameters,
 - one for the function body,
 - and possibly additional scopes in the function
 - for each *for* loop and
 - each nested block (delimited by curly braces)

Example

```
void f( int k ) { // k is a parameter
    int k = 0;      // also a local variable
    while (k) {
        int k = 1;      // another local variable, in a loop
    }
}
```

- the outmost scope includes just the name "f", and
- function f itself has three (nested) scopes:
 1. The outer scope for f just includes parameter k.
 2. The next scope is for the body of f, and includes the variable k that is initialized to 0.
 3. The innermost scope is for the body of the while loop, and includes the variable k that is initialized to 1.

TEST YOURSELF #1

-
- This is a C++ program. Match each use to its declaration, or say why it is a use of an undeclared variable.

```
int k=10, x=20;
void foo(int k) {
    int a = x;
    int x = k;
    int b = x;
    while (...) {
        int x;
        if (x == k) {
            int k, y;
            k = y = x;
        }
        if (x == k) { int x = y; }
    }
}
```

Dynamic scoping

- Not all languages use static scoping.
- Lisp, APL, and Snobol use **dynamic** scoping.
- Dynamic scoping:
 - A use of a variable that has no corresponding declaration in the same function corresponds to the declaration in the **most-recently-called still active** function.

Example

- For example, consider the following code:

```
void main() { f1(); f2(); }
```

```
void f1() { int x = 10; g(); }
```

```
void f2() { String x = "hello"; f3();  
g(); }
```

```
void f3() { double x = 30.5; }
```

```
void g() { print(x); }
```

TEST YOURSELF #2

-
- Assuming that dynamic scoping is used, what is output by the following program?

```
void main() { int x = 0; f1(); g(); f2(); }
```

```
void f1() { int x = 10; g(); }
```

```
void f2() { int x = 20; f1(); g(); }
```

```
void g() { print(x); }
```

Review of Part I

- Static vs dynamic scoping
 - generally, dynamic scoping is a bad idea
 - can make a program difficult to understand
 - a single use of a variable can correspond to
 - many different declarations
 - with different types!
- can a name be used before they are defined?
 - Java: a method or field name *can* be used before the definition appears,
 - *not* true for a variable!

Example

```
class Test {  
    void f() {  
        val = 0;  
        // field val has not yet been declared --  
        OK  
        g();  
        // method g has not yet been declared --  
        OK  
        x = 1;  
        // var x has not yet been declared --  
        ERROR!  
        int x;  
    }  
    void g() {}  
    int val;  
}
```

Simplification

- From now on, assume that our language:
 - uses static scoping
 - requires that *all* names be declared before they are used
 - does not allow multiple declarations of a name in the same scope
 - even for different kinds of names
 - *does* allow the same name to be declared in multiple nested scopes
 - but only once per scope
 - uses the same scope for a method's parameters and for the local variables declared at the beginning of the method
- This will make your life in P4 much easier !

Symbol Table Implementations

- In addition to the above simplification, assume that the symbol table will be used to answer two questions:
 1. Given a declaration of a name, is there already a declaration of the same name in the current scope
 - i.e., is it multiply declared?
 2. Given a use of a name, to which declaration does it correspond (using the "most closely nested" rule), or is it undeclared?

Note

- The symbol table is only needed to answer those two questions, i.e.
 - once all declarations have been processed to build the symbol table,
 - and all uses have been processed to link each ID node in the abstract-syntax tree with the corresponding symbol-table entry,
 - then the symbol table itself is no longer needed
 - because no more lookups based on name will be performed

What operation do we need?

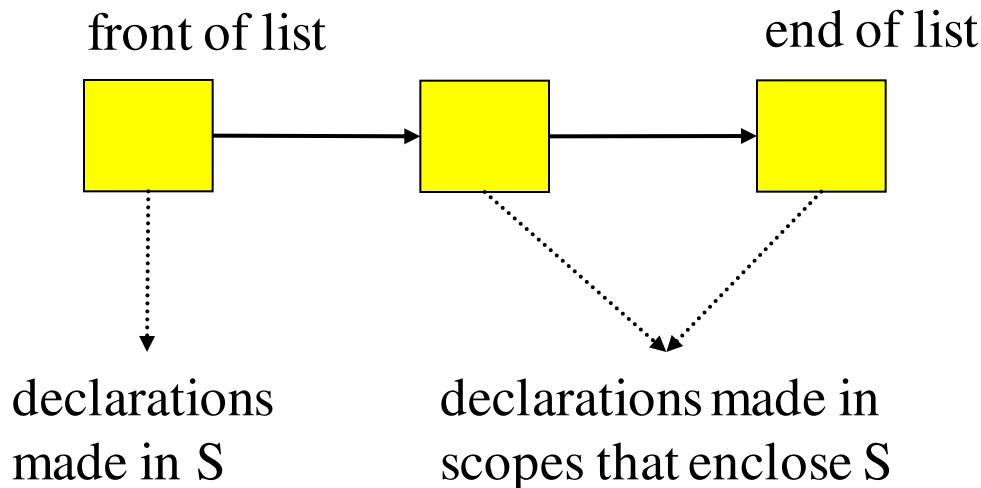
- Given the above assumptions, we will need:
 1. Look up a name in the current scope only
 - to check if it is multiply declared
 2. Look up a name in the current and enclosing scopes
 - to check for a use of an undeclared name, and
 - to link a use with the corresponding symbol-table entry
 3. Insert a new name into the symbol table with its attributes.
 4. Do what must be done when a new scope is entered.
 5. Do what must be done when a scope is exited.

Two possible symbol table implementations

1. a list of tables
 2. a table of lists
- For each approach, we will consider
 - what must be done when entering and exiting a scope,
 - when processing a declaration, and
 - when processing a use.
 - Simplification:
 - assume each symbol-table entry includes only:
 - the symbol name
 - its type
 - the nesting level of its declaration

Method 1: List of Hashtables

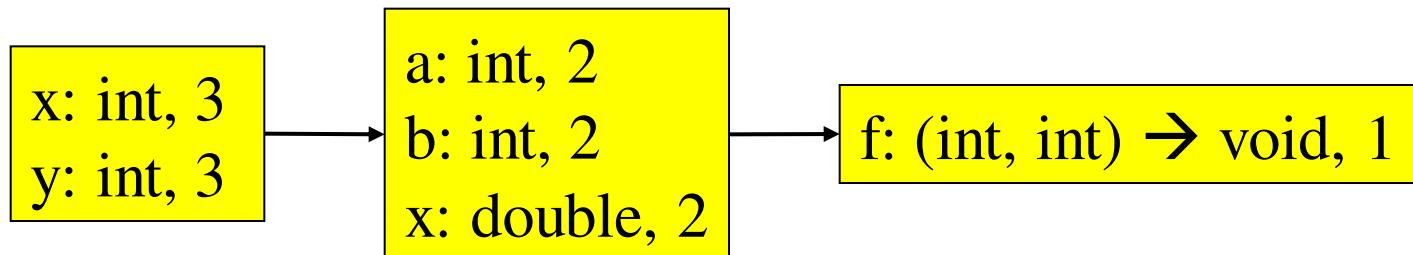
- The idea:
 - symbol table = a list of hashtables,
 - one hashtable for each currently visible scope.
- When processing a scope S:
 - front of list
 - end of list



Example:

```
void f(int a, int b) {  
    double x;  
    while (...) { int x, y; ... }  
}  
void g() { f(); }
```

- After processing declarations inside the while loop:



List of hashtables: the operations

1. On scope entry:

- increment the current level number and add a new empty hashtable to the front of the list.

2. To process a declaration of x:

- look up x in the first table in the list.
 - If it is there, then issue a "multiply declared variable" error;
 - otherwise, add x to the first table in the list.

... continued

3. To process a use of x:

- look up x starting in the first table in the list;
 - if it is not there, then look up x in each successive table in the list.
 - if it is not in *any* table then issue an "undeclared variable" error.

4. On scope exit,

- remove the first table from the list and decrement the current level number.

Remember

- method names belong into the hashtable for the outermost scope
 - not into the same table as the method's variables
- For example, in the example above:
 - method name f is in the symbol table for the outermost scope
 - name f is *not* in the same scope as parameters a and b, and variable x.
 - This is so that when the use of name f in method g is processed, the name is found in an enclosing scope's table.

The running times for each operation:

1. Scope entry:

- time to initialize a new, empty hashtable;
- probably proportional to the size of the hashtable.

2. Process a declaration:

- using hashing, constant expected time ($O(1)$).

3. Process a use:

- using hashing to do the lookup in each table in the list, the worst-case time is $O(\text{depth of nesting})$, when every table in the list must be examined.

4. Scope exit:

- time to remove a table from the list, which should be $O(1)$ if garbage collection is ignored

TEST YOURSELF #1

- **Question 1:** C++ does not use exactly the scoping rules that we have been assuming.
 - In particular, C++ **does** allow a function to have both a parameter and a local variable with the same name
 - any uses of the name refer to the local variable
 - Consider the following code. Draw the symbol table as it would be after processing the declarations in the body of *f* under:
 - the scoping rules we have been assuming
 - C++ scoping rules
 - ```
void g(int x, int a) { }
void f(int x, int y, int z) { int a, b, x;
 ... }
```

# ... continued

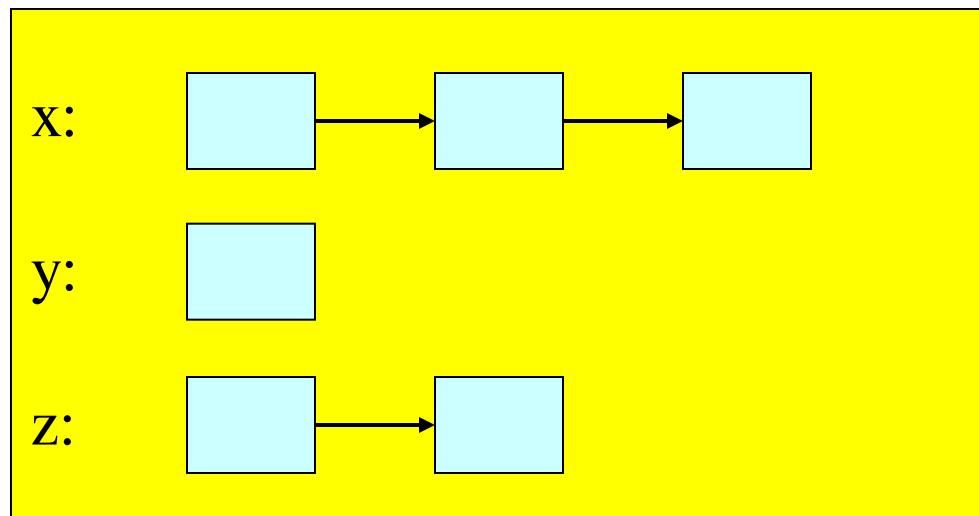
---

## ○ Question 2:

- Which of the four operations described above
  - scope entry,
  - process a declaration,
  - process a use,
  - scope exit
- would change (and how) if the following rules for name reuse were used instead of C++ rules:
  - the same name can be used within one scope as long as the uses are for different kinds of names, and
  - the same name *cannot* be used for more than one variable declaration in a nested scope

# Method 2: Hashtable of Lists

- the idea:
  - when processing a scope S, the structure of the symbol table is:



# Definition

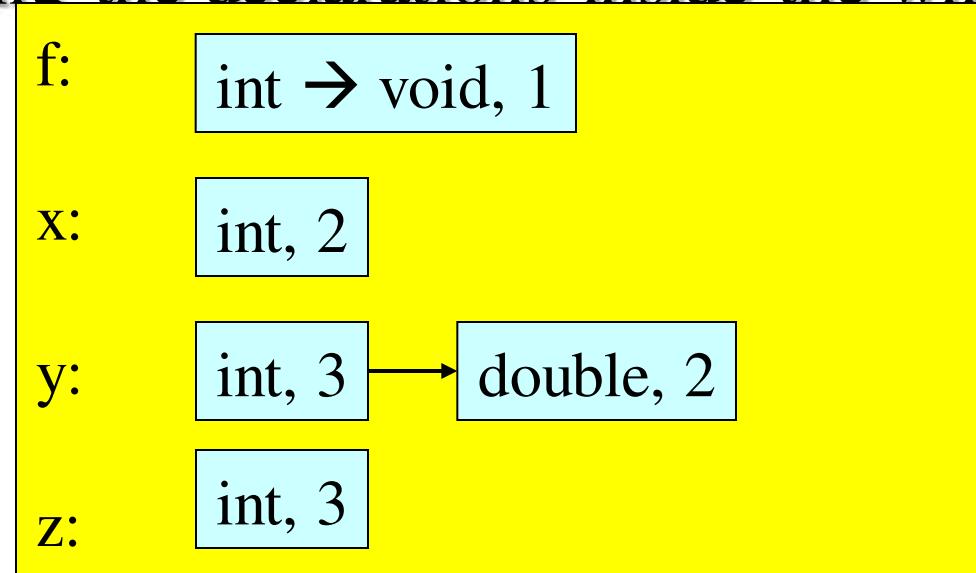
---

- there is just one big hashtable, containing an entry for each variable for which there is
  - some declaration in scope S or
  - in a scope that encloses S.
- Associated with each variable is a list of symbol-table entries.
  - The first list item corresponds to the most closely enclosing declaration;
  - the other list items correspond to declarations in enclosing scopes.

# Example

```
void f(int a) {
 double x;
 while (...) { int x, y; ... }
 void g() { f(); }
}
```

- After processing the declarations inside the while loop:



# Nesting level information is crucial

---

- the level-number attribute stored in each list item enables us to determine whether the most closely enclosing declaration was made
  - in the current scope or
  - in an enclosing scope.

# Hashtable of lists: the operations

---

## 1. On scope entry:

- increment the current level number.

## 2. To process a declaration of x:

- look up x in the symbol table.

➤ If x is there, fetch the level number from the first list item.

- If that level number = the current level then issue a "multiply declared variable" error;
- otherwise, add a new item to the front of the list with the appropriate type and the current level number.

# ... continue

---

- 1. To process a use of x:**
  - look up x in the symbol table.
  - If it is not there, then issue an "undeclared variable" error.
- 2. On scope exit:**
  - scan all entries in the symbol table, looking at the first item on each list. If that item's level number = the current level number, then remove it from its list (and if the list becomes empty, remove the entire symbol-table entry). Finally, decrement the current level number.

# Running times

---

## 1. Scope entry:

- time to increment the level number,  $O(1)$ .

## 2. Process a declaration:

- using hashing, constant expected time ( $O(1)$ ).

## 3. Process a use:

- using hashing, constant expected time ( $O(1)$ ).

## 4. Scope exit:

- time proportional to the number of names in the symbol table (or perhaps even the size of the hashtable if no auxiliary information is maintained to allow iteration through the non-empty hashtable buckets).

## TEST YOURSELF #2

---

- Assume that the symbol table is implemented using a hashtable of lists.
- Draw pictures to show how the symbol table changes as each declaration in the following code is processed.

```
void g(int x, int a) {
 double d;
 while (...) {
 int d, w;
 double x, b;
 if (...) { int a,b,c; }
 }
 while (...) { int x,y,z; }
}
```

# Type Checking

---

- the job of the type-checking phase is to:
  - Determine the type of each expression in the program
    - (each node in the AST that corresponds to an expression)
  - Find type errors
- The **type rules** of a language define
  - how to determine expression types, and
  - what is considered to be an error.
- The type rules specify, for every operator (including assignment),
  - what types the operands can have, and
  - what is the type of the result.

# Example

---

- both C++ and Java allow the addition of an int and a double, and the result is of type double.
- However,
  - C++ also allows a value of type double to be assigned to a variable of type int,
  - Java considers that an error.

## TEST YOURSELF #3

---

- List as many of the operators that can be used in a Java program as you can think of
  - don't forget to think about the logical and relational operators as well as the arithmetic ones
- For each operator,
  - say what types the operands may have, and
  - what is the type of the result.

# Other type errors

---

- the type checker must also
  1. find type errors having to do with the **context** of expressions,
    - e.g., the context of some operators must be boolean,
  2. type errors having to do with method calls.
- Examples of the context errors:
  - the condition of an *if* statement
  - the condition of a *while* loop
  - the termination condition part of a *for* loop
- Examples of method errors:
  - calling something that is not a method
  - calling a method with the wrong number of arguments
  - calling a method with arguments of the wrong types

# Compilers Construction

## Chapter 6

### Semantic Analysis

*Dr. Mohammed Salem Atoum*

[M.Atoum@inu.edu.jo](mailto:M.Atoum@inu.edu.jo)

# Type Expression And Type Constructors

---

- Predefined
  - A number of built-in, with names like int and double .
  - Such data type are simple type , in that their values four-byte two's complement form
  - Atypical representation for real , or floating-point, number is four or eight bytes.
  
- New data can be created using type constructors, such as array and record or struct

# Arrays

---

- the array type constructor take two type parameters
  - **index type** and
  - **component type** and
  - produces a new array type
- In Pascal-like syntax, we write
  - array [index-type] of component-type
  - For example, the Pascal type expression.
    - Array [color] of char;
  - **index type** is *color* and **component type** is *char*, such type include
    - *integer* and *character subranges* and *enumerated type*

# Arrays

---

- One complication in the declaration of array is that of **multi-dimensioned** array
  - array [0 .. 9] of array [color] of integer;
- or in a shorter version in which the index set are listed together:
  - array [0 .. 9, color] of integer;
- Open-indexed array is especially useful in the declaration of array parameters to function.
- For example, the C declaration :
  - void sort (int a[ ] , int first , int last)

# Records

---

- A **record** or **structure** type constructor takes a list of name and associated type and constructor a new type, as in the C:

```
struct
{ double r;
 int i;
}
```

- The previously given record structure need six bytes of storage, which is allocated as follows:



# Records

---

- A **record** or **structure** type constructor takes a list of name and associated type and constructor a new type, as in the C:

```
struct
{ double r;
 int i;
}
```

- The previously given record structure corresponds to the cartesian product
  - $(r \times R) \times (i \times I)$ .

# Union

---

- A union type corresponds to the set union operation

```
union
{ double r;
 int i;
}
```

- If  $x$  is a variable of the given union type, then  $x.r$  means the value of  $x$  as a real number, while  $x.i$  means the value of  $x$  as an integer.
- Mathematic, this union is defined by writing :
  - $(r \times R) \cup (i \times I)$ .
- Union is to allocate memory in parallel for each component
- If a real number requires four bytes and an integer two bytes, then the previously given union structure need only four bytes of storage (the maximum of the memory requirements of its components)

# Union

---

- Union leads to error in data interpretation.
- For example, in C if x is a variable of the given union type, then writing
  - x.r = 2.0;
  - printf("%d",x.i);
- will not cause a compilation error, but will not print the value 2, but a garbage value instead.

# Pointers

---

- A pointer type consists of values that references to value of another type.
- Thus, a value of a pointer type is a memory address whose location holds a value of its base type.
- Yet they are pointer type constructor, there is also no standard set operation that directly corresponds to the pointer type constructor, as cartesian product corresponds to the record constructor.
- In pascal the character ^ corresponds to the pointer type constructor, so that the type expression **^ integer** means “pointer to integer”.
- Pointer type are allocated space based on the address size of the target machine.

# Functions

---

- we have already noted that an array can be viewed as a function from its index set to its component set.
- For example, in modula-2 the declaration
  - var f: procedure (integer) : integer;
  - Declares the variable **f** to be of function (or procedure) type, with an integer parameter and yielding an integer result.
- **Class:** most object-oriented languages have a class declaration that is similar to record declaration
- Class hierarchy (a directed acyclic graph), which implement inheritance

# Errors

---

- Compiler must check that the source program follows both the *syntactic* and *semantic* rules of the source language.
- **Type checking** is the activity of ensuring that the operands of an operator are of compatible types
- A **compatible type** is one that is either legal for the operator, or is allowed under **language rules** to be implicitly converted, by compiler-generated code, to a legal type.
- This automatic conversion is called a **coercion**

# Type checker

---

- **Type checks:**
  - A compiler should report an error if an operator is applied to an incompatible operand.
    - an array variable is added to a function variable
  - Verifies that the type of a construct matches the expected type (using the context)
- **Examples:**
  - Pascal **mod** built-in function requires integers operands, so the type checker must verify this
  - Dereferencing is applied only to pointers: \*ptr
  - Indexing is done only to arrays: s[10]
  - User-defined function is applied to the correct number and type of arguments

# Type checker

---

- Compiler may supply **coercion** of types (auto conversion) that convert operand to an expected type

- e.g.  $x(\text{float}) = y(\text{int}) + z(\text{float})$



Auto convert to float

# Type System

---

- The design of type checker for a language is based on:
  - Information about syntactic constructs in the language
  - The notion of types
  - Rules for assigning types to language constructs
- Examples of information needed for compilers writers:
  - Pascal: if both operands of addition, sub, and mul are **integers**, then the result is of type **integer**
  - C: the result of unary operator **&** is a pointer to the same type to which **&** operator applied to

```
int x, *ptr;
ptr = &x;
```
- Each **expression** has a *type* associated with it

# Type System

---

- Types are either **basic** or **constructed**
- **Basic types:**
  - are atomic with no internal structure as far as the program is constructed
    - Character, integer, real, boolean, double
- **Constructed types:**
  - Types constructed from other types
    - Pointers, arrays, functions, structures can be treated as constructor types

# Type Expressions

---

- A *type expression* is either:
  - a *basic type* or
  - is formed by applying an operator called *type constructor* to other type expressions
  
- The following is *Type expressions*:
  - A *basic type*
    - integer, real, void, etc
  - A *type name*
    - since type expressions may be named
  - A *type constructor*
    - array, records, etc

# Constructors

---

- Array:
  - If  $T$  is a type expression, then **array**( $I, T$ ) is a type expression, where  $I$  is an index set (often a range of integers), and  $T$  is *type expression* associated with the elements of the array:
    - var A: array [1..10] of integer;  
associates the type expression **array(1..10, integer)** with A
- Products:
  - if  $T_1$  and  $T_2$  are type expressions, then  $T_1 \times T_2$  is a type expression

# Constructors

## ○ Records (structures):

- differs from the products as fields have names.
- record type constructor will be applied to a tuple of formed from field names and field types

```
type row = record
 address: integer;
 lexeme: array[1..15] of char
 end;
var table: array[1..101] of row;
```

## ○ Declares the type name **row** representing the type expression:

- record ((address × integer) × (lexeme × array(1..15, char)))

## ○ And the variable **table** to be an

- array(1..101, row)

# Constructors

---

- Pointers:
  - If  $T$  is a type expression, then  $\text{pointer}(T)$  is a type expression
  - $\text{pointer}(T)$  is a pointer to an object of type  $T$ 
    - $\text{var } p: \uparrow \text{row}$        $p$  has type  $\text{pointer}(\text{row})$
- Functions:
  - Functions map a domain type  $D$  to a range type  $R$ :  
 $D \rightarrow R$
  - **mod** function in Pascal:  
 $\text{int} \times \text{int} \rightarrow \text{int}$
- function  $f$  has 2 char arguments and returns a pointer to integer object. The type of  $f$  thus denoted by the type expression:
  - $\text{char} \times \text{char} \rightarrow \text{pointer}(\text{integer})$

# Test Your Self

- A function  $g$  has a **function** as an argument that maps **integer** to an **integer**, and  $g$  produces as a result another **function** of the same type.

- The answer is:

□  $(\text{integer} \rightarrow \text{integer}) \rightarrow (\text{integer} \rightarrow \text{integer})$



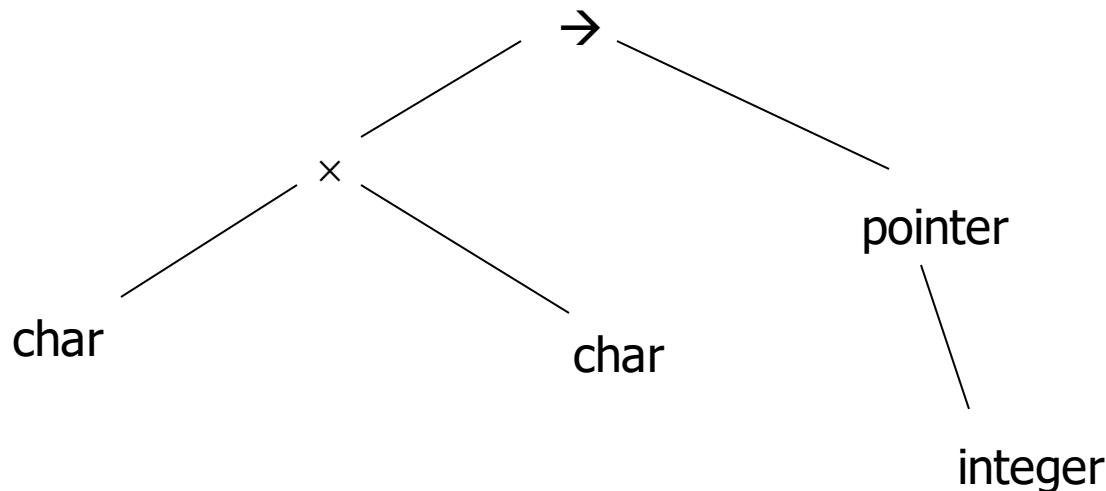
**Argument function**

**Return function**

# Type Expressions

- Representation of type expression using tree:

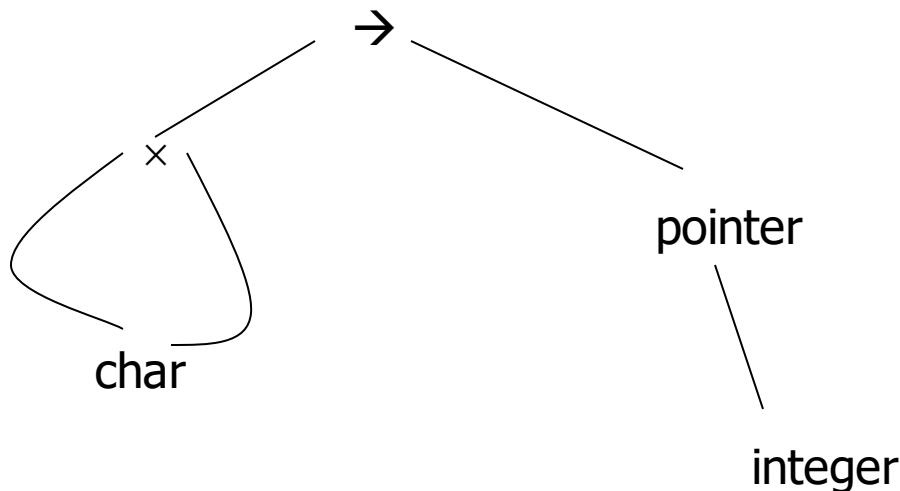
$\text{char} \times \text{char} \rightarrow \text{pointer}(\text{integer})$



# Type Expressions

- Representation of type expression using **dag**:

$\text{char} \times \text{char} \rightarrow \text{pointer}(\text{integer})$



# Type Systems

---

- A **type system** is a collection of rules for assigning *type expressions* to the various parts of a program
- A **type checker** implements a type system
- For example:
  - The type of an **array** includes the **index set** of an array, so a **function** with an array argument can only be applied to arrays with that index set
  - Some compilers allow the **index set** to be left unspecified when an array is passed as an argument
  - These compilers use a different **type system**

# Static and Dynamic Checking of Types

---

- **Static checks:**
  - reported during the compilation phase
- **Dynamic checks**
  - occur during the execution of the program
- A **sound** type system
  - eliminates the need for **dynamic checking** for type errors
  - because it allows us to determine **statically** that these errors cannot occur when the target program runs
- A language is **strongly** typed if its compiler can guarantee that the programs it accepts will execute without type errors
- Some type errors can be done only dynamically, for example:  
**table: array[0..255] of char**  
**i: integer;**  
**table[i] = 5;**

then the compiler can not guarantee that during execution, the value of **i** lies in the range 0..255. (determined at run time)

# Error Recovery

---

- Since type checking has the potential for catching errors in programs, it is important to do something when an error is discovered
  - nature of error
  - location
  - how to recover from the errors

# Specifications of a Simple Type Checker

---

- The type of each identifier must be declared before the identifier is used
- The **type checker** is a translation scheme that synthesizes the type of each expression from the type of its sub-expressions
- Type Checker can handle
  - arrays, pointers, statements and functions

# A Simple Language

---

- $P \rightarrow D ; E$
- $D \rightarrow D ; D \mid id : T$
- $T \rightarrow char \mid integer \mid array [ num ] of T \mid \uparrow T$
- $E \rightarrow literal \mid num \mid id \mid E \text{ mod } E \mid E[ E ] \mid E \uparrow$
  
- One program generated by this grammar is:
  - key : integer;
  - key mod 1999
- 
- This language has two basic types: char and integer; and the third type type\_error is used to signal errors

# Translation Schema

---

- $P \rightarrow D; E$
- $D \rightarrow D; D$
- $D \rightarrow id : T \{ addtype(id.entry, T.type); \}$
- $T \rightarrow \text{char} \{ T.type = \text{char}; \}$
- $T \rightarrow \text{integer} \{ T.type = \text{integer}; \}$
- $T \rightarrow \uparrow T_1 \{ T.type = \text{pointer}(T_1.type); \}$
- $T \rightarrow \text{array}[num] \text{ of } T_1$   
 $\{ T.type = \text{array}(1..num.val, T_1.type); \}$

# Translation Schema

---

- $P \rightarrow D ; E$ 
  - As we know the types of all declared identifiers will be saved before the expression generated by  $E$  is checked
  - Because  $D$  appears before  $E$  on the right side of production above
- $D \rightarrow id : T \{ addtype(id.entry, T.type); \}$ 
  - Saves a type in a symbol-table entry for an identifier
- Action **addtype (id.entry, T.type)** applied to:
  - synthesized attribute *entry* pointing to the symbol-table entry for **id** and
  - type expression represented by synthesized attribute *type* of nonterminal **T**
  - So, if **T** generates char or integer, then **T.type** is defined to be char or integer respectively

# Type Checking of Expression

---

- In the following rules, the synthesized attribute **type** for E gives the type expression assigned by the type system to the expression generated by E.
  - $E \rightarrow \text{literal}$  {E.type:=char}
  - $E \rightarrow \text{num}$  {E.type:=integer}
  - $E \rightarrow \text{id}$  {E.type:=lookup (id.entry)}
- **lookup(e)** is a function used to fetch the type saved in the symbol-table entry pointed to by e
- So, when an identifier appear in an expression, its declared type is fetched and assigned to the attribute type

# Type Checking of Expressions

---

- $E \rightarrow E_1 \bmod E_2$ 
  - {E.type := If (E1.type = integer) && if (E2. Type = integer) then integer; else type-error;}
- $E \rightarrow E_1[E_2]$ 
  - {E.type := if ((E2.type = integer) && (E1.type=array(s, t)) then t; else type-error; }
- $E \rightarrow E_1 \uparrow$ 
  - {E.type := if (E1.type = pointer(t)) then t else type-error;}

# Type Checking of statements

---

- Because the language constructs statements typically do not have values, the special basic type **void** can be assigned to them
- The statement that we consider are **assignment**, **conditional** and **while** statements
- 
- $S \rightarrow id := E$ 
  - { $S.type :=$  if  $id.type = E.type$  then void else type\_error}
- This checks that the left and right sides of an assignment statement have the same type

# Type Checking for Statements

---

- $S \rightarrow \text{if } E \text{ then } S_1$ 
  - $\{S.\text{type} := \text{if } E.\text{type} = \text{boolean} \text{ then } S_1.\text{type} \text{ else type\_error}\}$
- $S \rightarrow \text{while } E \text{ do } S_1$ 
  - $\{S.\text{type} := \text{if } E.\text{type} = \text{boolean} \text{ then } S_1.\text{type} \text{ else type\_error}\}$
- These rules specify that expressions in **conditional** and **while** statements must have type **boolean**
- $S \rightarrow S_1 ; S_2$ 
  - $\{S.\text{type} := \text{if } S_1.\text{type} = \text{void} \text{ and } S_2.\text{type} = \text{void} \text{ then void else type\_error}\}$

# Type Checking of Functions

---

- The application of function to an argument can be captured by the production
  - $E \rightarrow E(E)$
  - in which an expression is the application of one expression to another
- the rules for associating type expressions with nonterminal  $T$  can be augmented by the following production and action to permit function types in declarations
- $T \rightarrow T_1 ' \rightarrow ' T_2$ 
  - $\{T.type := T_1.type \rightarrow T_2.type\}$
- Note that the quotes around the arrow used as a function constructor distinguish it from the arrow used as the metasymbol in a production

# Type Checking of Functions

- The rule for checking the type of a function application is
- $E \rightarrow E_1 ( E_2 )$ 
  - $E.type :=$  if  $E_2.type = s$  and  $E_1.type = s \rightarrow t$  then  $t$  else  $type\_error$
- In an expression formed by applying  $E_1$  to  $E_2$ , the type of  $E_1$  must be a function  $s \rightarrow t$  from the type  $s$  of  $E_2$  to some range type  $t$ ; the type of  $E_1 ( E_2 )$  is  $t$
- The generalization to functions with more than one argument is done by constructing a **product** type consisting of the arguments
- For Example, we might write
  - $root : (real \rightarrow real) \times integer \rightarrow real$
- To declare a function **root** that takes a **function** from **reals** to **reals** and a **integer** as arguments and returns a **real**.
- In pascal:
  - $function root ( function f (real) : real; x: integer ) : real$

# Type Equivalence

---

- As long as type expressions are built from basic types and constructors, a notion of equivalence between two type expressions is *structural equivalence*
- Two type expressions are structurally equivalent iff they are identical
- Ex: pointer(integer) is equivalent to pointer(integer)

# Structural Equivalence

---

boolean **sequival** (s, t)

{

    if s and t are the same basic type return true;  
    else if s = array(s1,s2) and t = array(t1,t2) then  
        return sequival(s1,t1) and sequival(s2,t2)  
    else if s = s1×s2 and t = t1×t2 then return  
        sequival(s1,t1) and sequival(s2,t2)  
    else if s= pointer(s1) and t=pointer(t1) then return  
        sequival(s1,t1)  
    else if s = s1 → s2 and t = t1 → t2 then return  
        sequival(s1,t1) and sequival(s2,t2)  
    else return false

}

# Type Name, Type Declaration, and Recursive Types

---

- Such type declarations (sometime also called type definitions) include the `typedef` mechanism of C and the type definition of pascal.
- For example, the C code

```
Typedef struct
{ double r;
 int I;
} RealIntRec
```

- Defines the name **RealIntRec** as a name for the record type constructed by the struct type expression that precedes it.

# Type Name, Type Declaration, and Recursive Types

---

- Type declarations cause the declared type names to be entered into the symbol table just as variable declarations cause variable name to be entered.
- The C language has a small exception to this rule in that names associated to struct or union declaration can be reused as typeddef names:

```
struct RealIntRec
{ double r;
 int r;
};
```

```
Typedef struct RealIntRec RealIntRec;
/* a legal declaration ! */
```

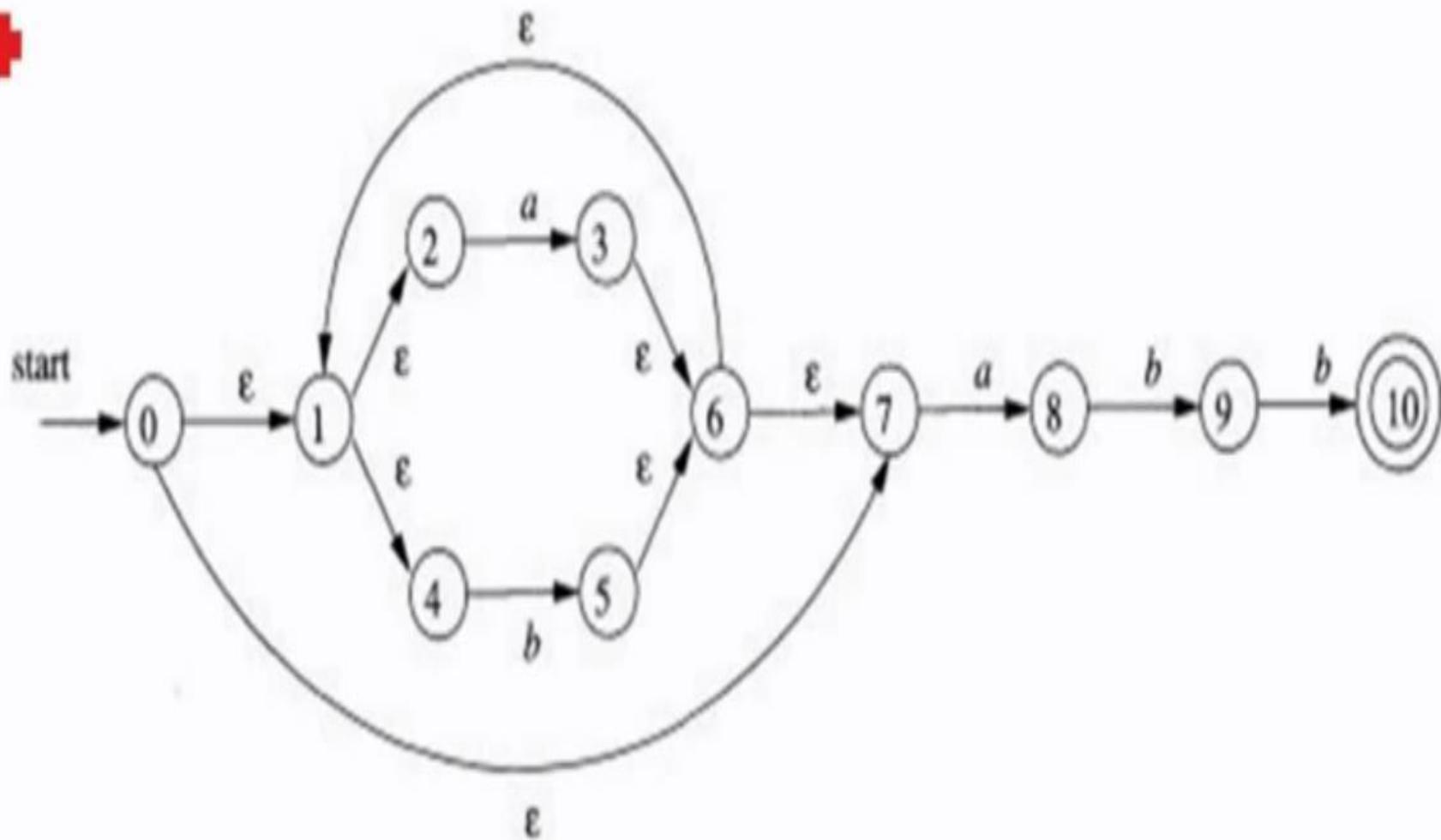
# **Conversion NFA to DFA**

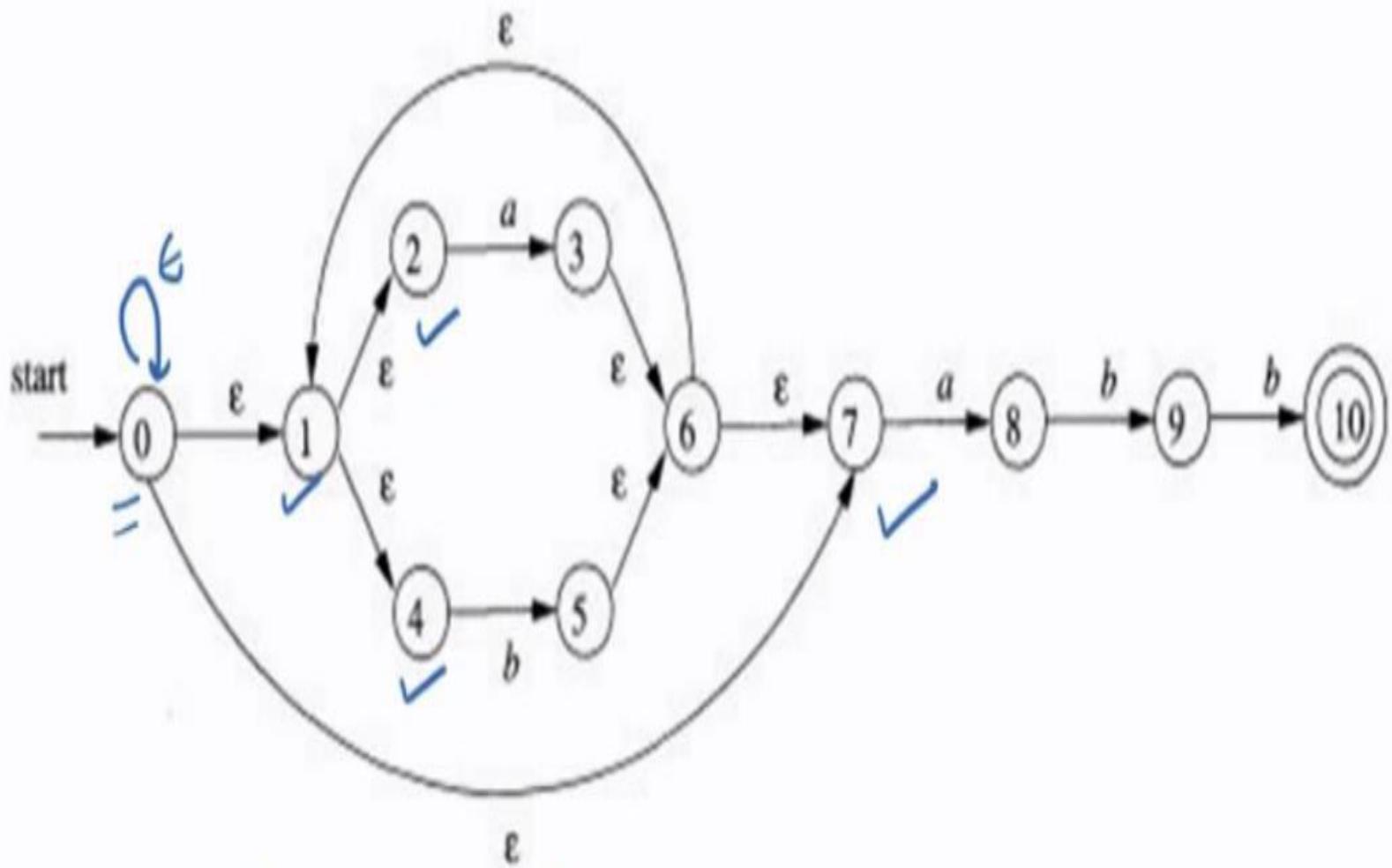
**Prepare By:**

**Dr. Mohammed Salem Atoum**

Example :

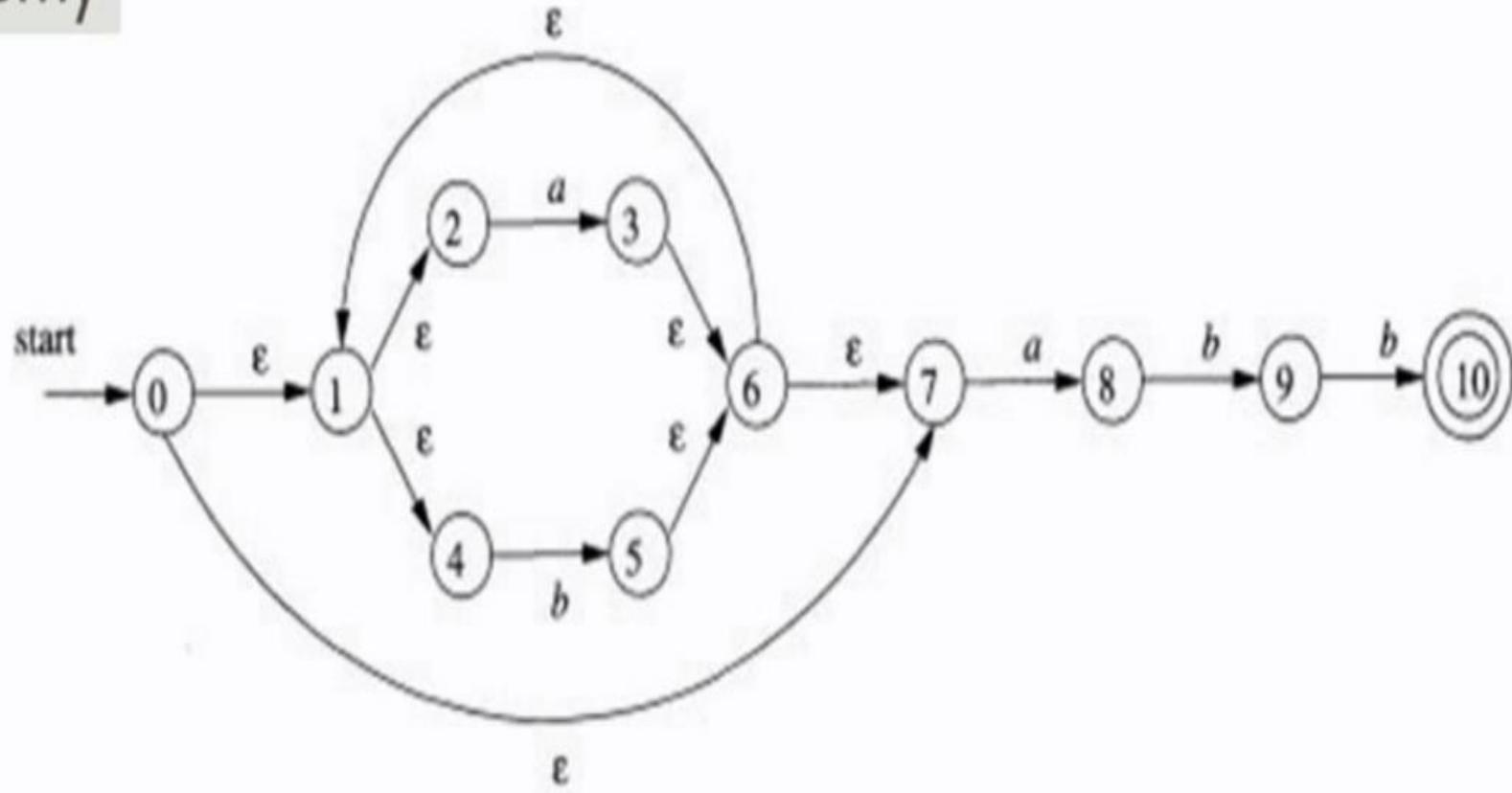
$(alb)^*abb$





$\text{E-closure}(0) = \{0, 1, 2, 4, 5\}$





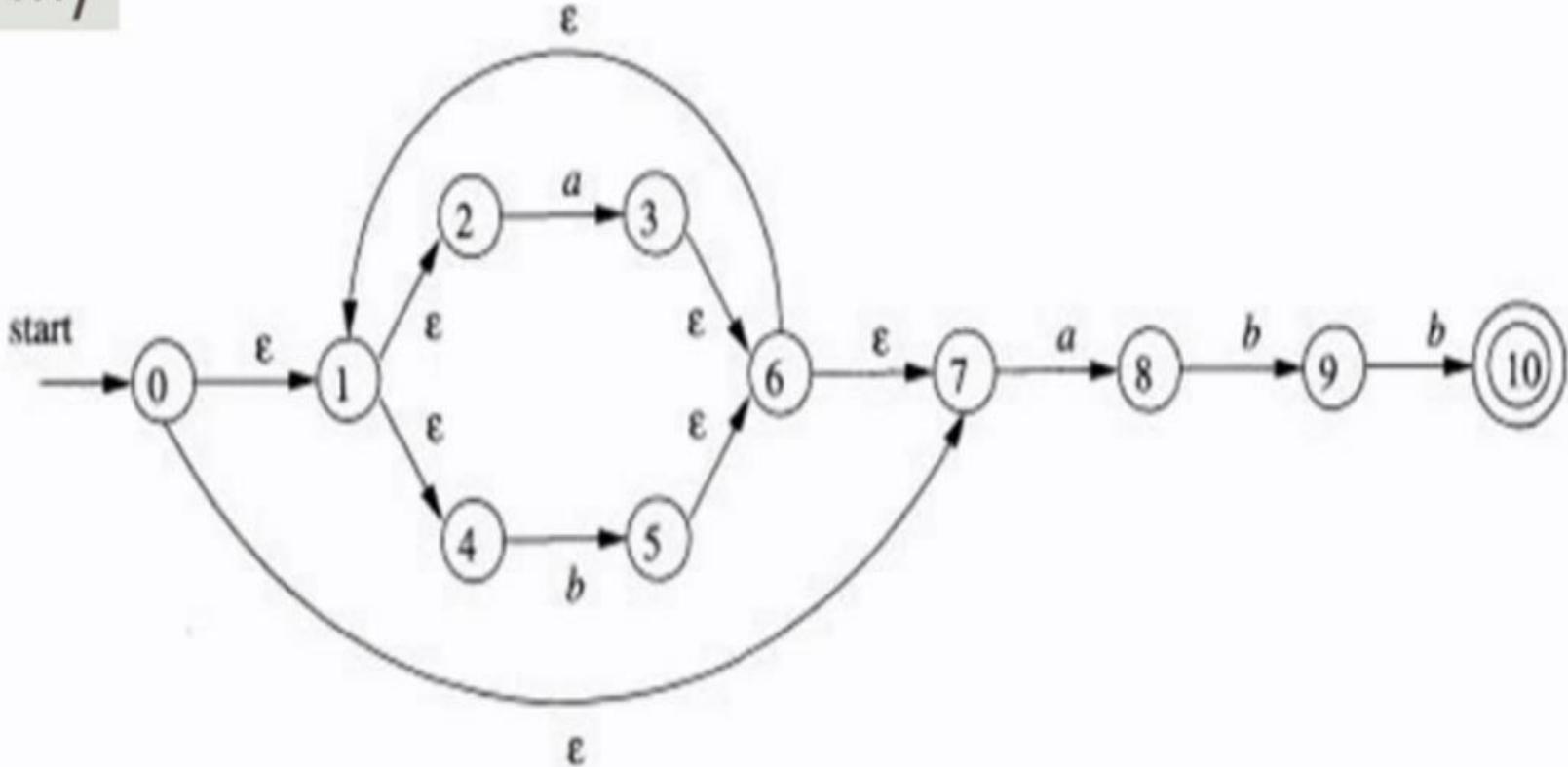
Give a name to E-closure(0) , State A of the DFA required.

State A of DFA contains these states of the NFA :

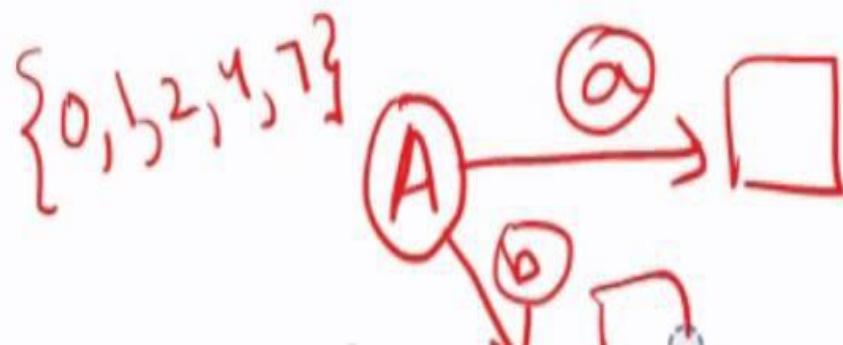
$$\{0, 1, 2, 4, 7\}$$

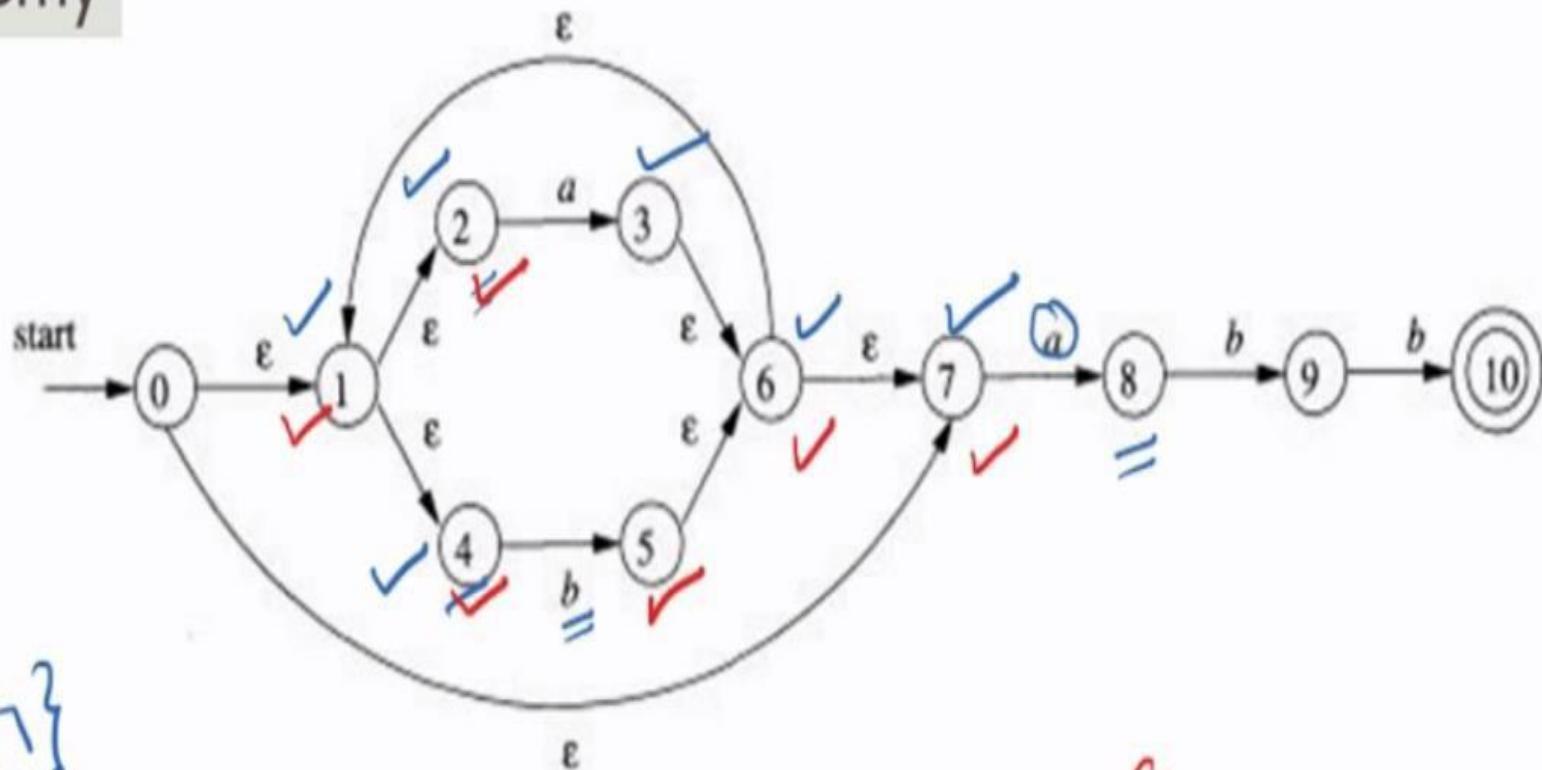
Inputs possible: a or b

A → state  
in DFA



Whenever we form a new set of states on any transition on this NFA, it is a new state for the DFA we require.




 $\{0, 1, 2, 4, 7\}$ 

A

 $\{3, 6, 1, 4, 2, 7, 8\}$ 

B

 $\{1, 2, 4, 5, 6, 7\}$ 

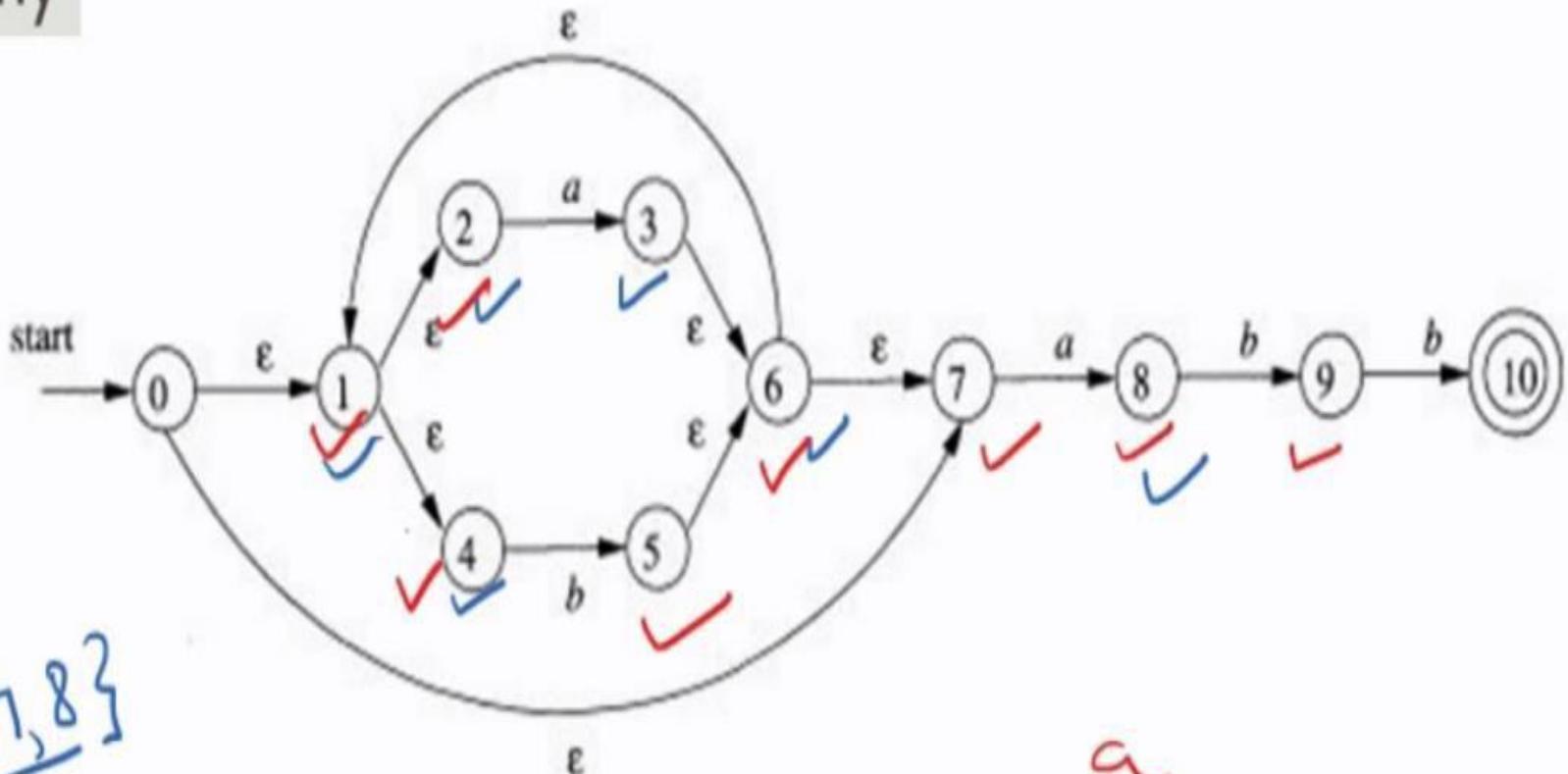
C

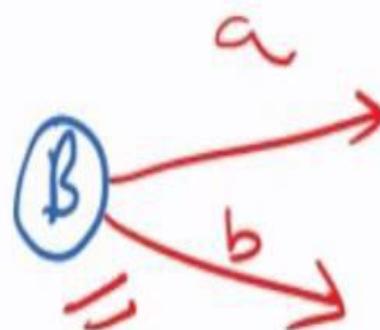
a

b

a

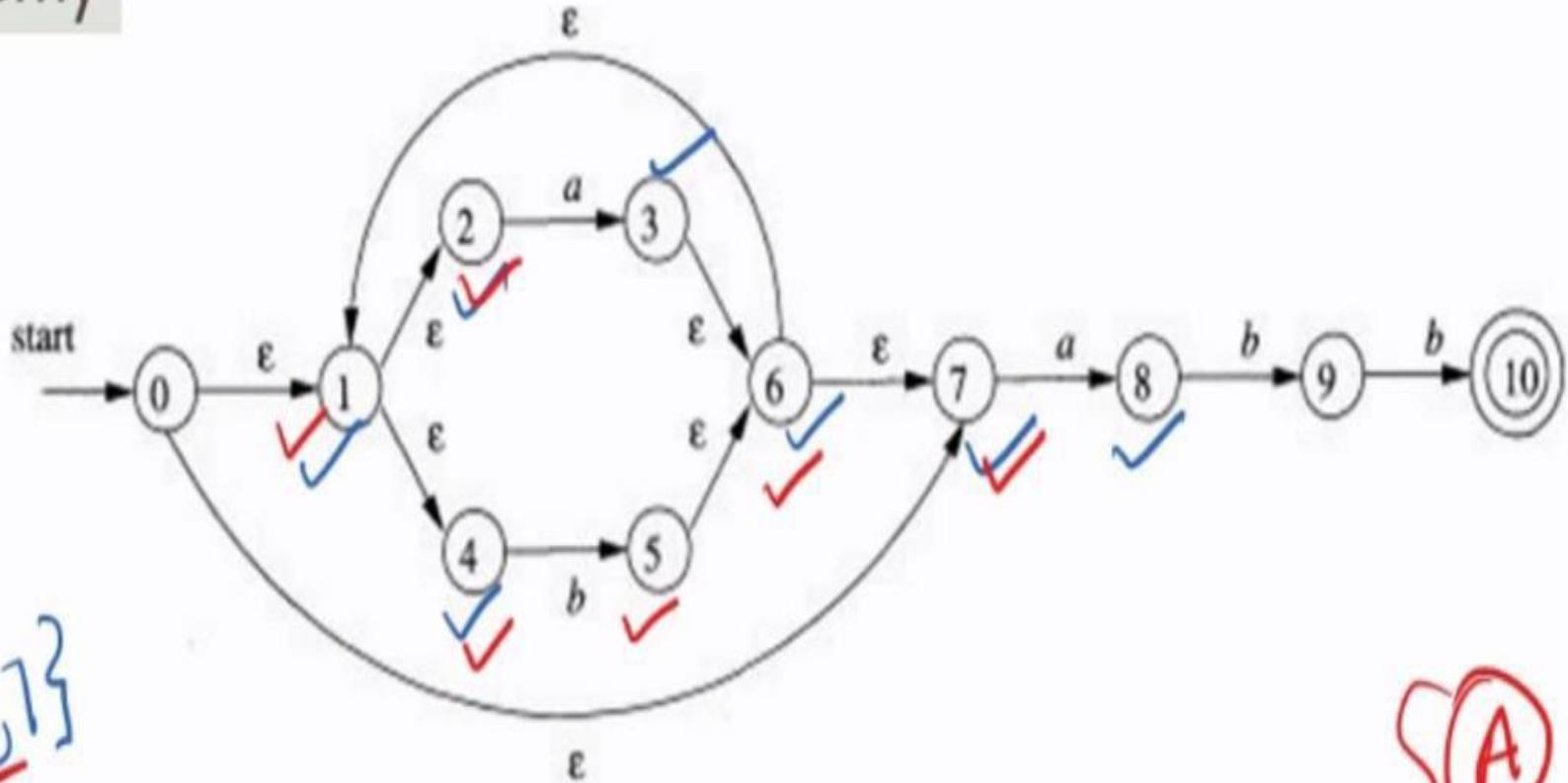
b


 $\{1, 2, 3, 4, 6, 7, 8\}$ 

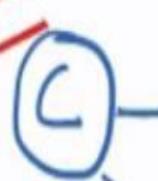
 $\{1, 2, 3, 4, 6, 7, 8\}$ 

 $\{1, 2, 4, 5, 6, 7, 9\}$ 

$D$

$\xrightarrow{a} \quad \xrightarrow{b}$



$\{1, 2, 4, 5, 6, 7\}$



a

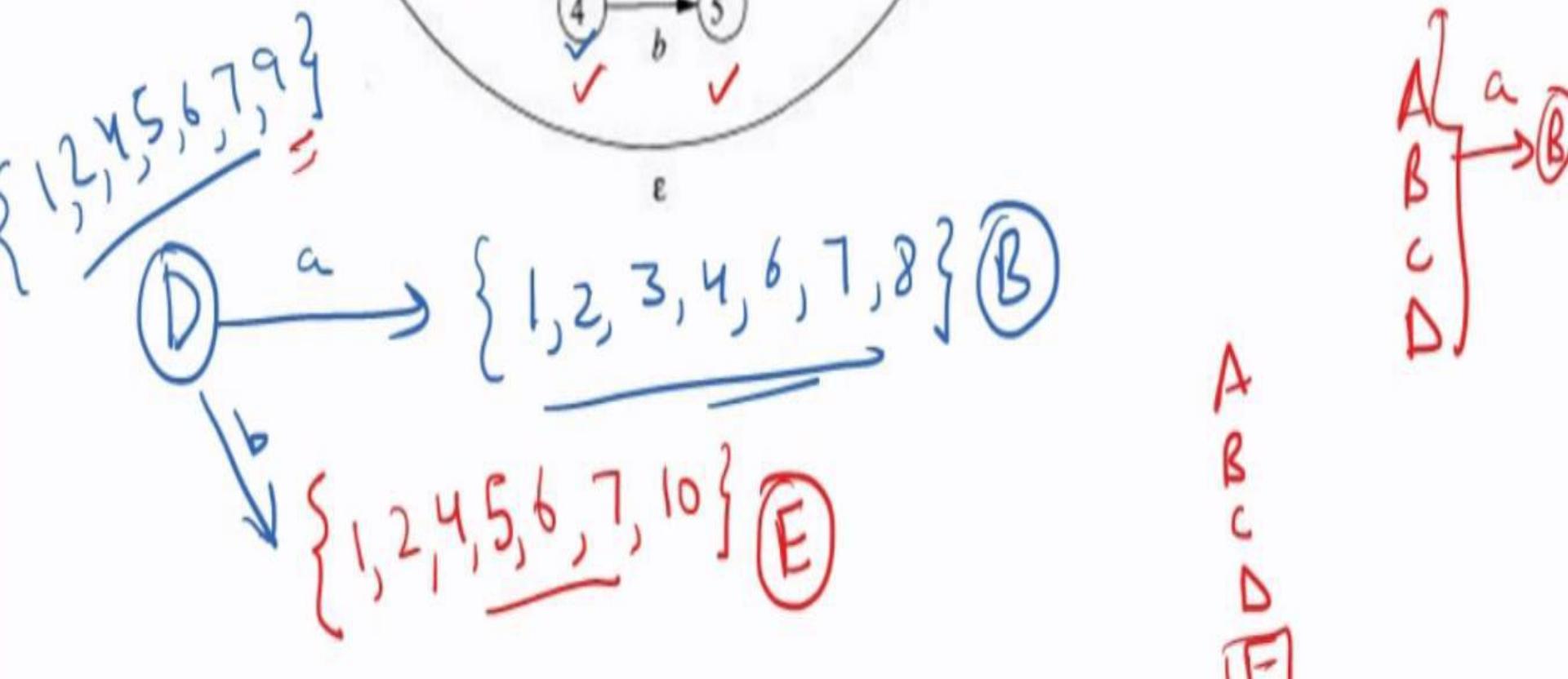
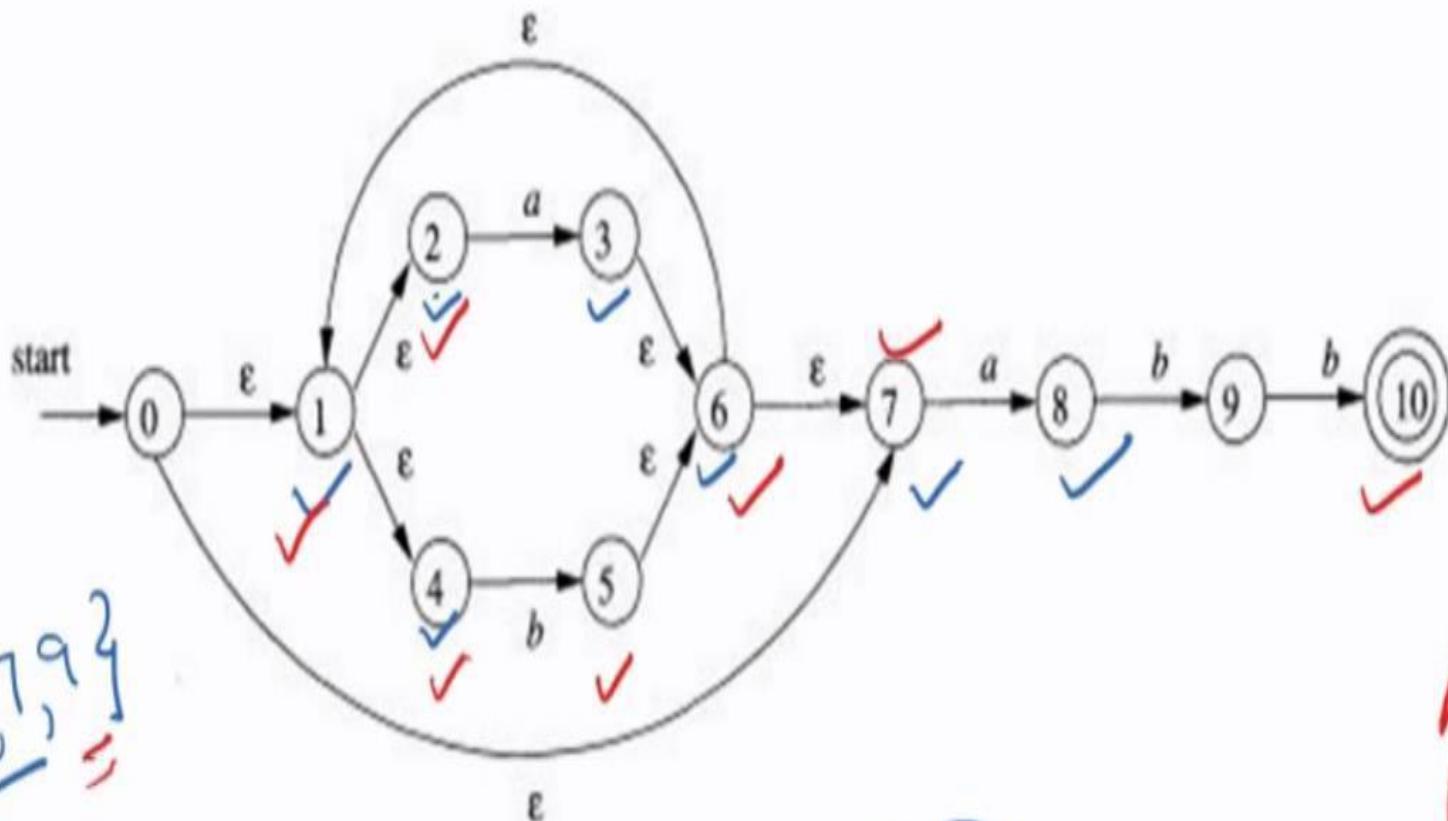
$\{1, 2, 4, 3, 6, 7, 8\}$  B

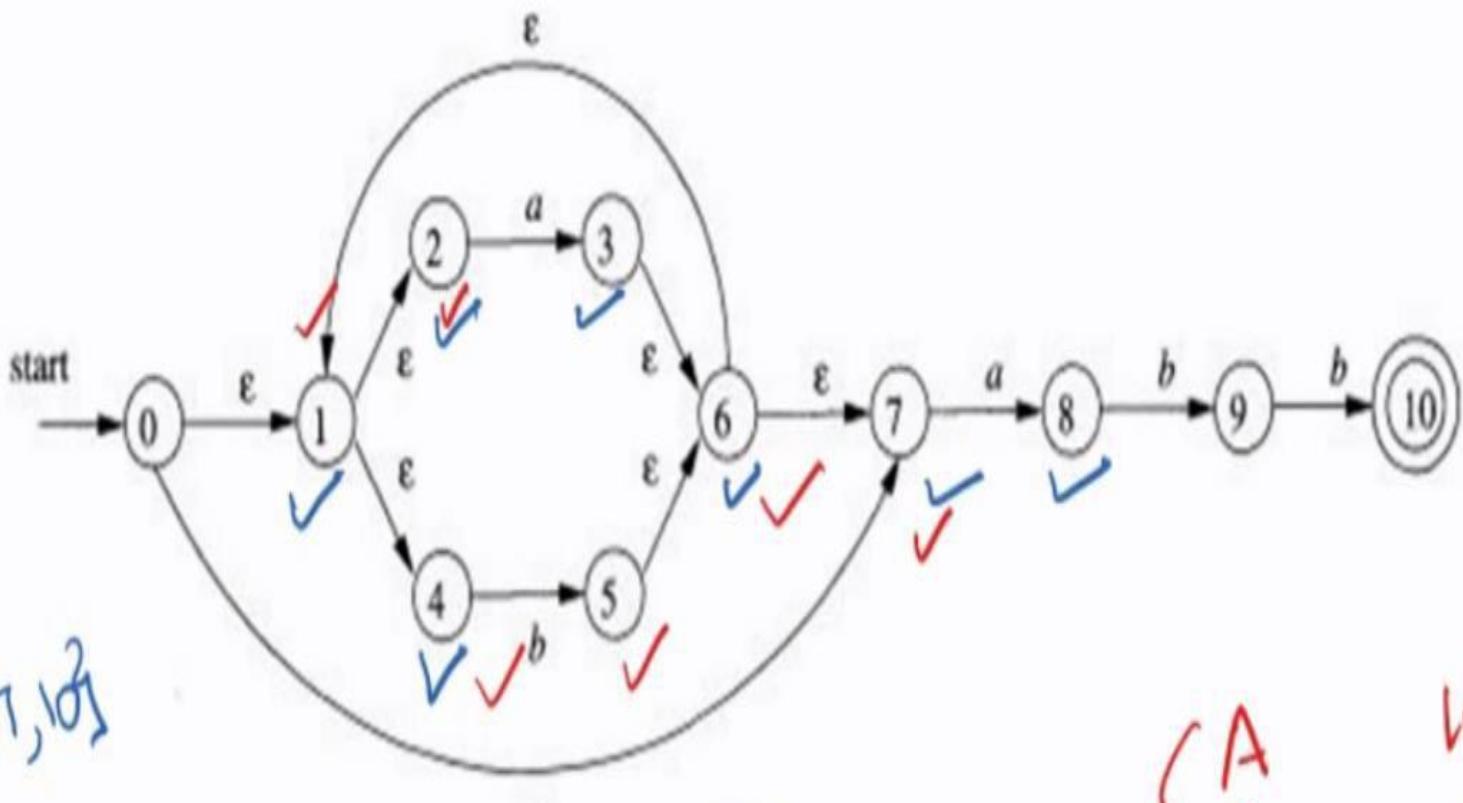


b

$\{1, 2, 4, 5, 6, 7\}$  C







$\{1, 2, 4, 5, 6, 7, 8\}$

E  $\xrightarrow{a} \{1, 2, 4, 5, 6, 7, 8\}$  B

E  $\xrightarrow{b} \{1, 2, 4, 5, 6, 7\}$  C

DFA states { A, B, C, D, E }  
 10 NFA states +

Final Sets:

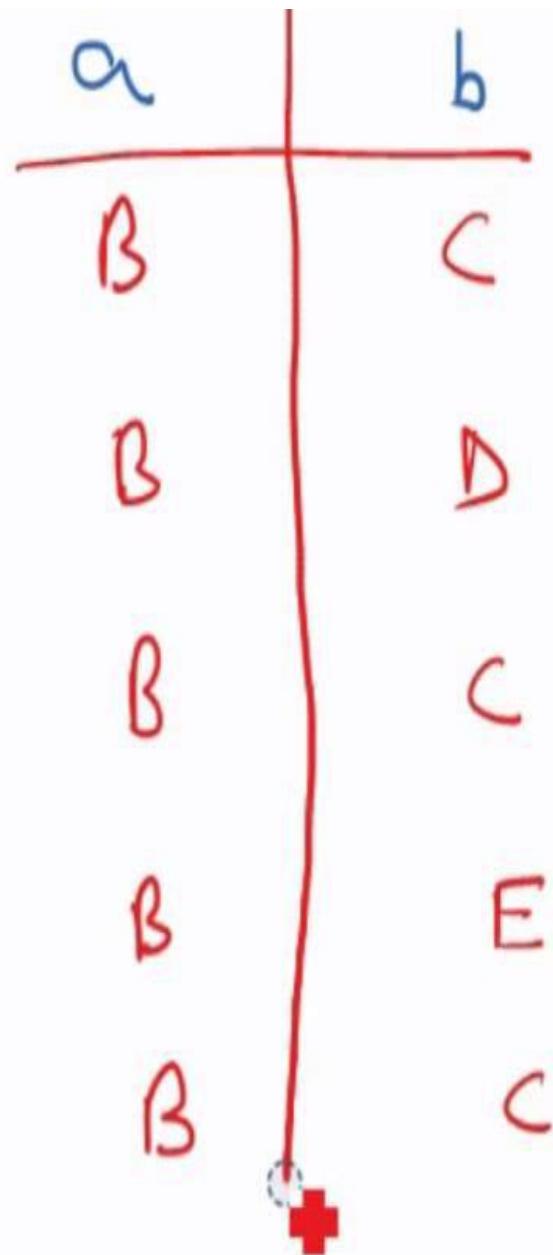
A  $\{0, 1, 2, 4, 7\}$

B  $\{1, 2, 3, 4, 6, 7, 8\}$

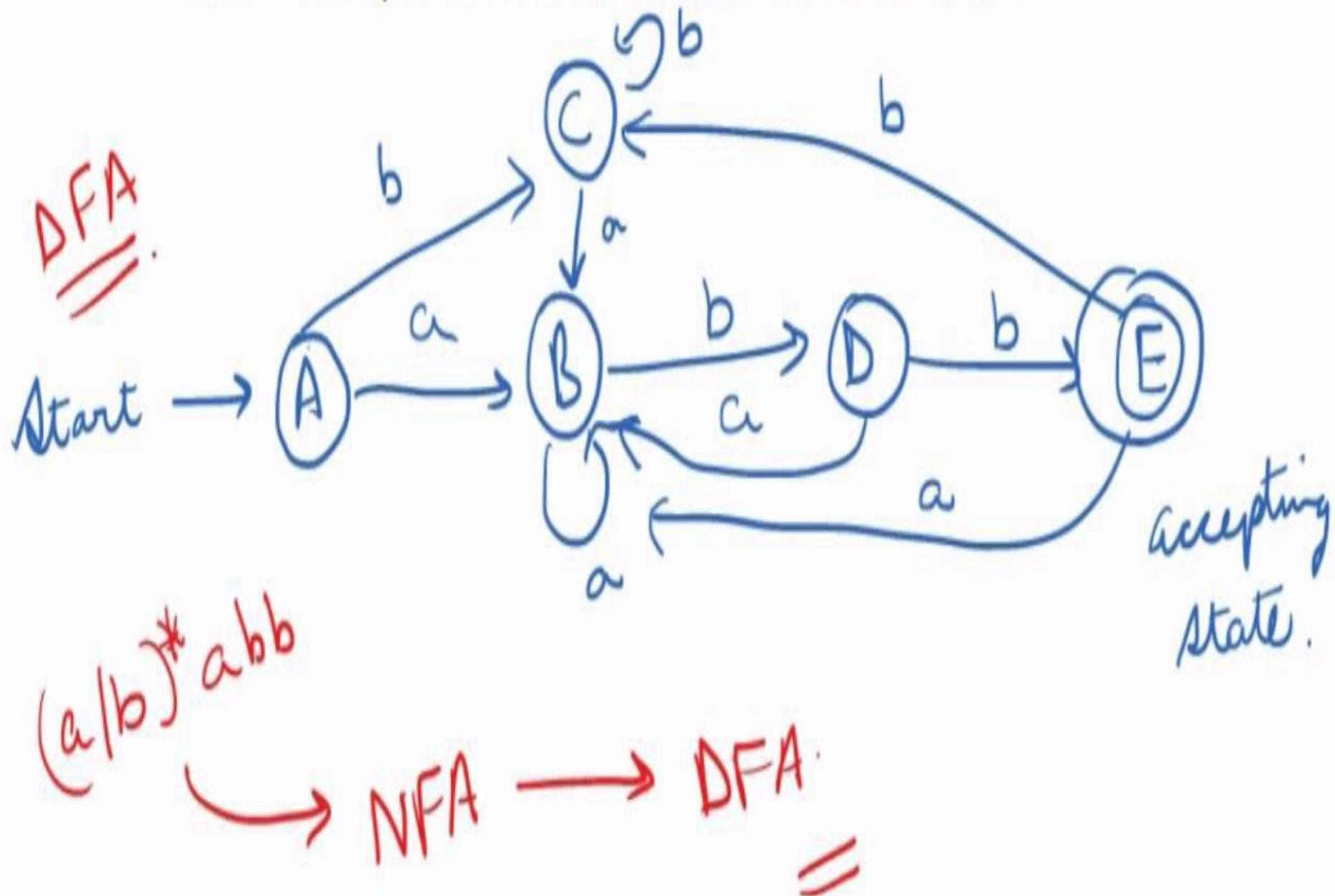
C  $\{1, 2, 4, 5, 6, 7\}$

D  $\{1, 2, 4, 5, 6, 7, 9\}$

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



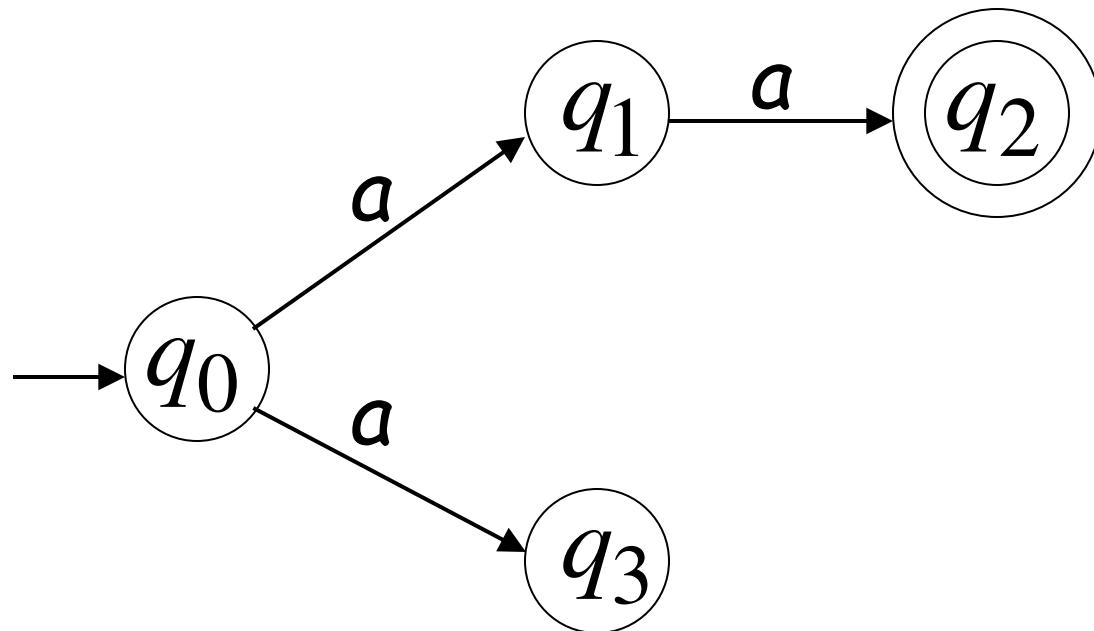
Now we finally use the sets and transitions to form the DFA.



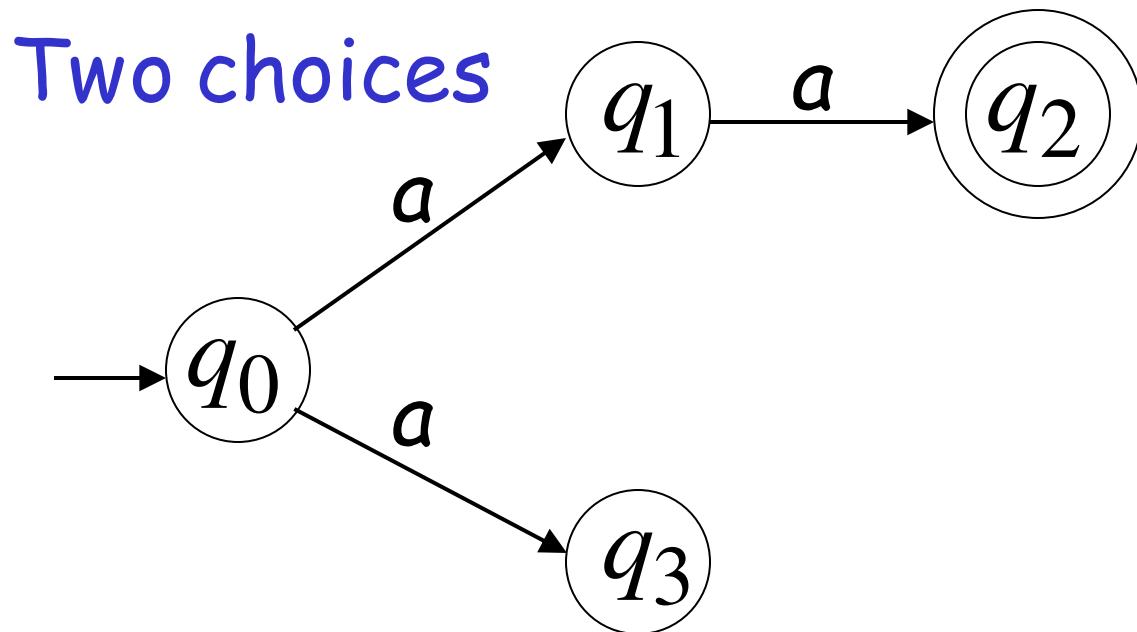
# Non-Deterministic Finite Automata

# Nondeterministic Finite Automaton (NFA)

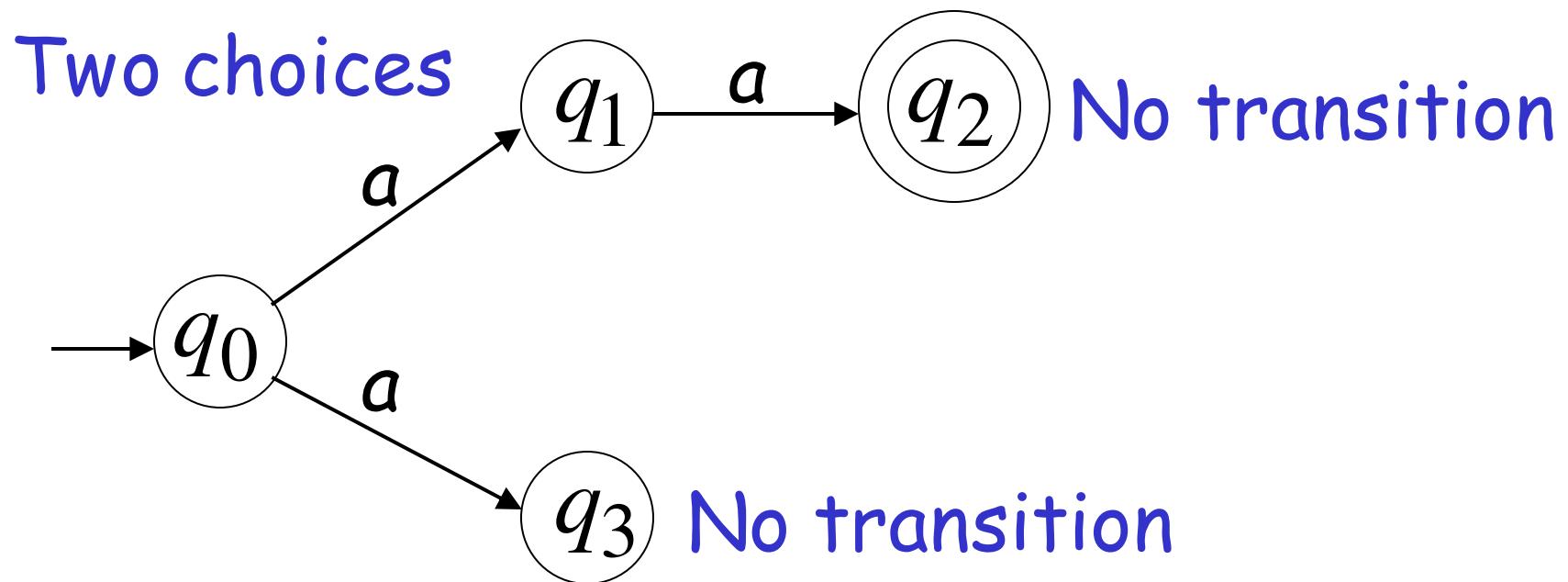
Alphabet =  $\{a\}$



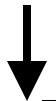
Alphabet =  $\{a\}$



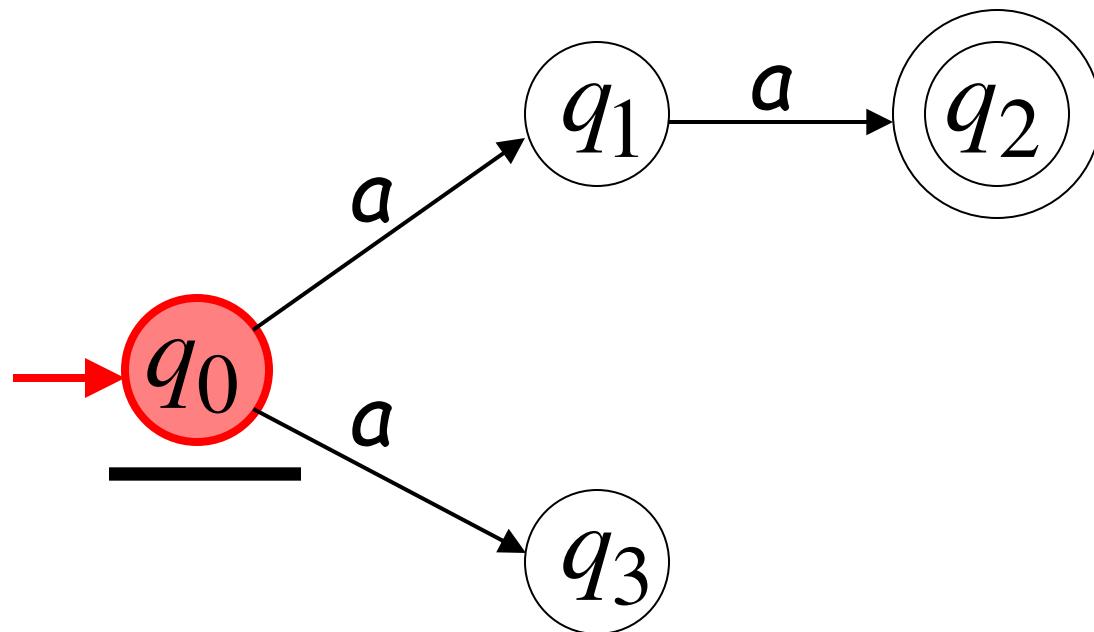
Alphabet =  $\{a\}$



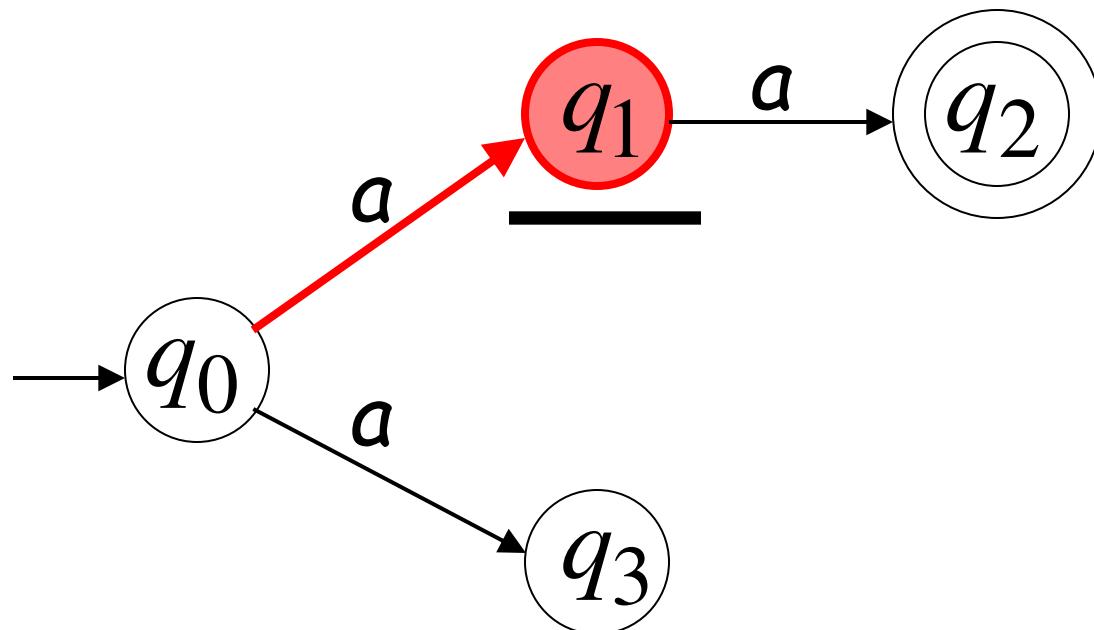
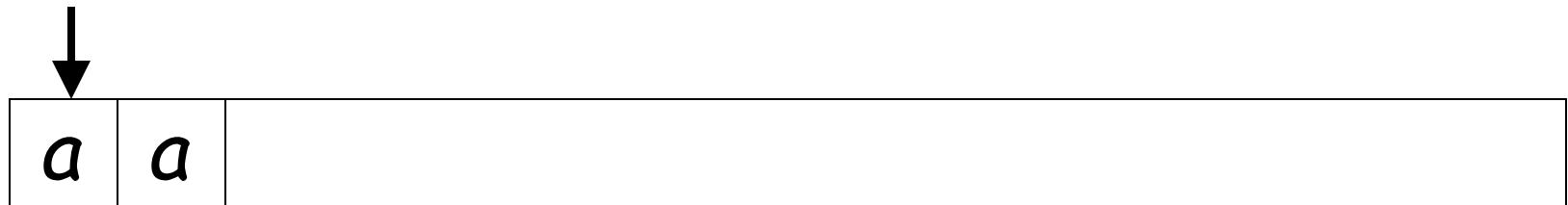
# First Choice



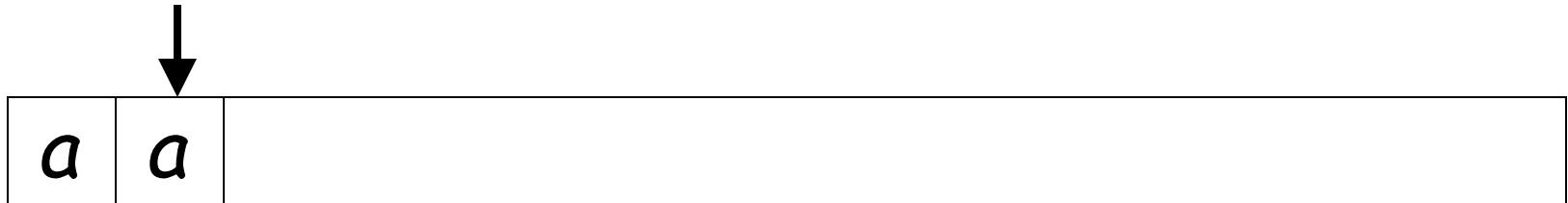
|     |     |  |
|-----|-----|--|
| $a$ | $a$ |  |
|-----|-----|--|



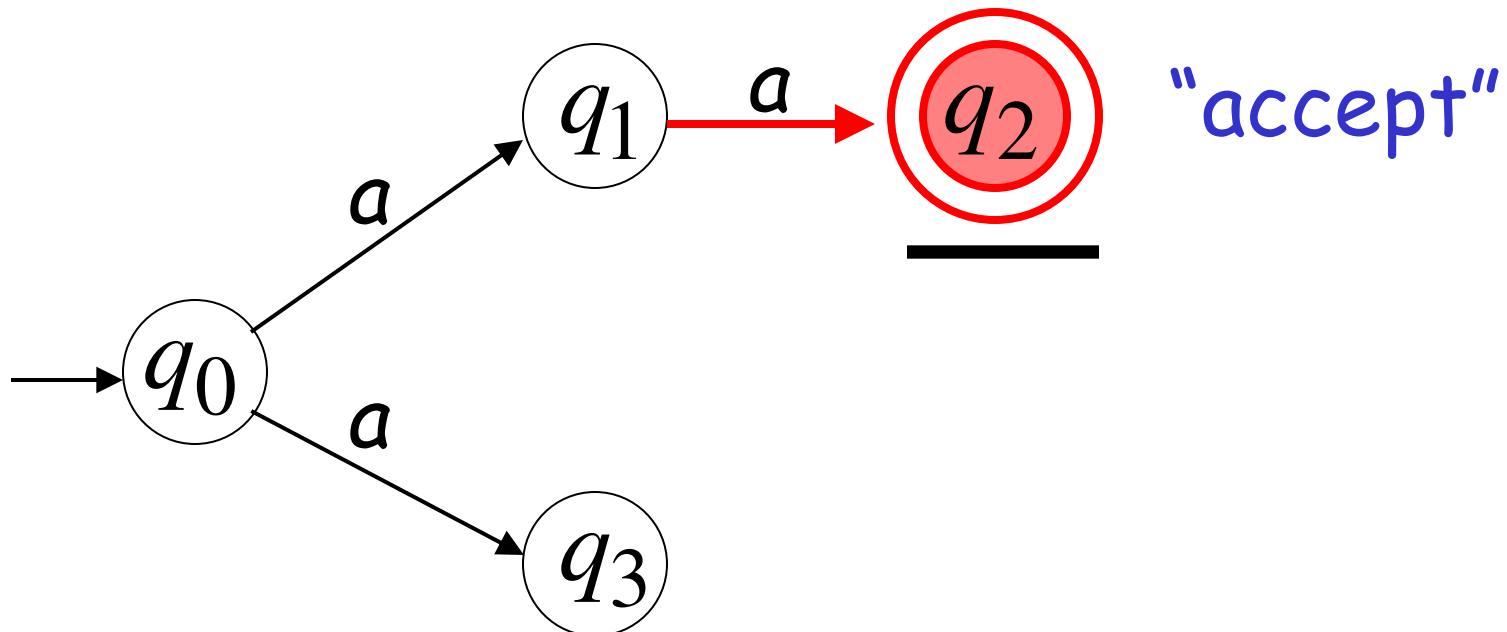
# First Choice



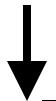
# First Choice



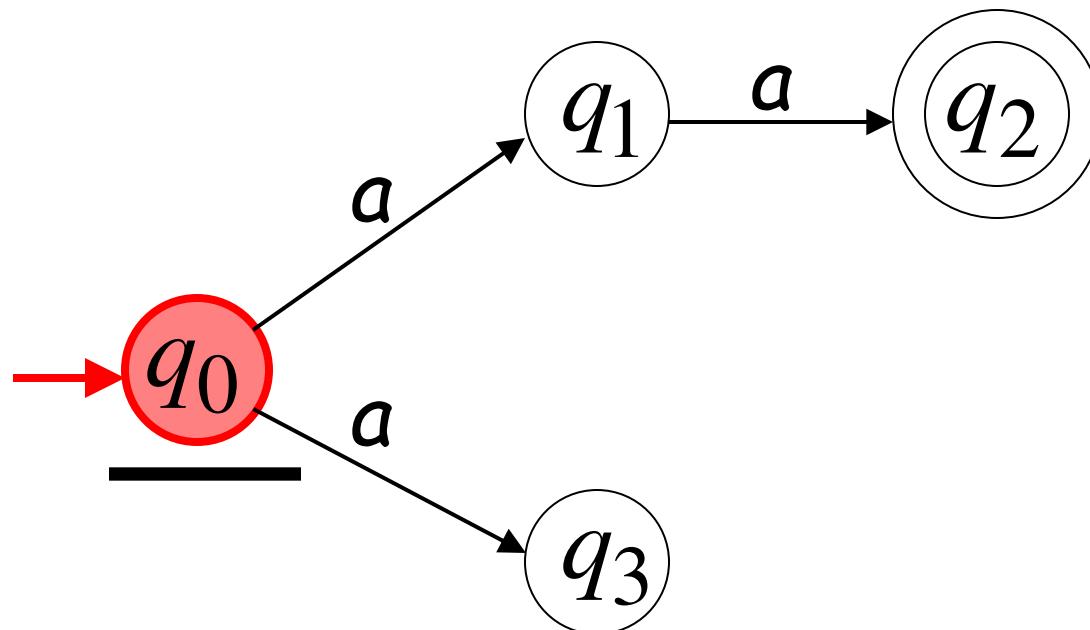
All input is consumed



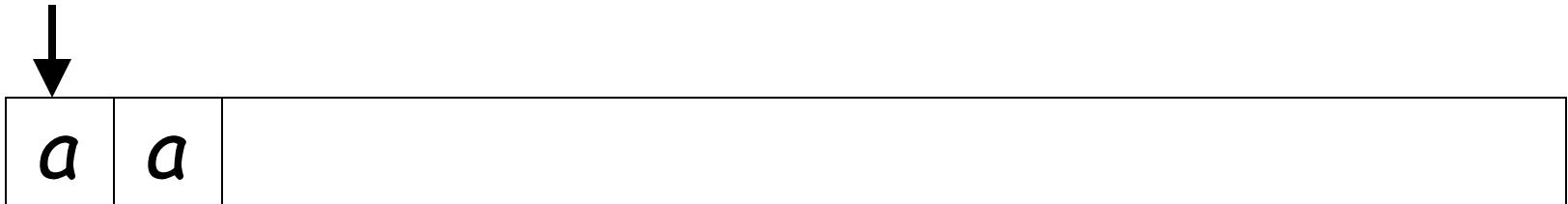
# Second Choice



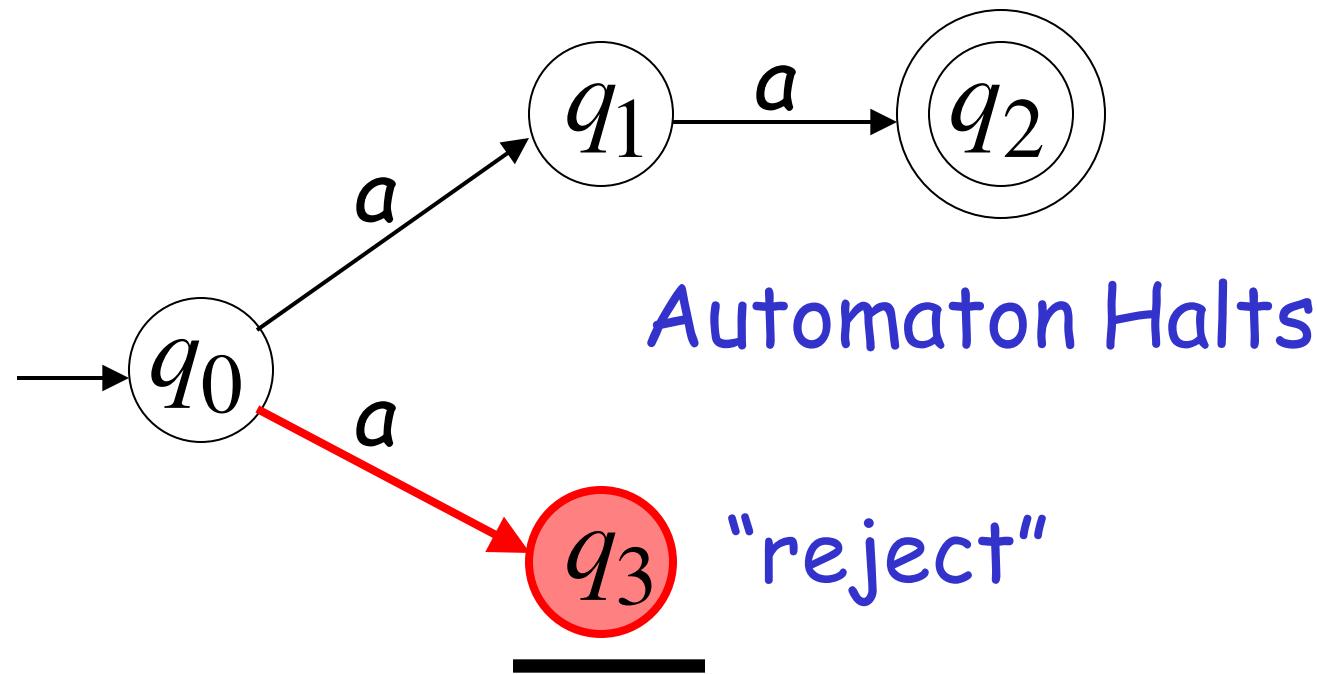
|     |     |  |
|-----|-----|--|
| $a$ | $a$ |  |
|-----|-----|--|



# Second Choice



Input cannot be consumed

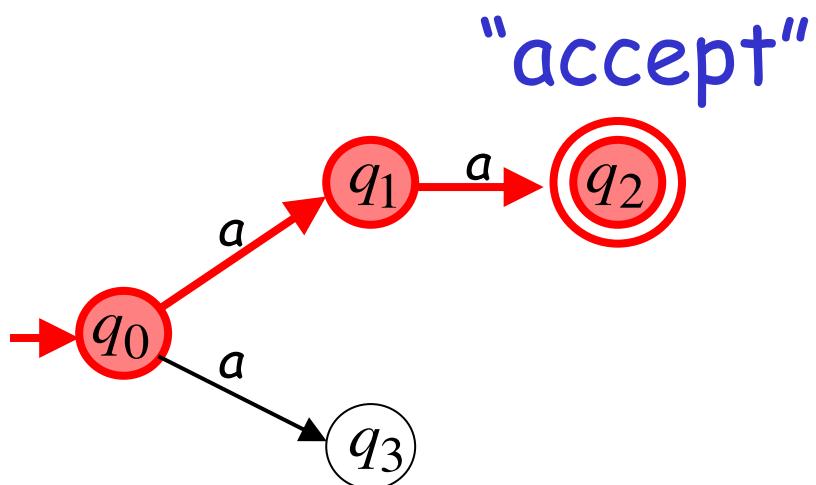


An NFA accepts a string:

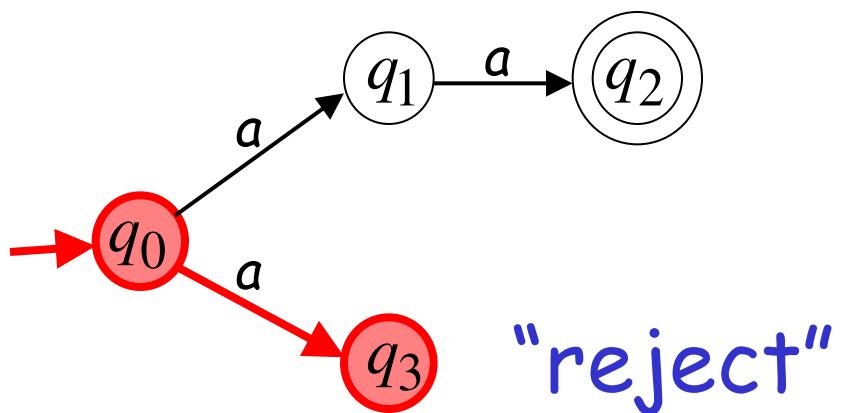
if there is a computation of the NFA  
that accepts the string

i.e., all the input string is processed and the  
automaton is in an accepting state

*aa* is accepted by the NFA:

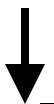


because this computation accepts *aa*

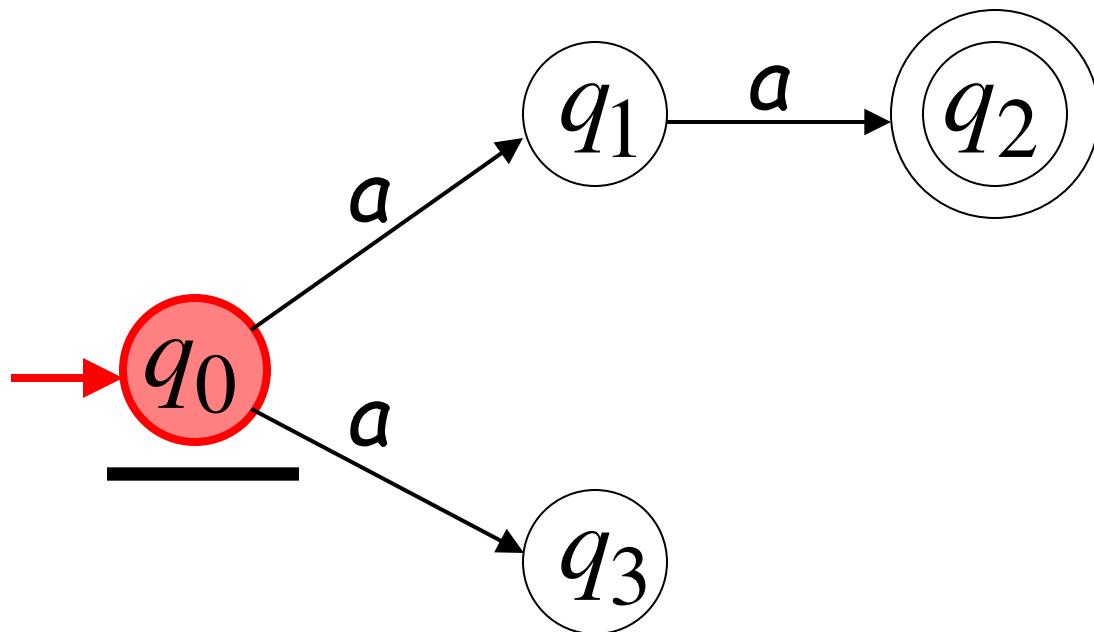


this computation is ignored

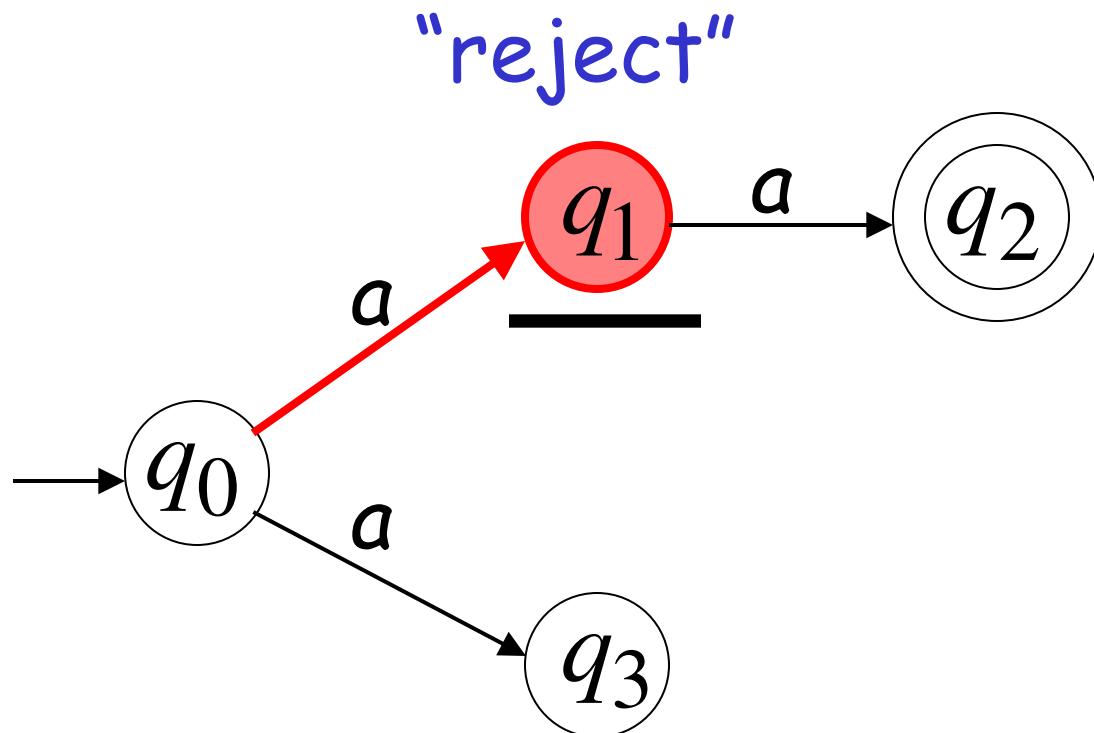
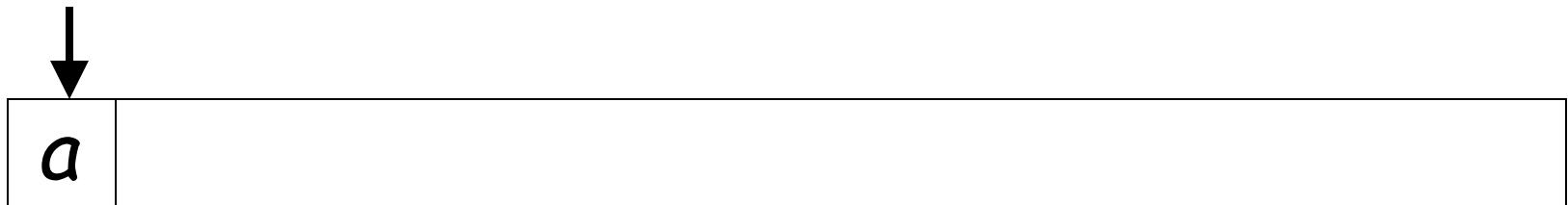
# Rejection example



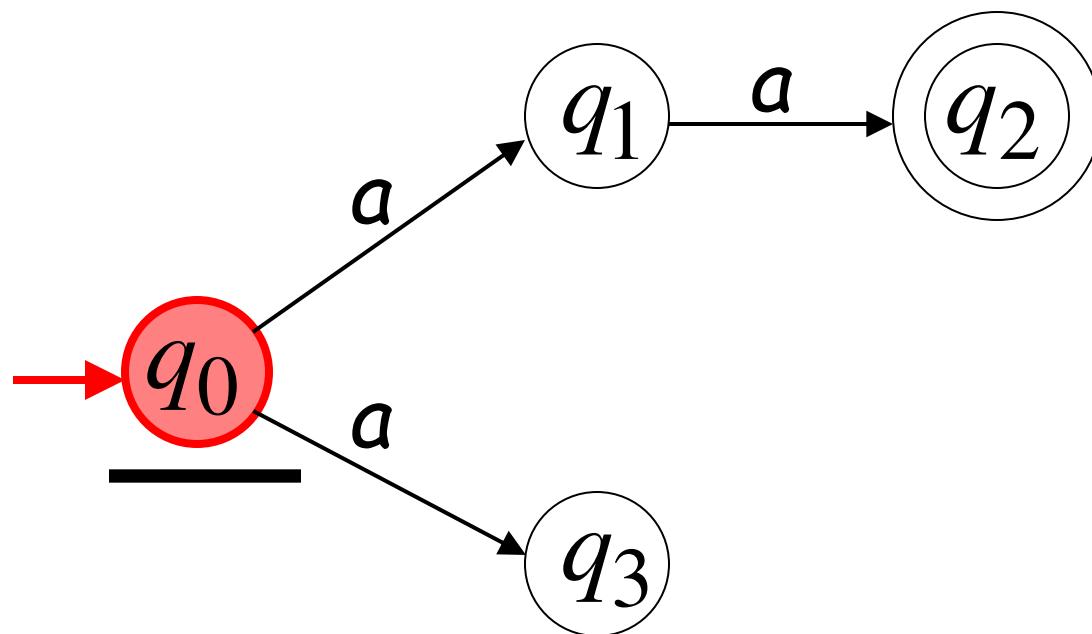
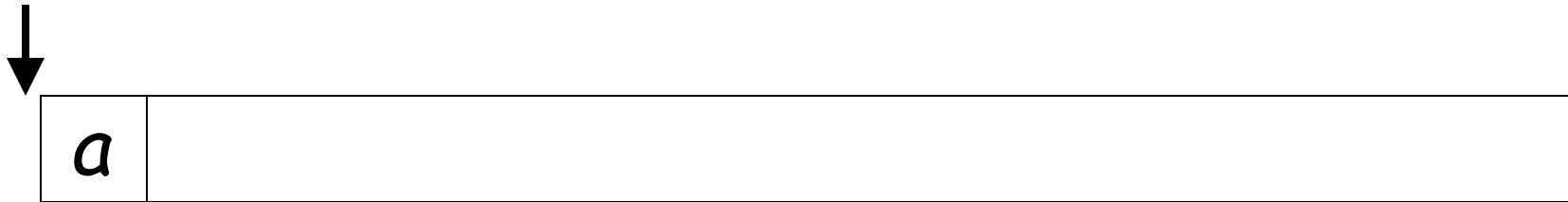
|     |  |
|-----|--|
| $a$ |  |
|-----|--|



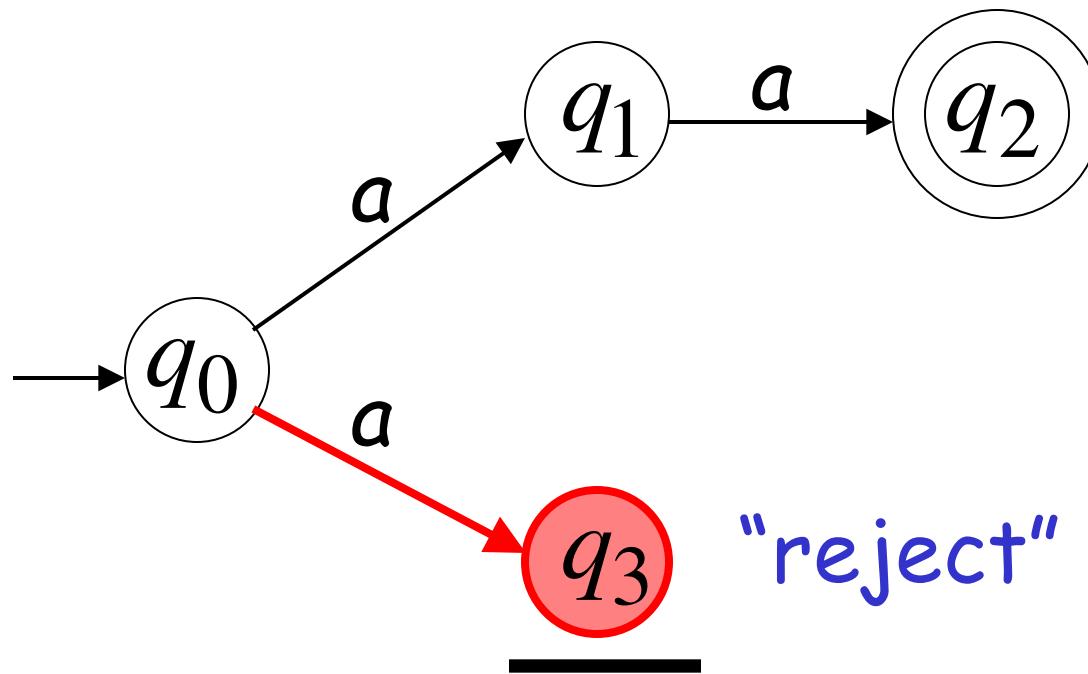
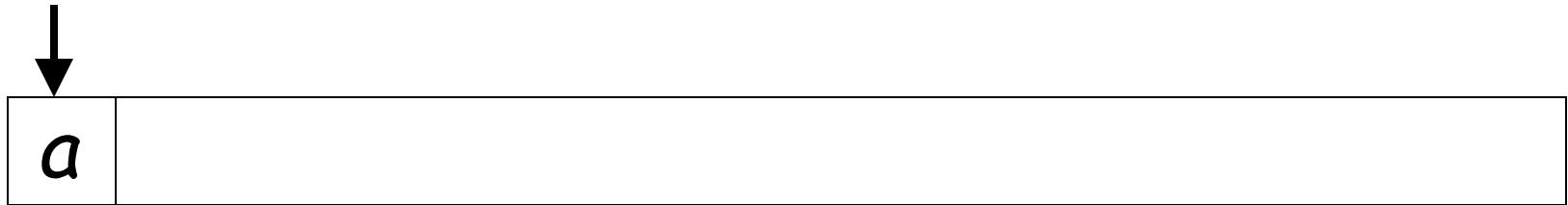
# First Choice



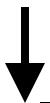
# Second Choice



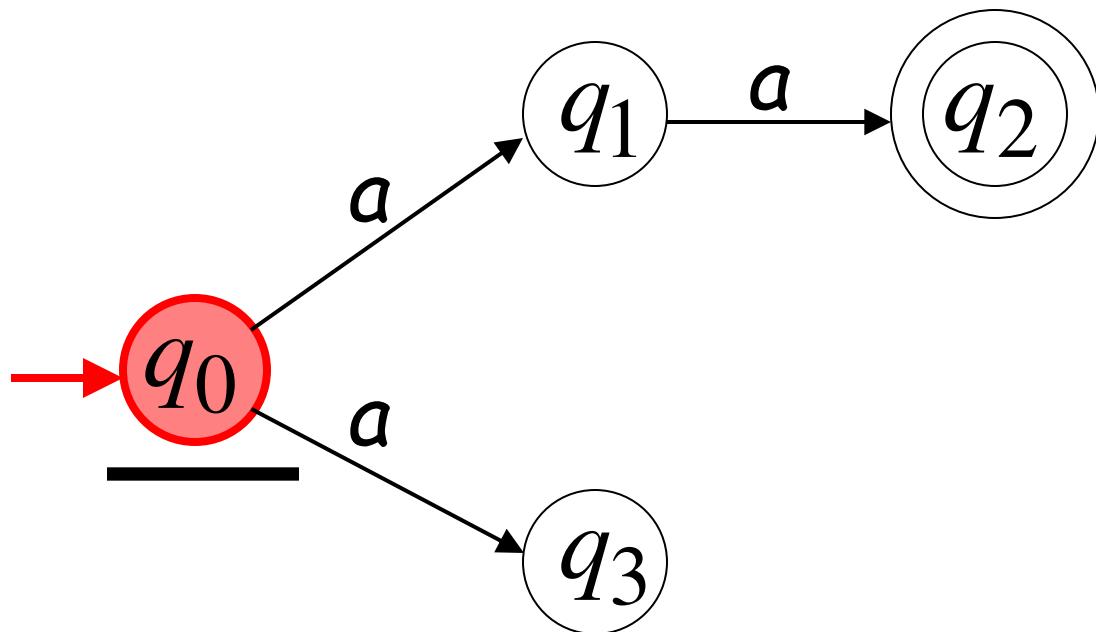
# Second Choice



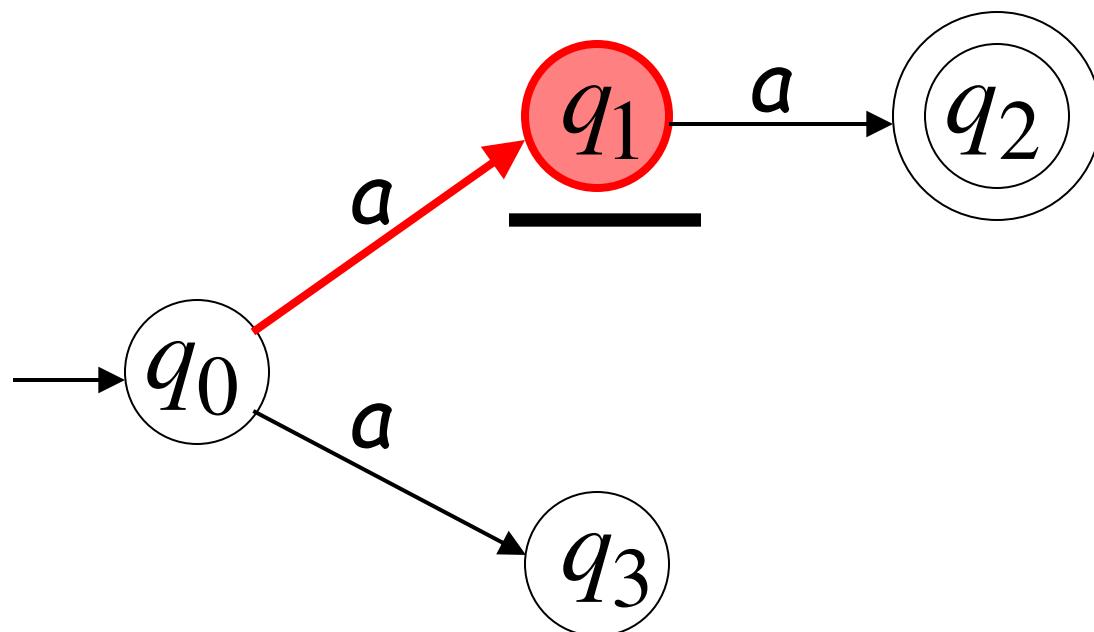
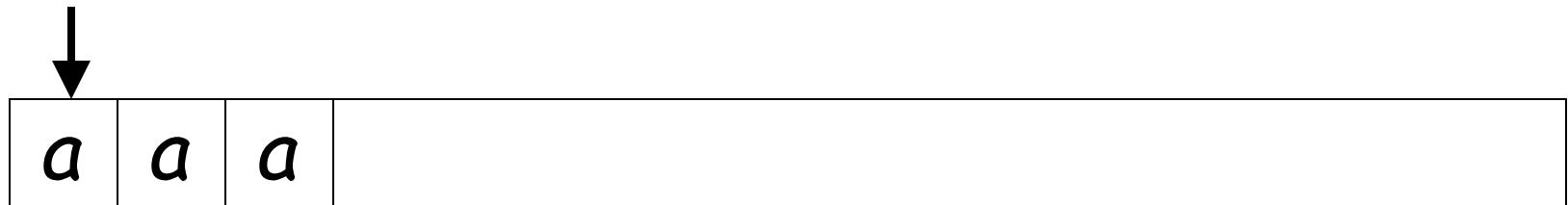
# Another Rejection example



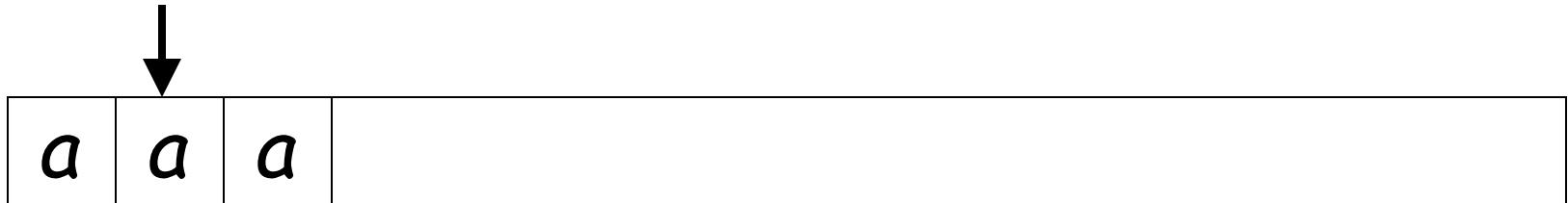
|     |     |     |  |
|-----|-----|-----|--|
| $a$ | $a$ | $a$ |  |
|-----|-----|-----|--|



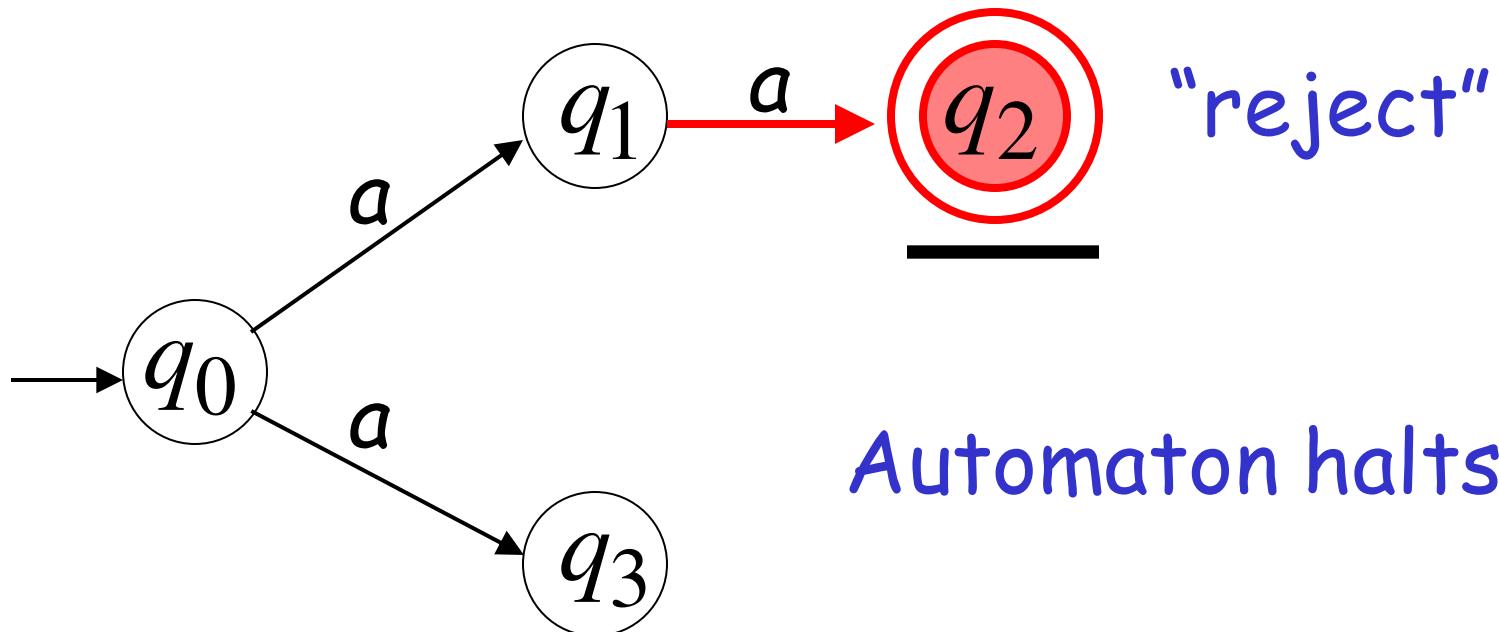
# First Choice



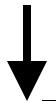
# First Choice



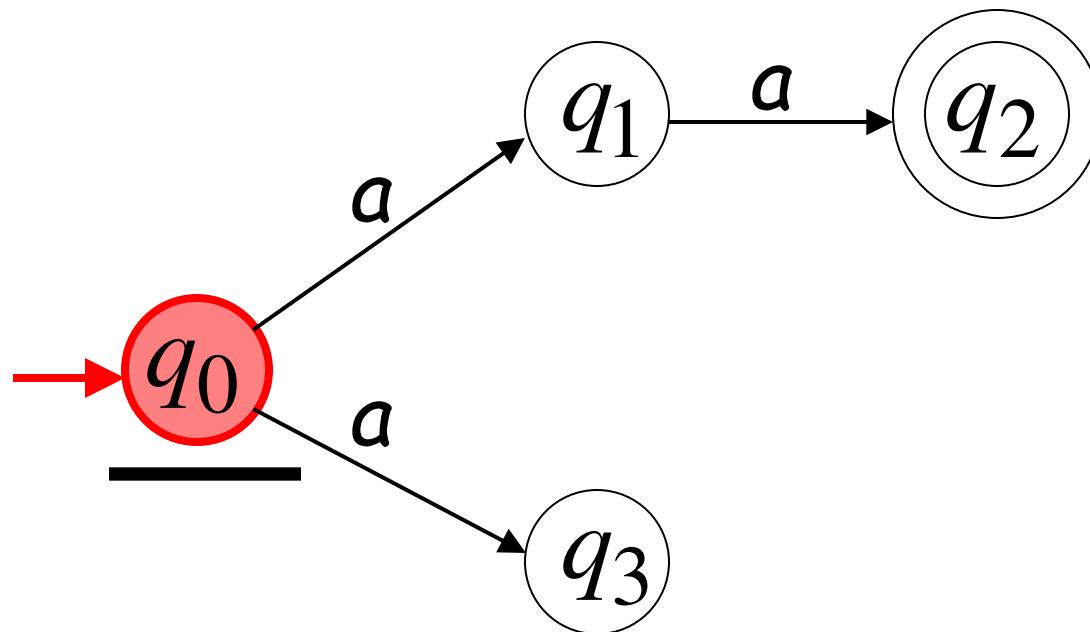
Input cannot be consumed



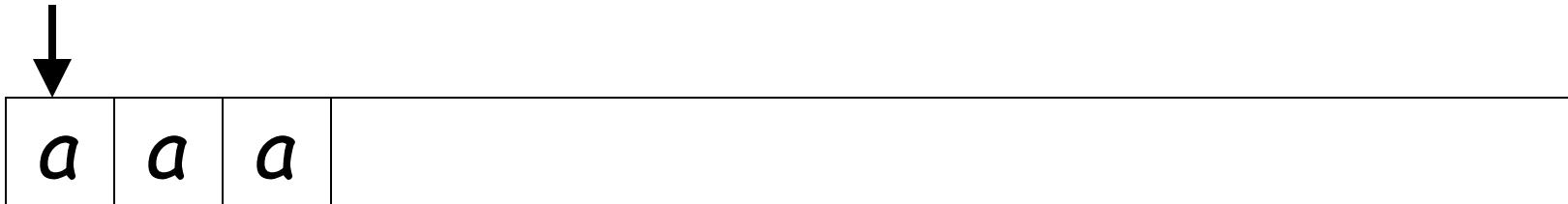
# Second Choice



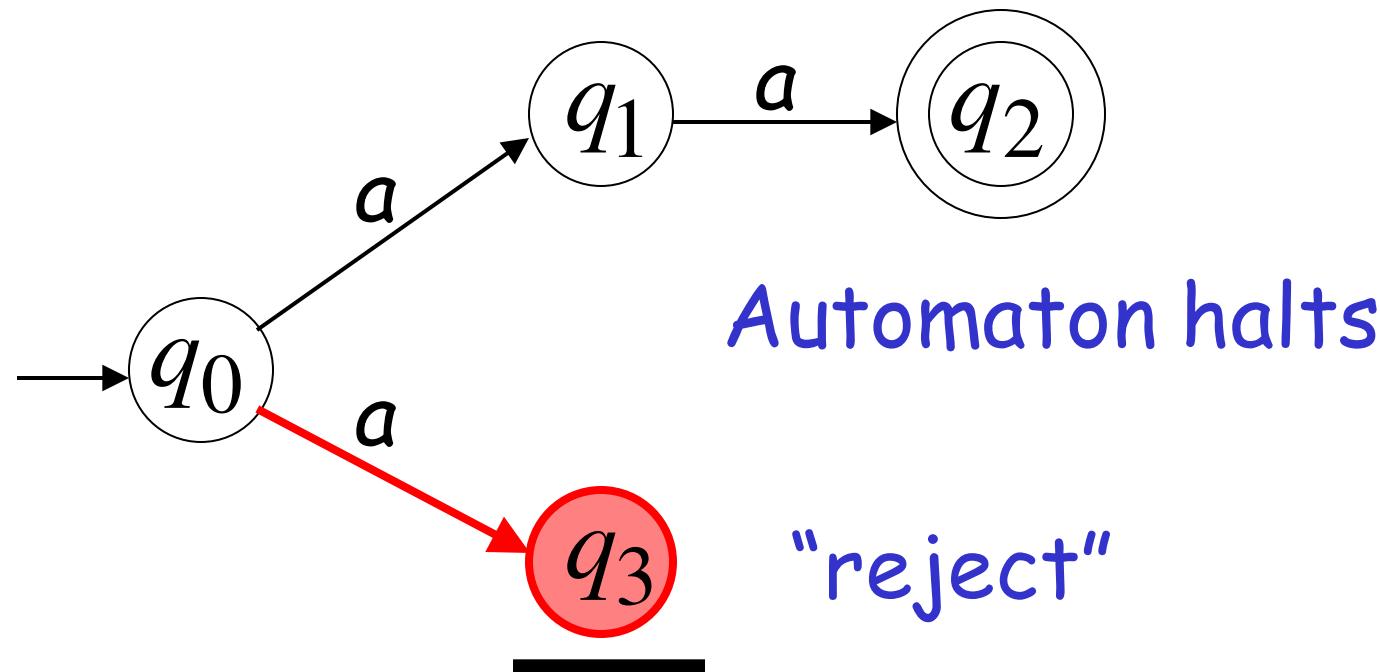
|     |     |     |  |
|-----|-----|-----|--|
| $a$ | $a$ | $a$ |  |
|-----|-----|-----|--|



# Second Choice



Input cannot be consumed



## An NFA rejects a string:

if there is no computation of the NFA that accepts the string.

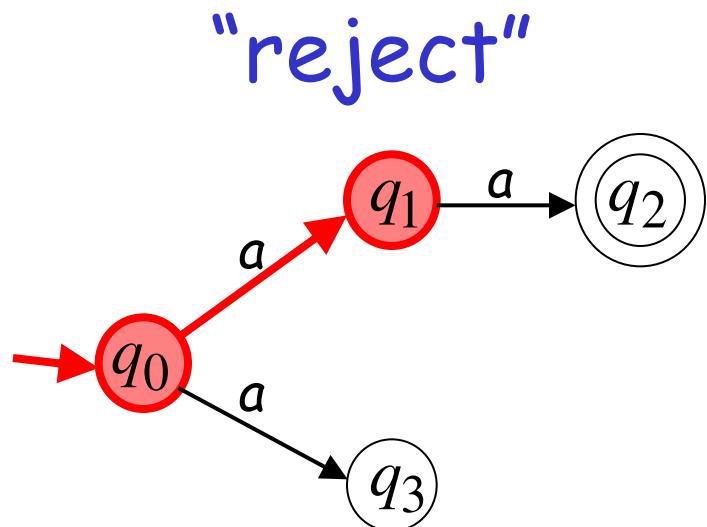
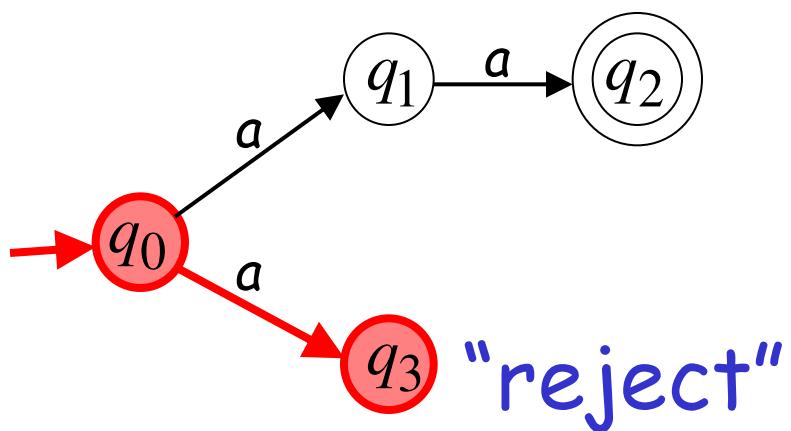
For each computation:

- All the input is consumed and the automaton is in a non final state

OR

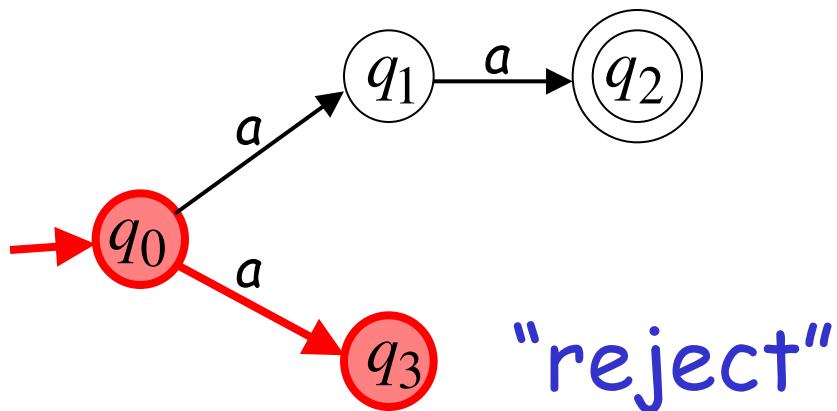
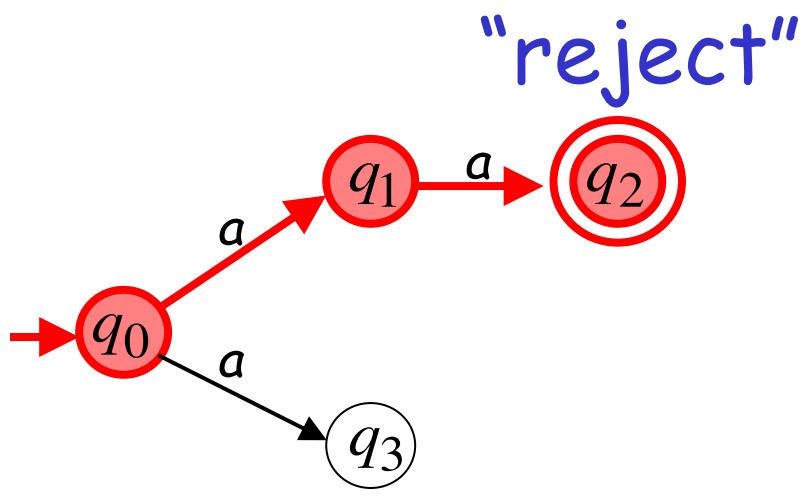
- The input cannot be consumed

a is rejected by the NFA:



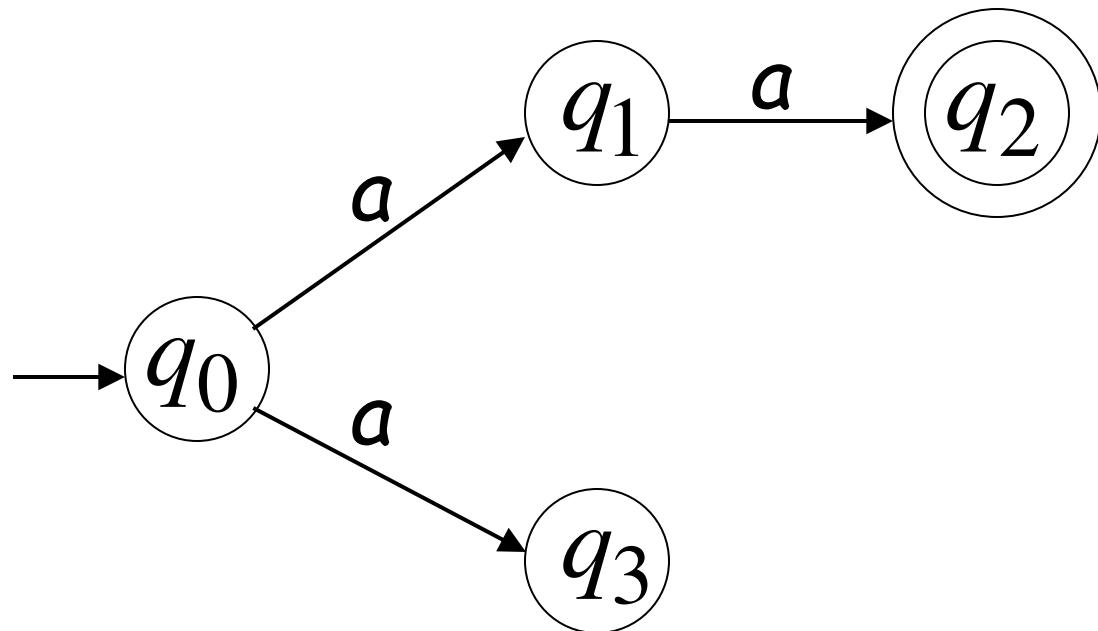
All possible computations lead to rejection

**aaa** is rejected by the NFA:

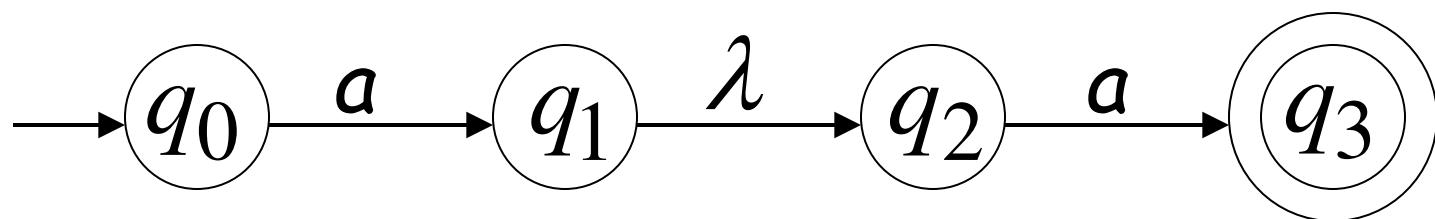


All possible computations lead to rejection

Language accepted:  $L = \{aa\}$

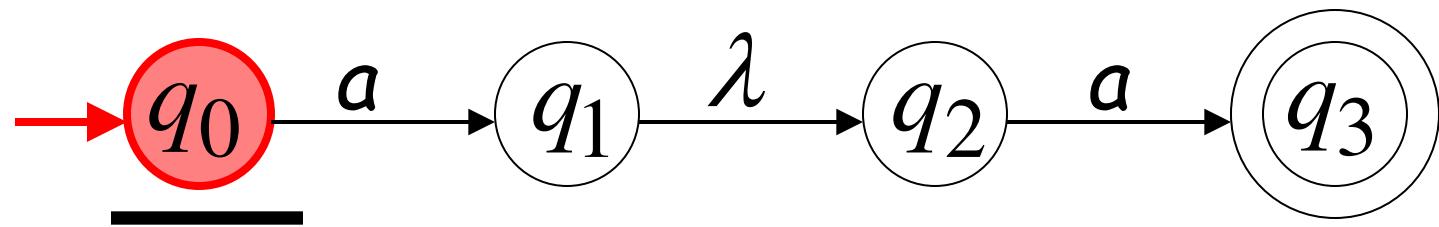


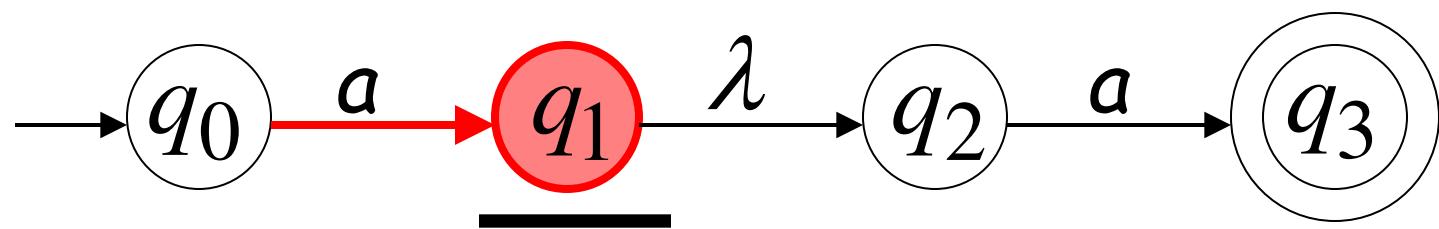
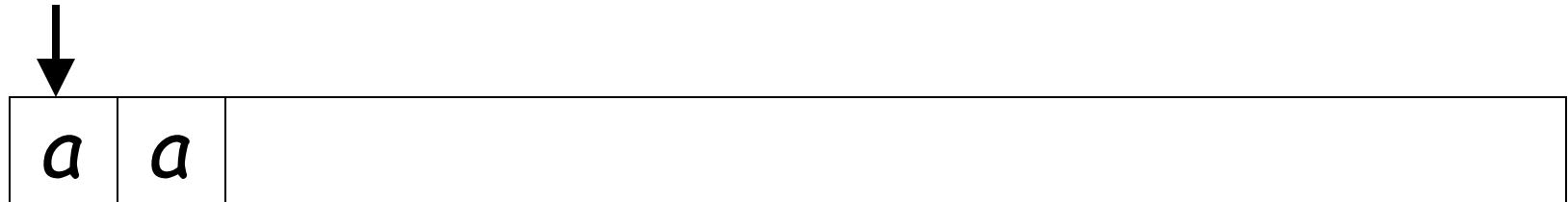
# Lambda Transitions



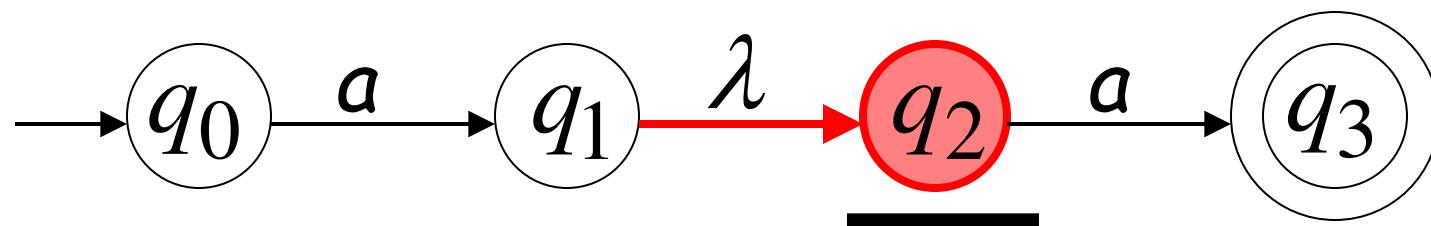
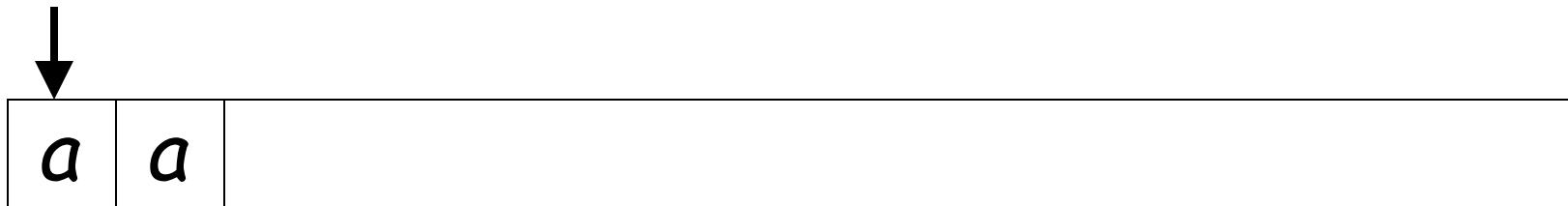


|     |     |  |
|-----|-----|--|
| $a$ | $a$ |  |
|-----|-----|--|

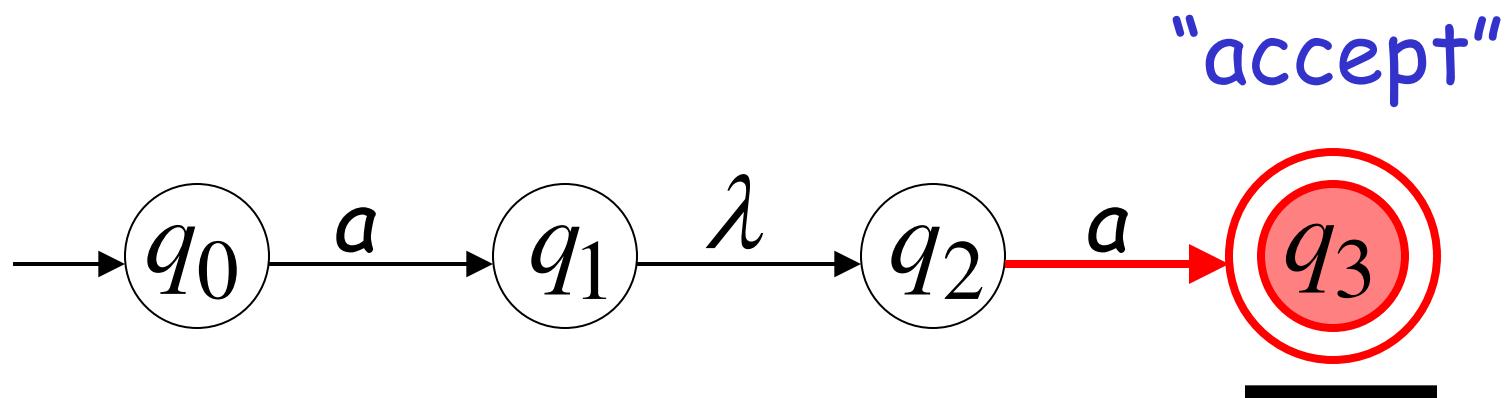




input tape head does not move

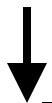


all input is consumed

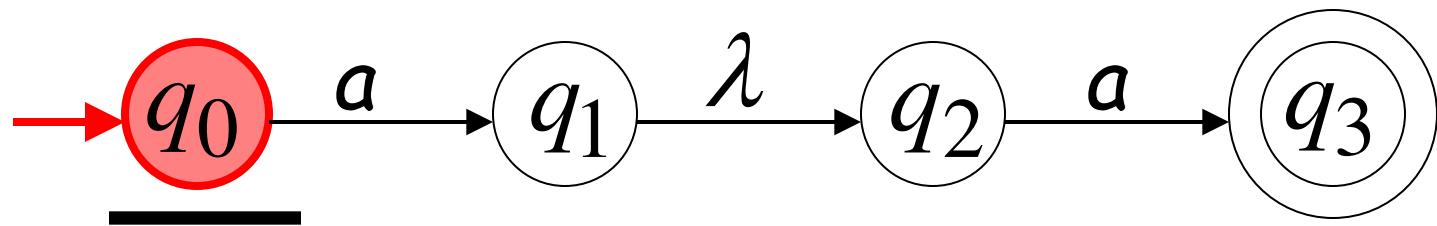


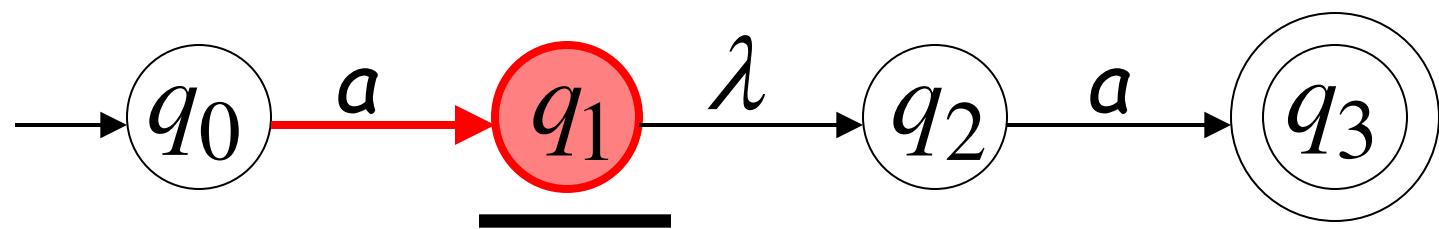
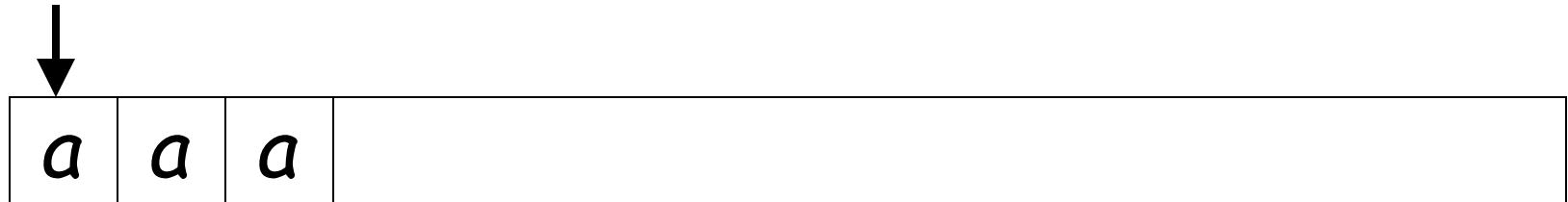
String  $aa$  is accepted

# Rejection Example

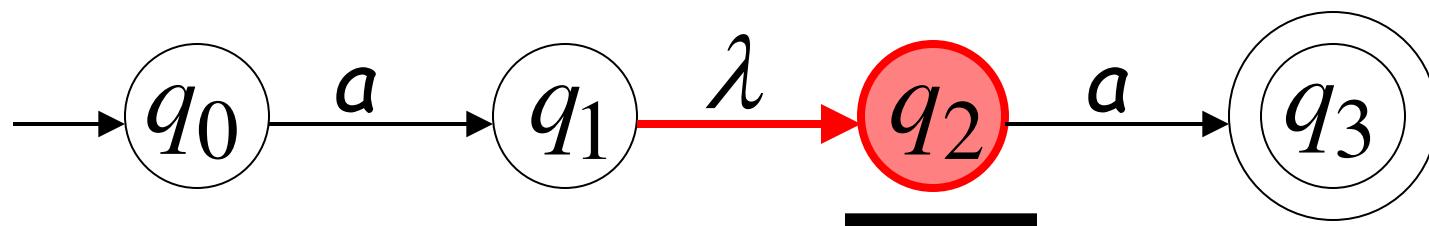
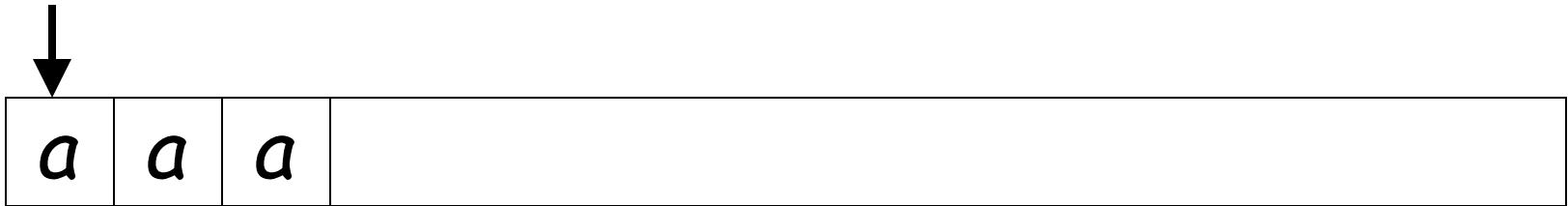


|     |     |     |  |
|-----|-----|-----|--|
| $a$ | $a$ | $a$ |  |
|-----|-----|-----|--|





(read head doesn't move)

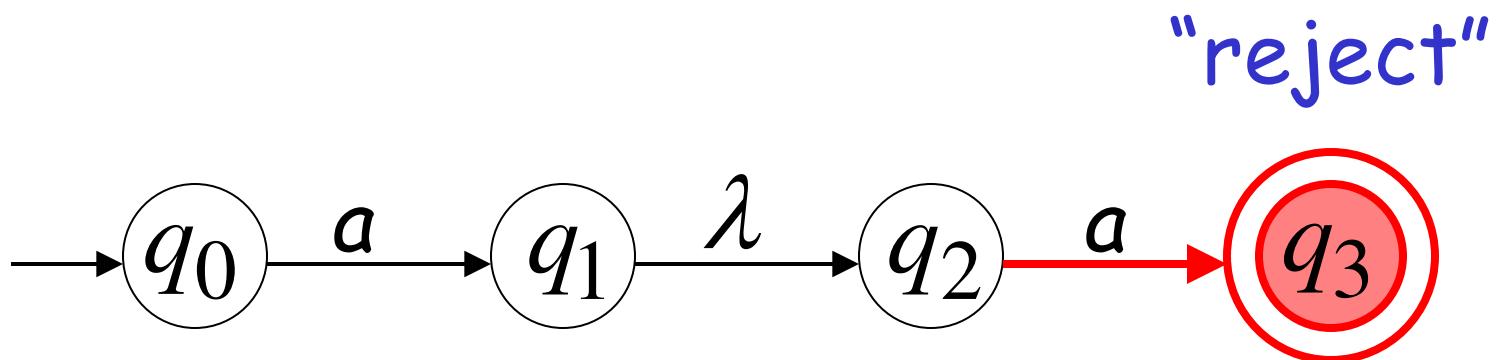


# Input cannot be consumed



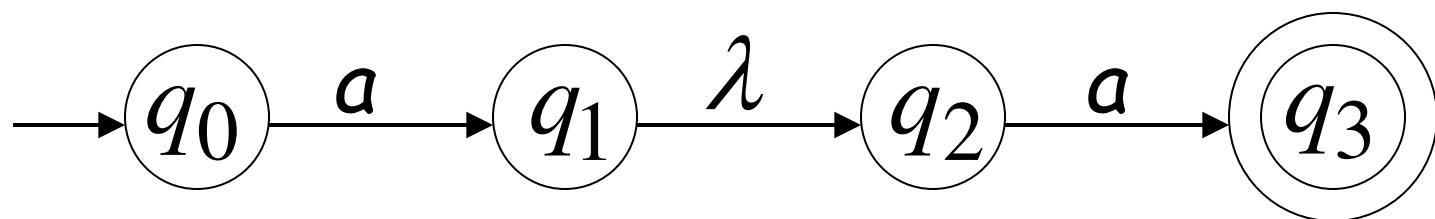
|   |   |   |  |
|---|---|---|--|
| a | a | a |  |
|---|---|---|--|

Automaton halts

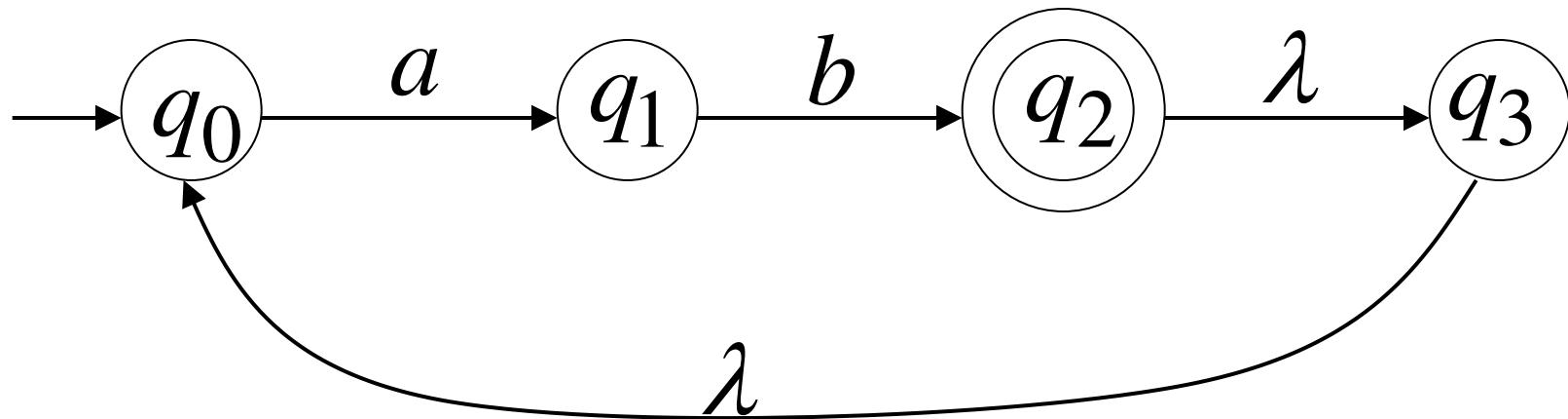


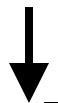
String aaa is rejected

Language accepted:  $L = \{aa\}$

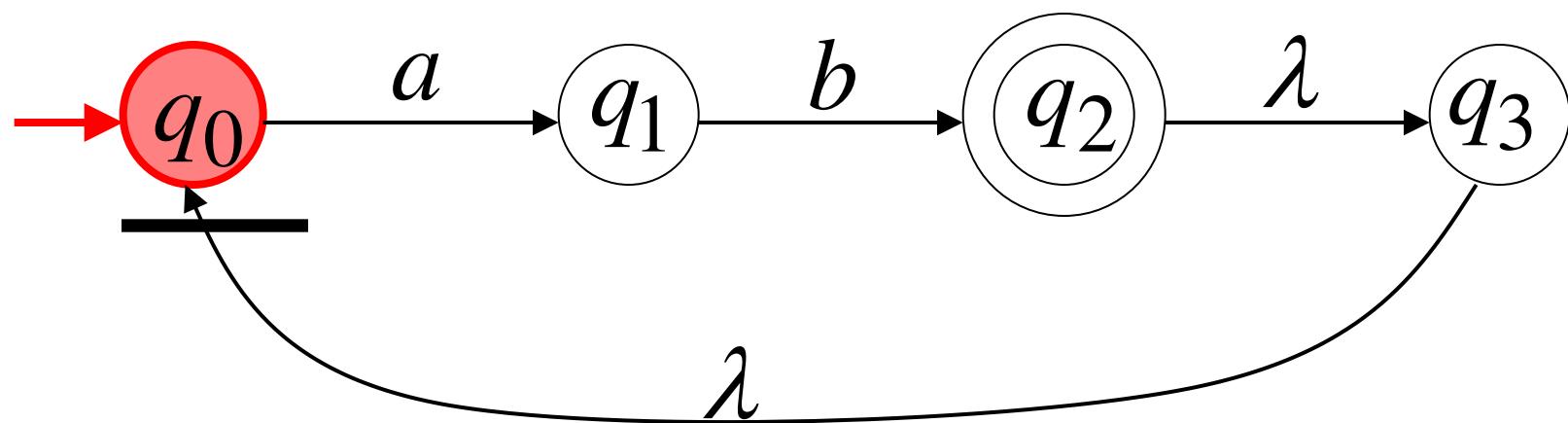


# Another NFA Example

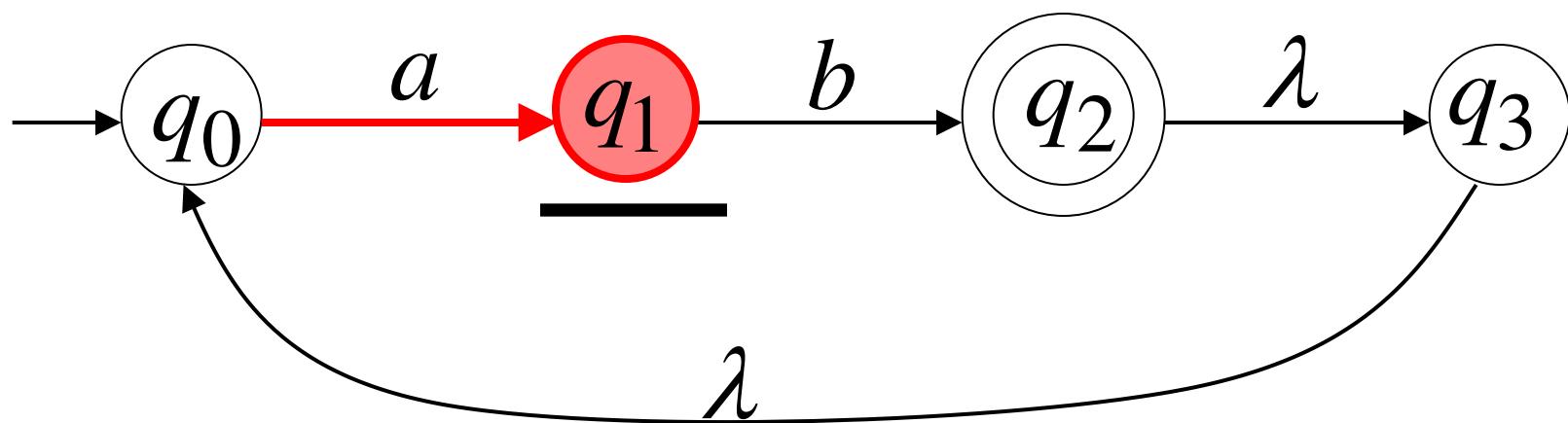




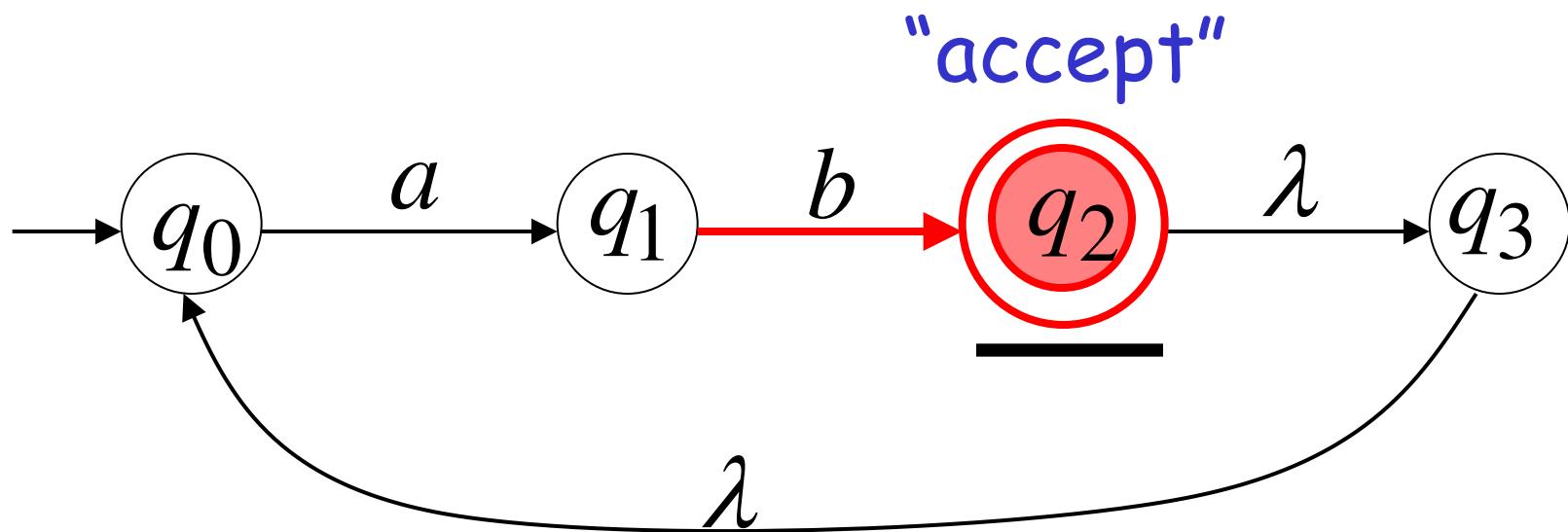
|     |     |  |
|-----|-----|--|
| $a$ | $b$ |  |
|-----|-----|--|



|     |     |  |
|-----|-----|--|
| $a$ | $b$ |  |
|-----|-----|--|



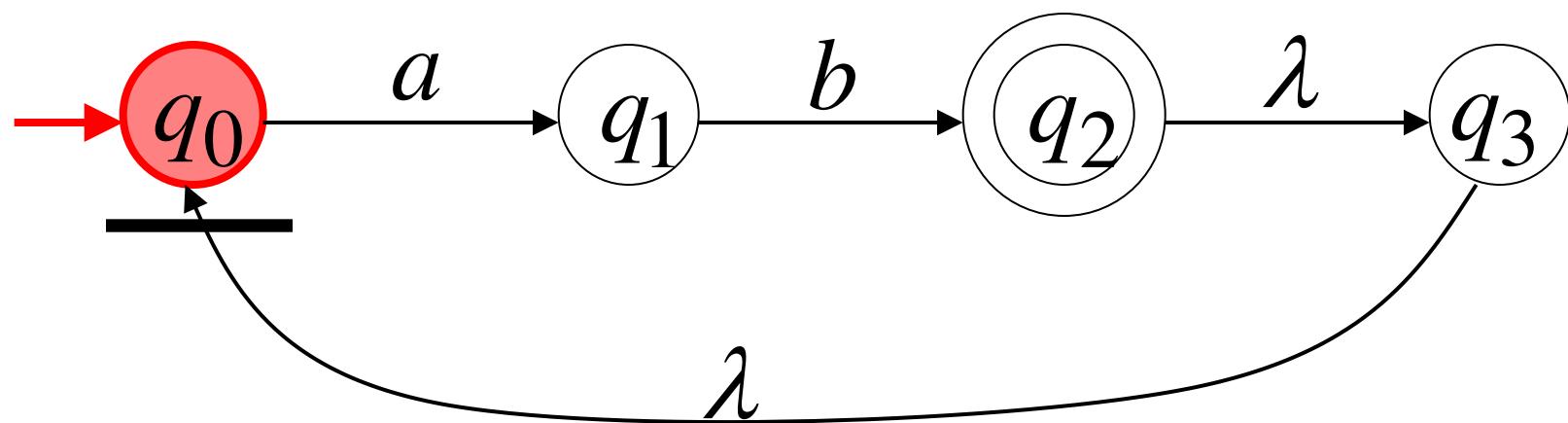
|     |     |  |
|-----|-----|--|
| $a$ | $b$ |  |
|-----|-----|--|

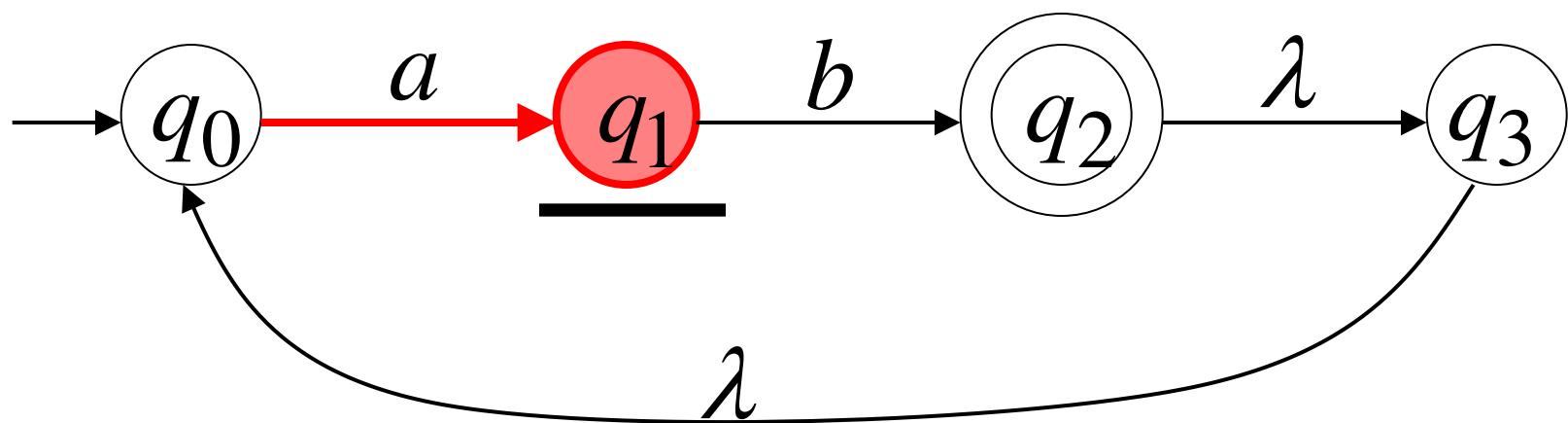
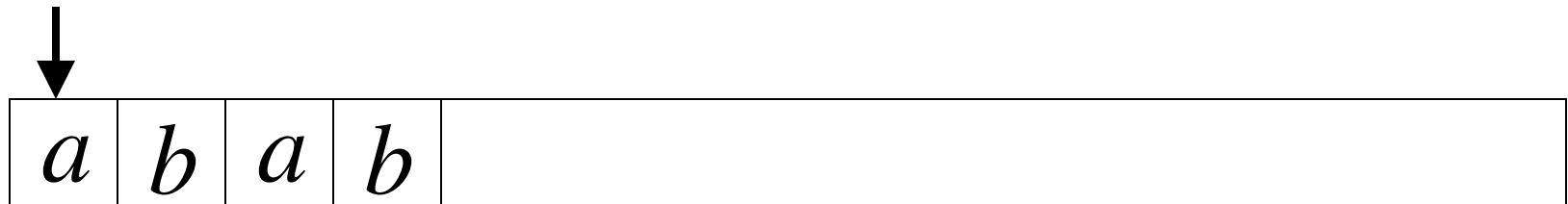


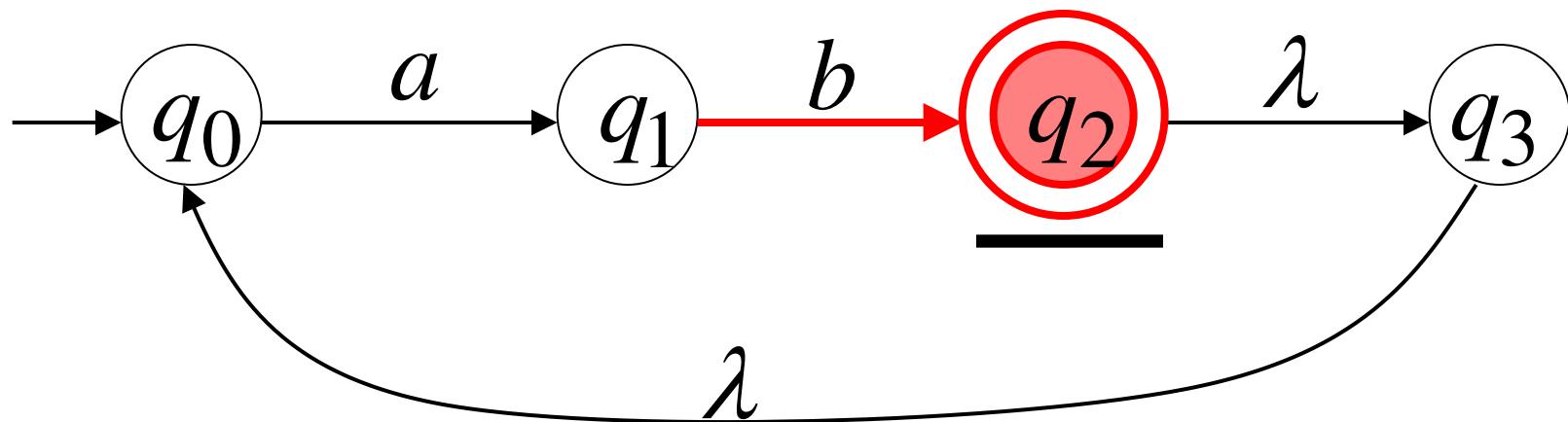
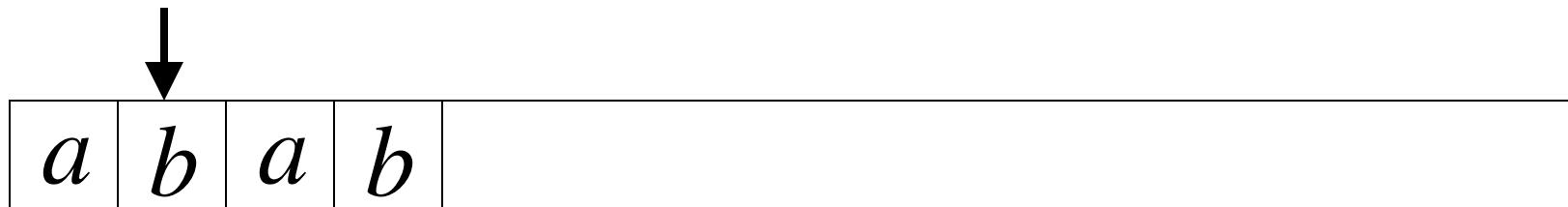
## Another String

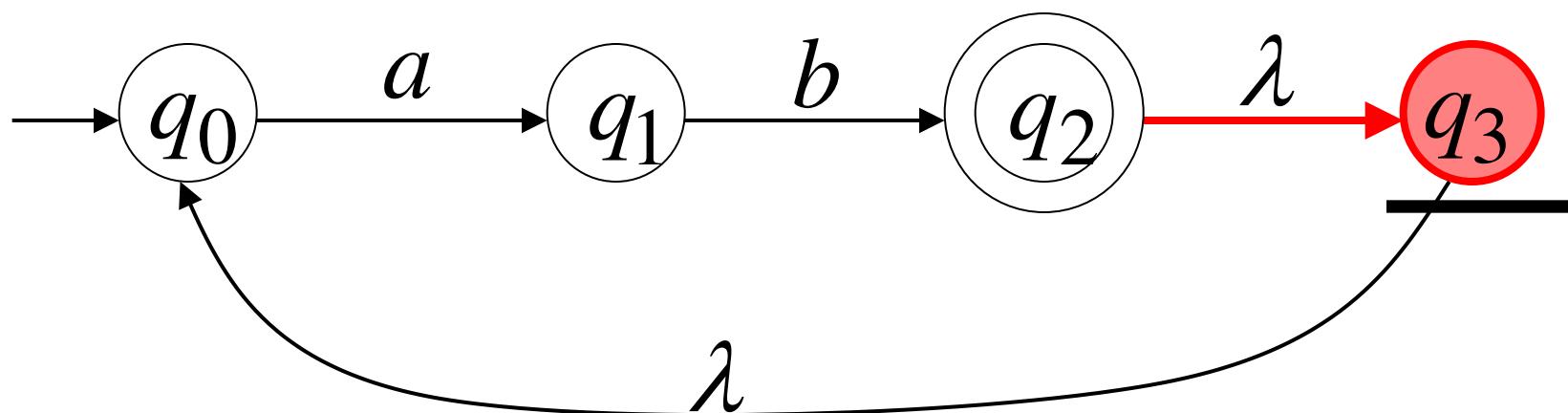
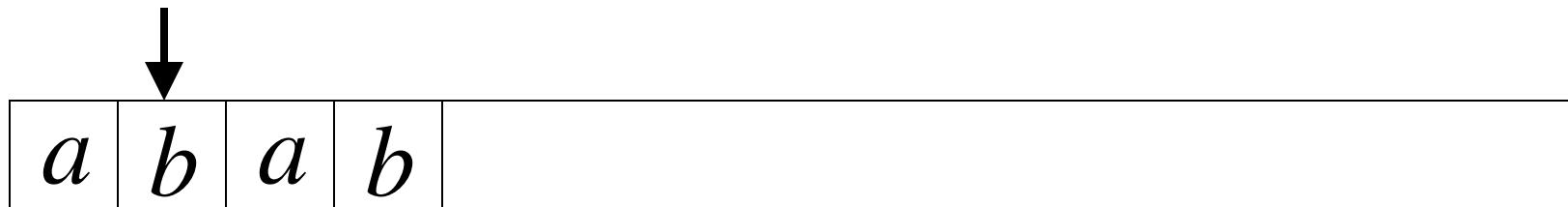


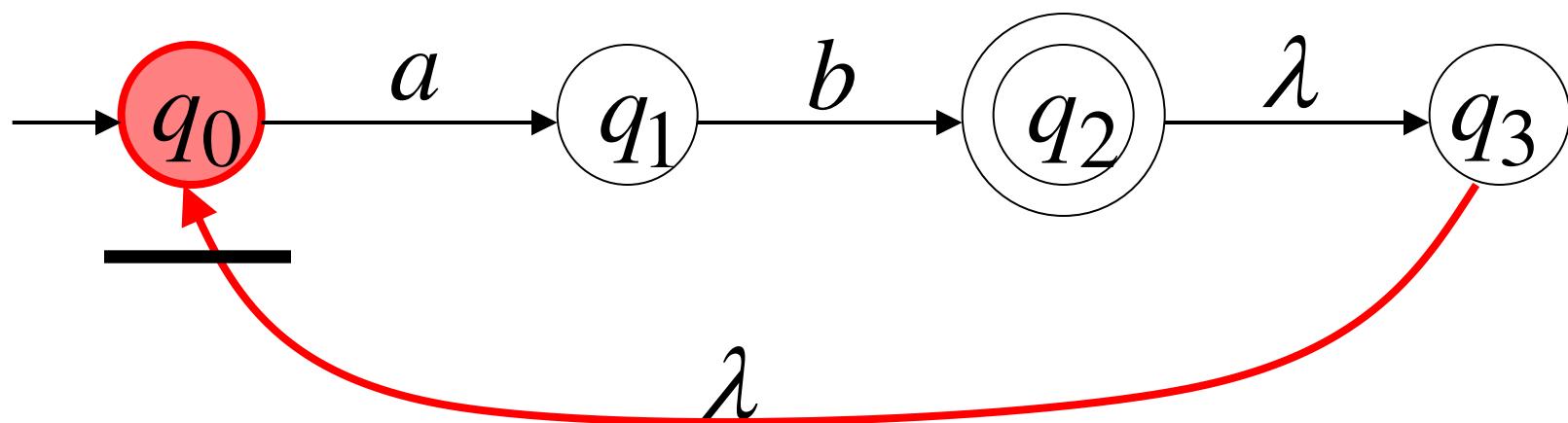
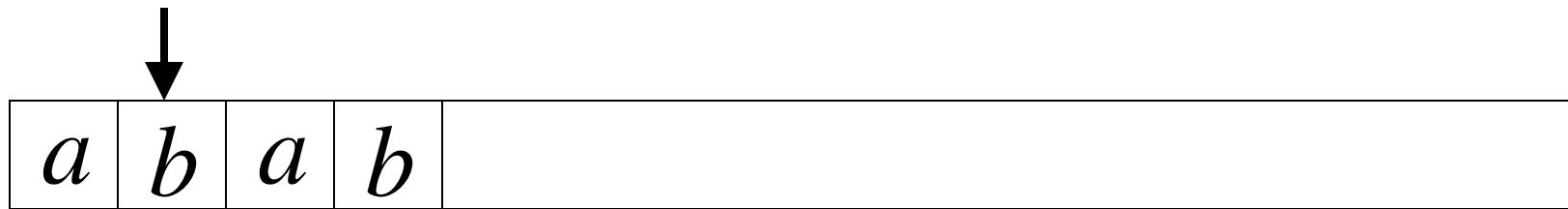
|     |     |     |     |  |
|-----|-----|-----|-----|--|
| $a$ | $b$ | $a$ | $b$ |  |
|-----|-----|-----|-----|--|

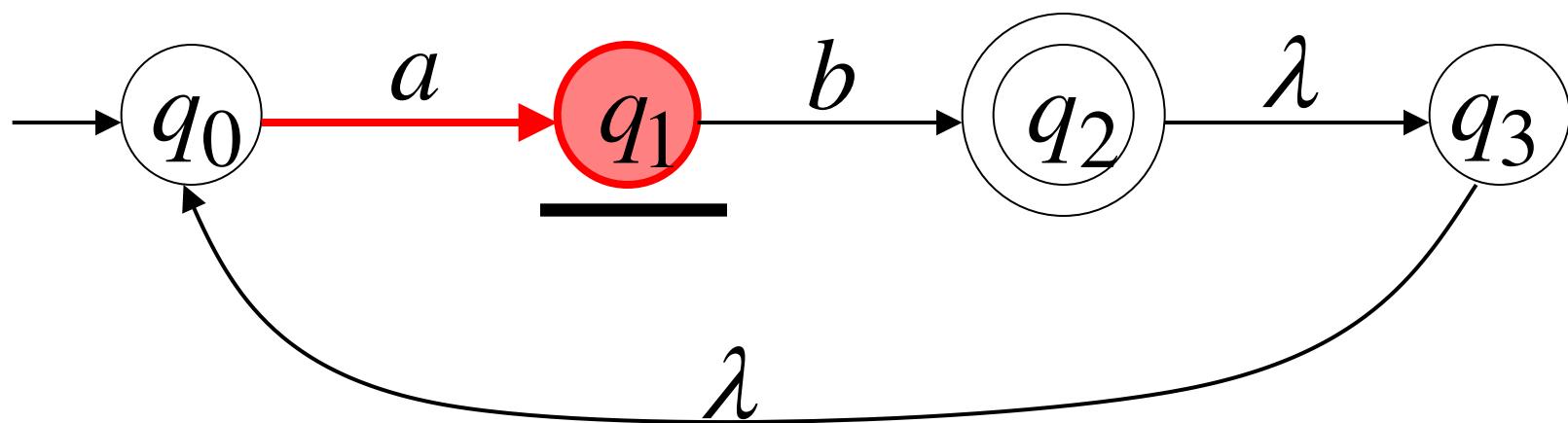
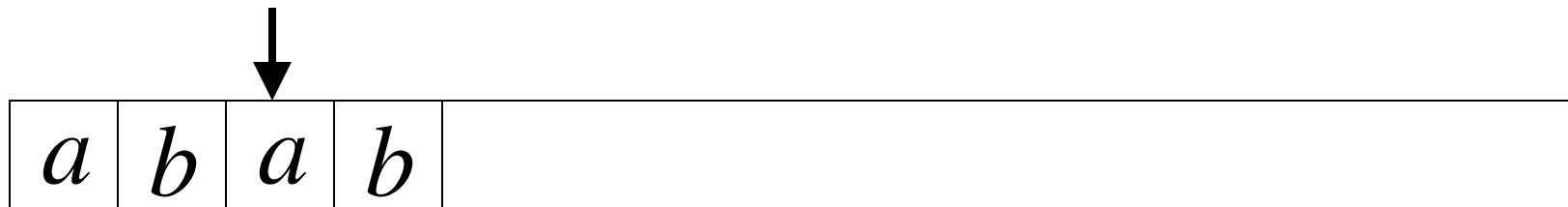


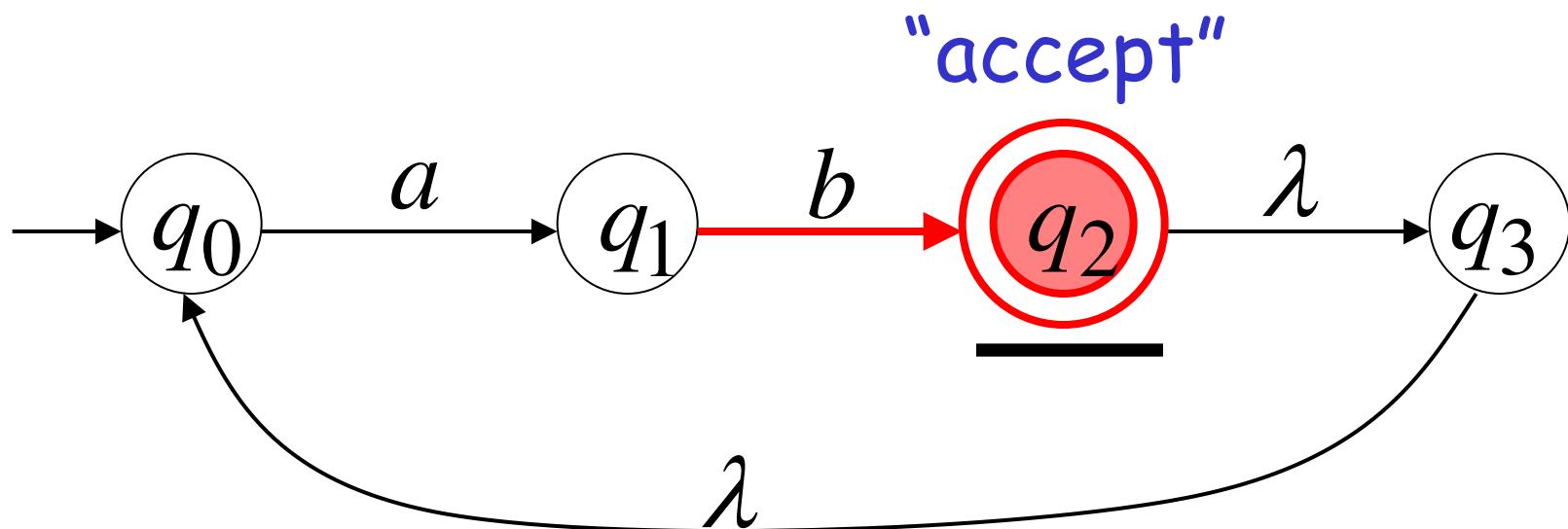
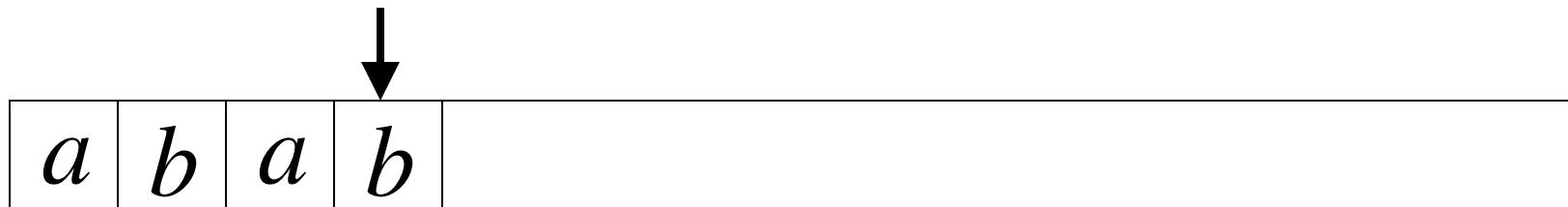








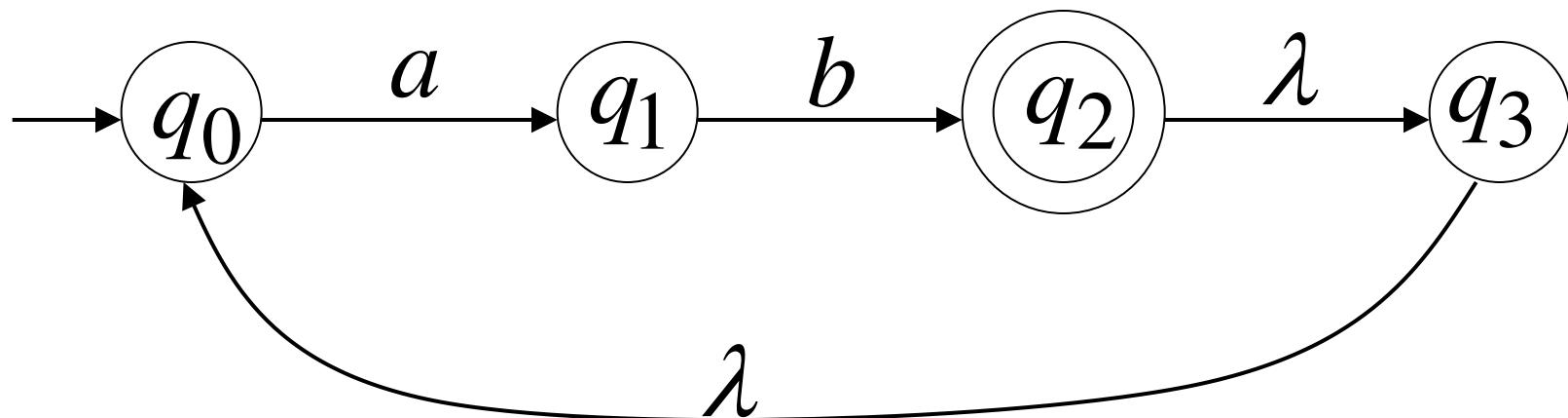




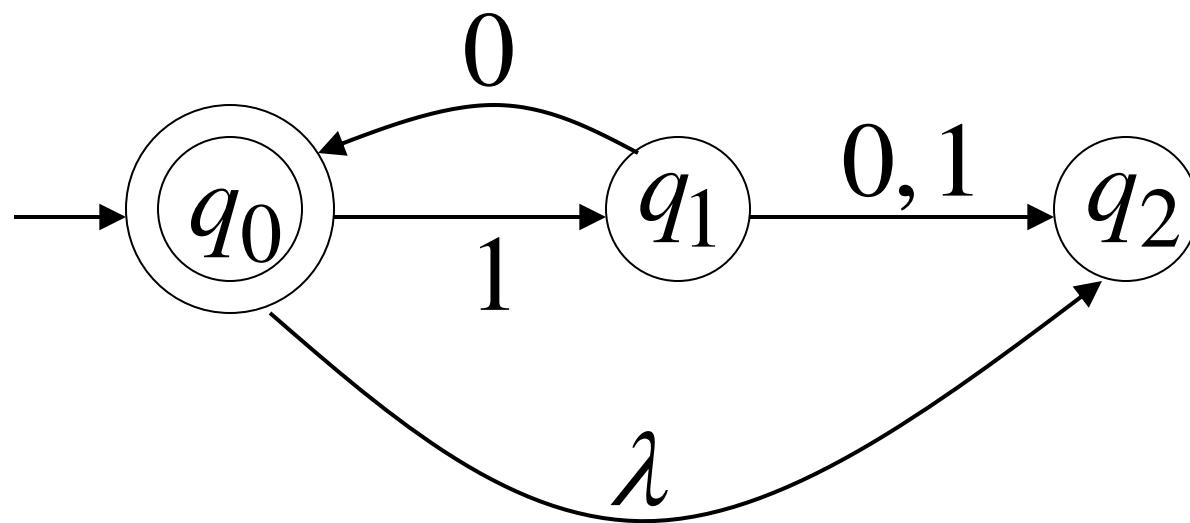
Language accepted

$$L = \{ab, abab, ababab, \dots\}$$

$$= \{ab\}^+$$

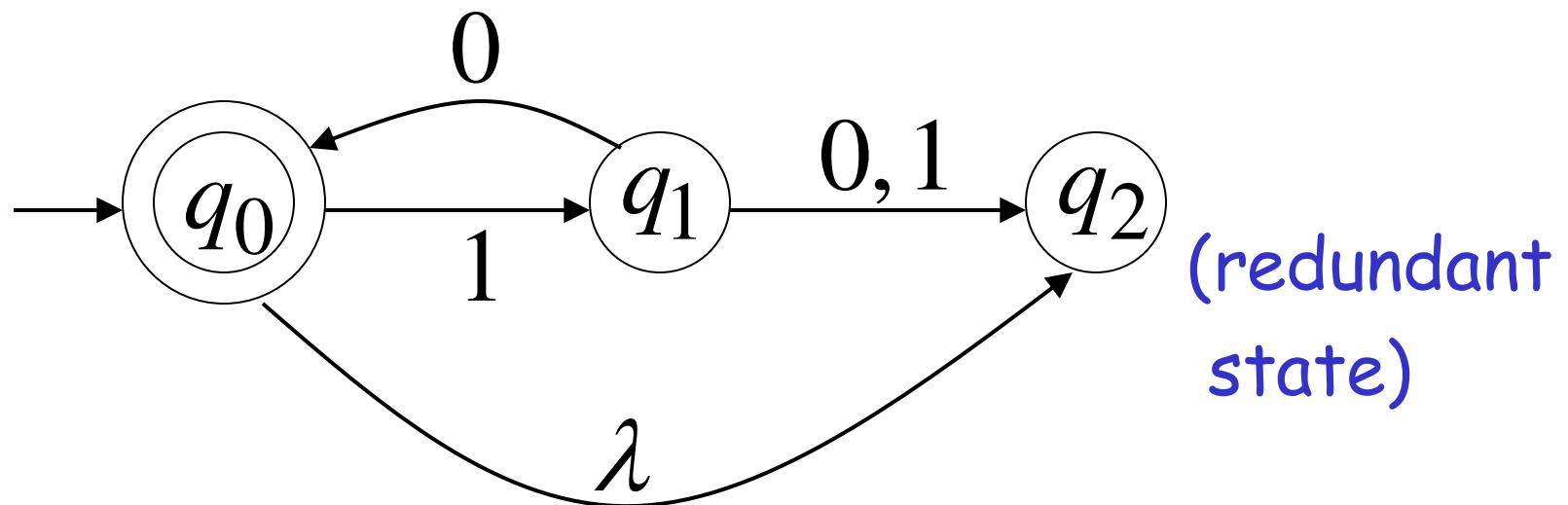


# Another NFA Example



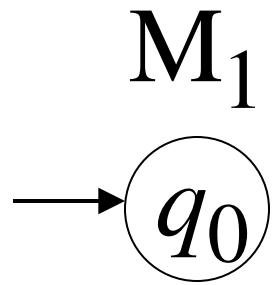
Language accepted

$$\begin{aligned}L(M) &= \{\lambda, 10, 1010, 101010, \dots\} \\&= \{10\}^*\end{aligned}$$

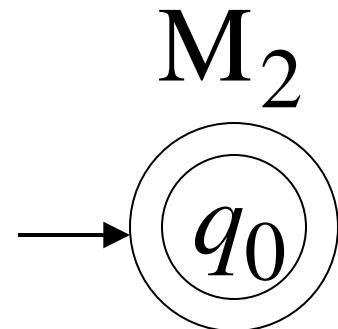


## Remarks:

- The  $\lambda$  symbol never appears on the input tape
- Simple automata:

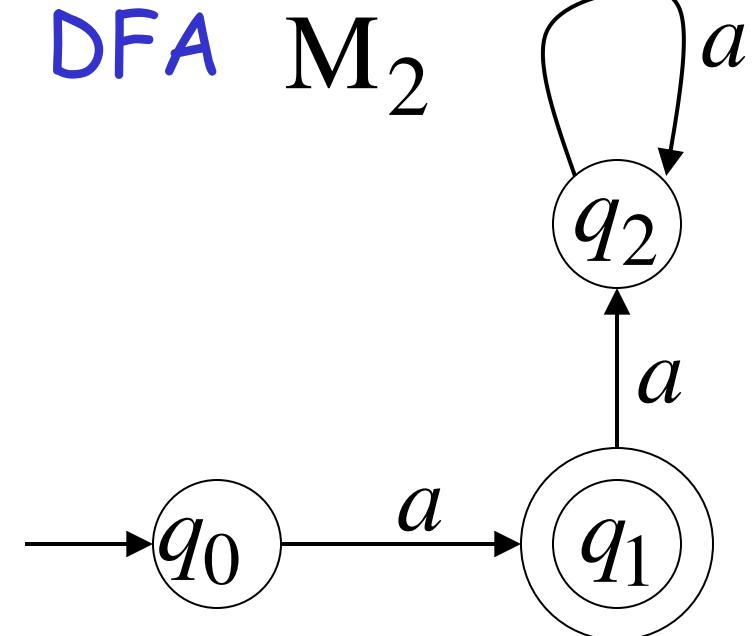
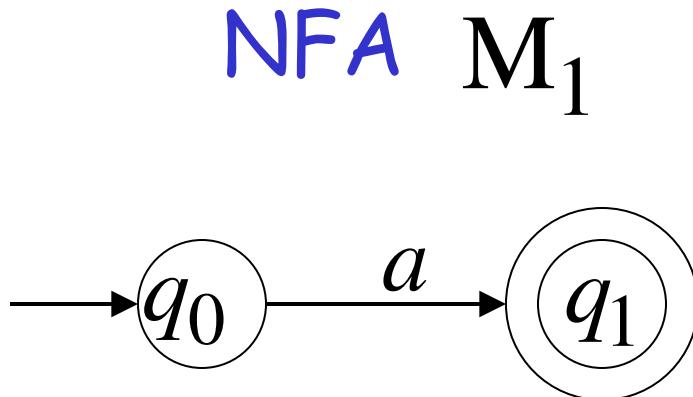


$$L(M_1) = \{ \}$$



$$L(M_2) = \{ \lambda \}$$

- NFAs are interesting because we can express languages easier than DFAs



$$L(M_1) = \{a\}$$

$$L(M_2) = \{a\}$$

# Formal Definition of NFAs

$$M = (Q, \Sigma, \delta, q_0, F)$$

$Q$ : Set of states, i.e.  $\{q_0, q_1, q_2\}$

$\Sigma$ : Input alphabet, i.e.  $\{a, b\}$        $\lambda \notin \Sigma$

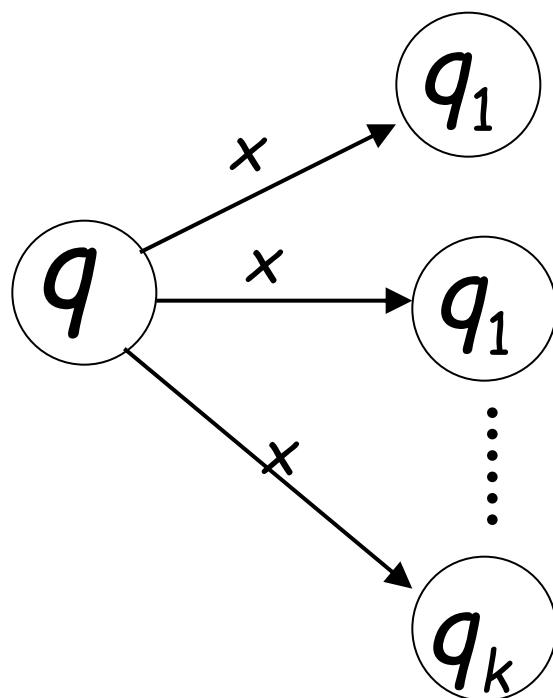
$\delta$ : Transition function

$q_0$ : Initial state

$F$ : Accepting states

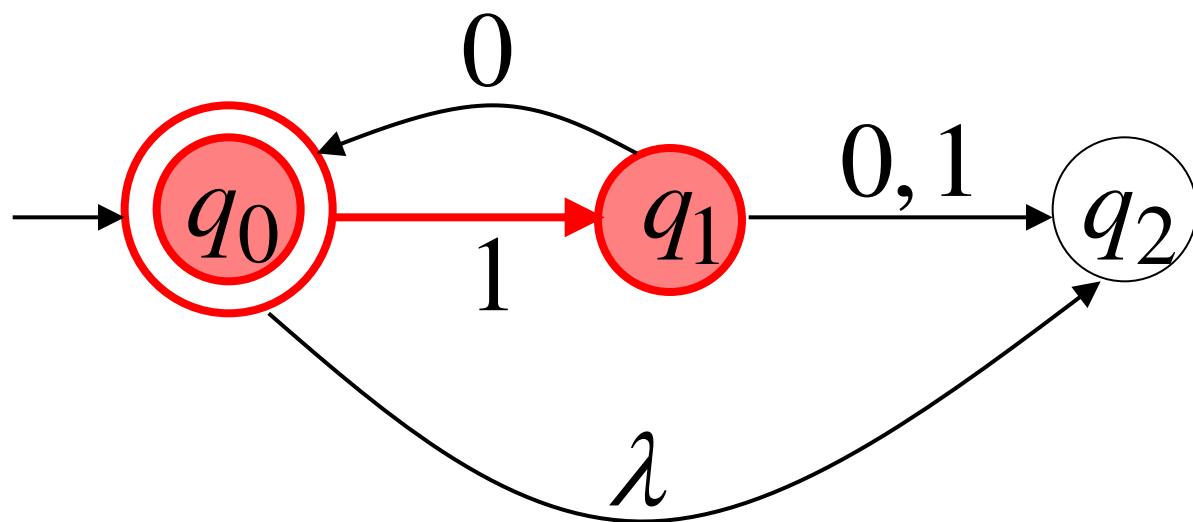
# Transition Function $\delta$

$$\delta(q, x) = \{q_1, q_2, \dots, q_k\}$$

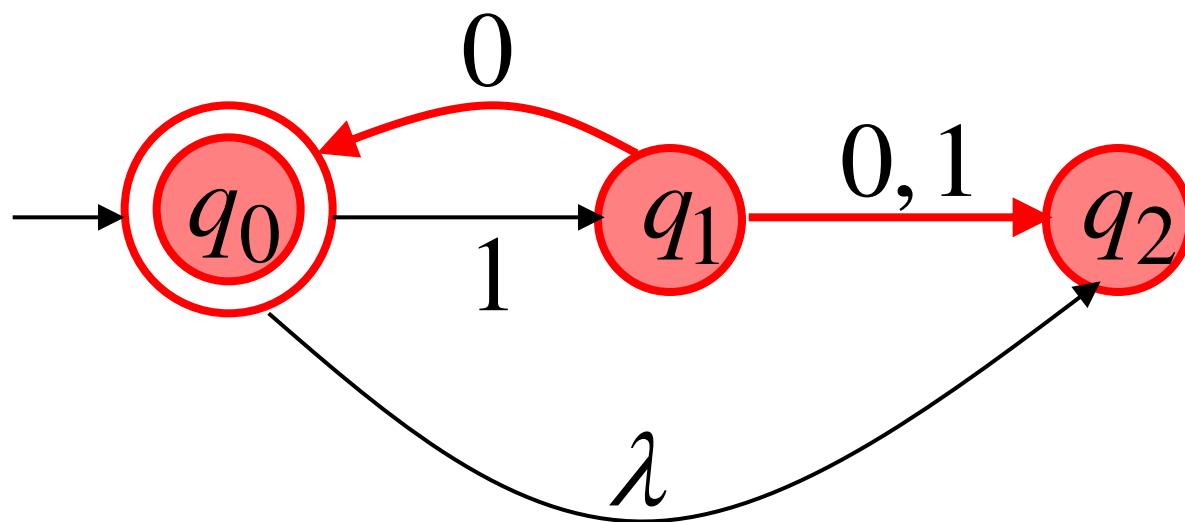


resulting states with  
following **one** transition  
with symbol  $x$

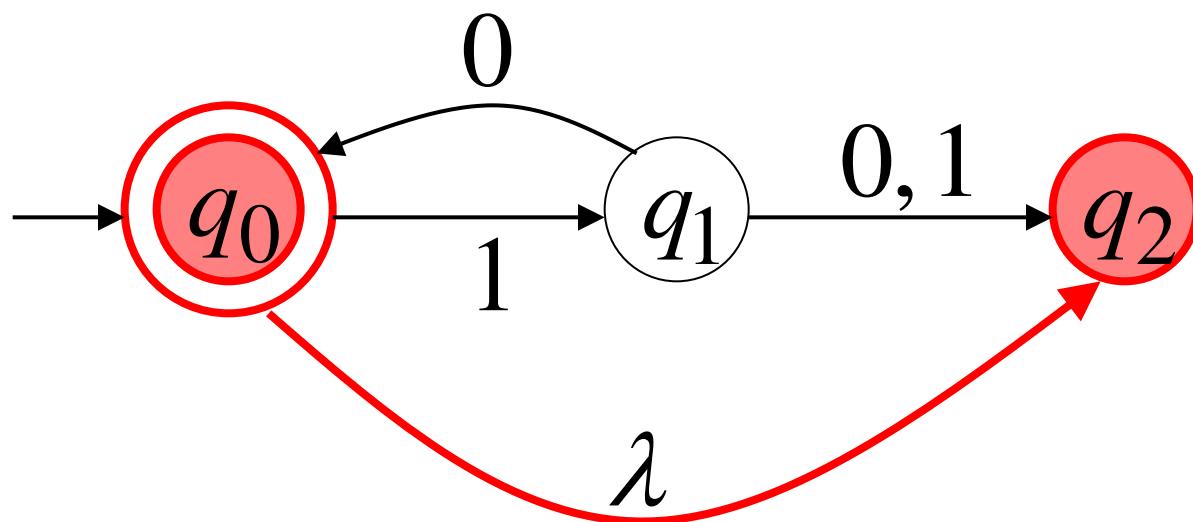
$$\delta(q_0, 1) = \{q_1\}$$



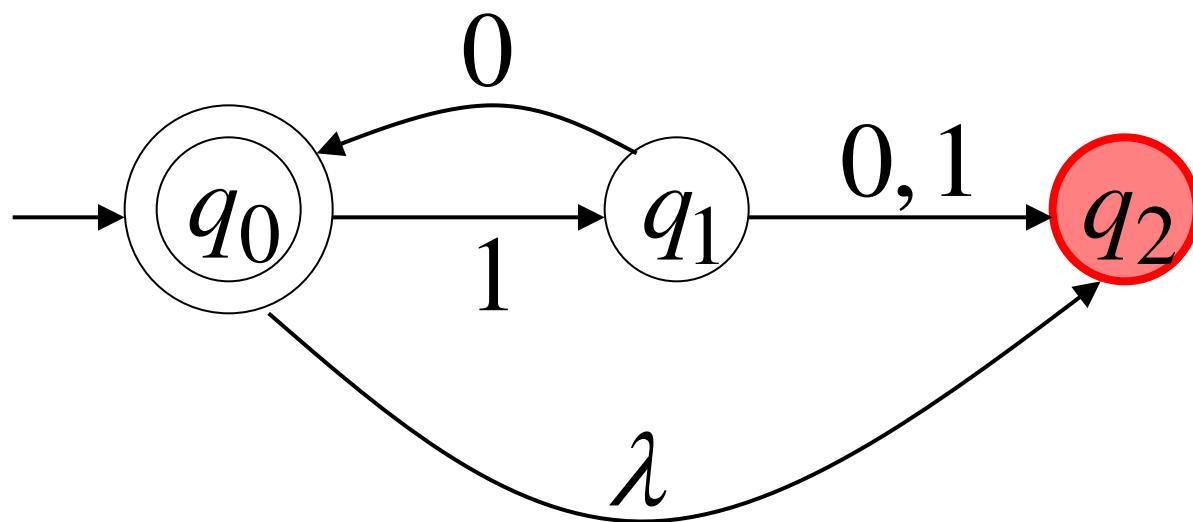
$$\delta(q_1, 0) = \{q_0, q_2\}$$



$$\delta(q_0, \lambda) = \{q_2\}$$



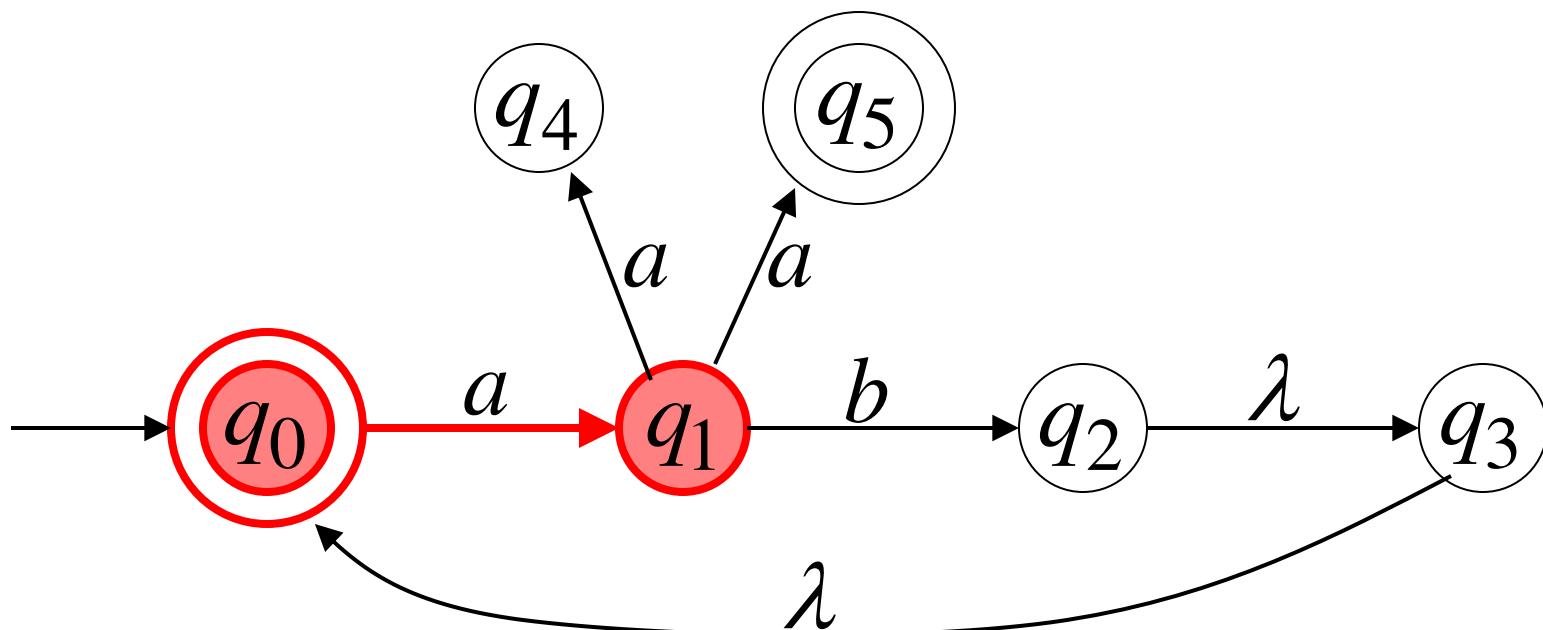
$$\delta(q_2, 1) = \emptyset$$



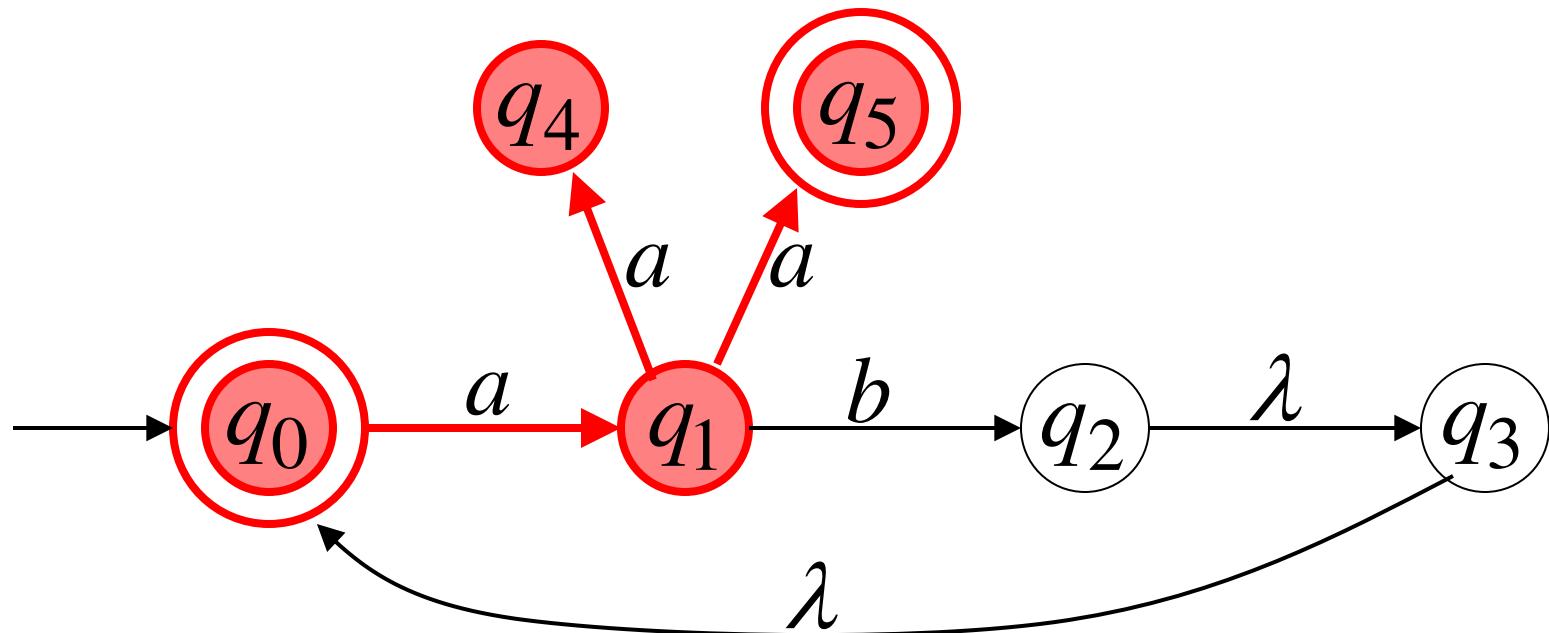
# Extended Transition Function $\delta^*$

Same with  $\delta$  but applied on strings

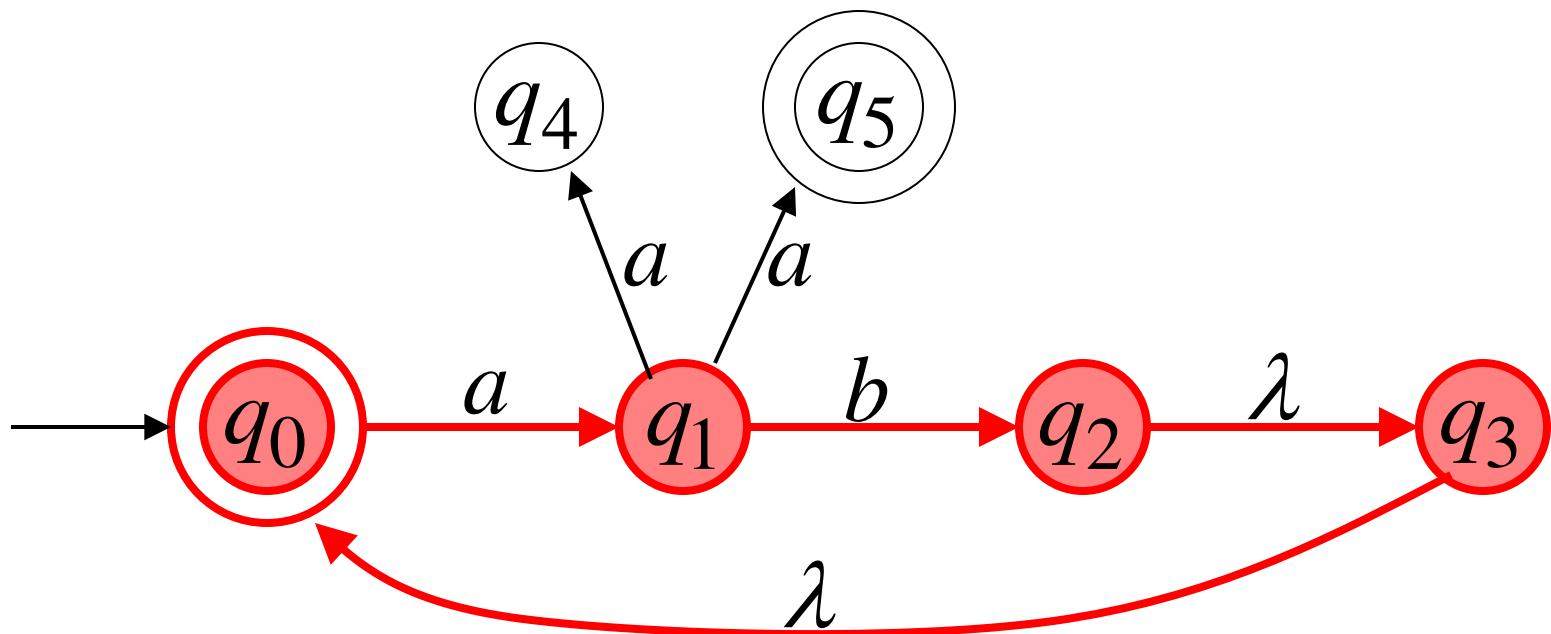
$$\delta^*(q_0, a) = \{q_1\}$$



$$\delta^*(q_0, aa) = \{q_4, q_5\}$$



$$\delta^*(q_0, ab) = \{q_2, q_3, q_0\}$$



Special case:

for any state  $q$

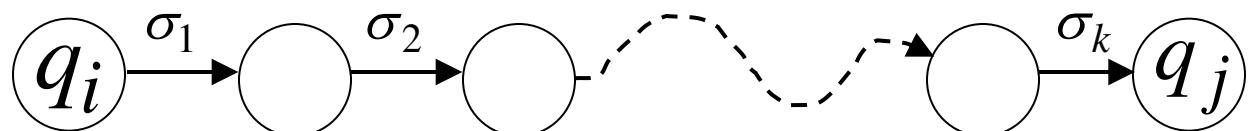
$$q \in \delta^*(q, \lambda)$$

In general

$q_j \in \delta^*(q_i, w)$  : there is a walk from  $q_i$  to  $q_j$   
with label  $w$



$$w = \sigma_1 \sigma_2 \cdots \sigma_k$$



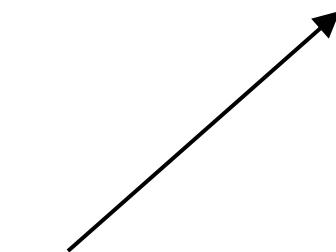
# The Language of an NFA $M$

The language accepted by  $M$  is:

$$L(M) = \{w_1, w_2, \dots, w_n\}$$

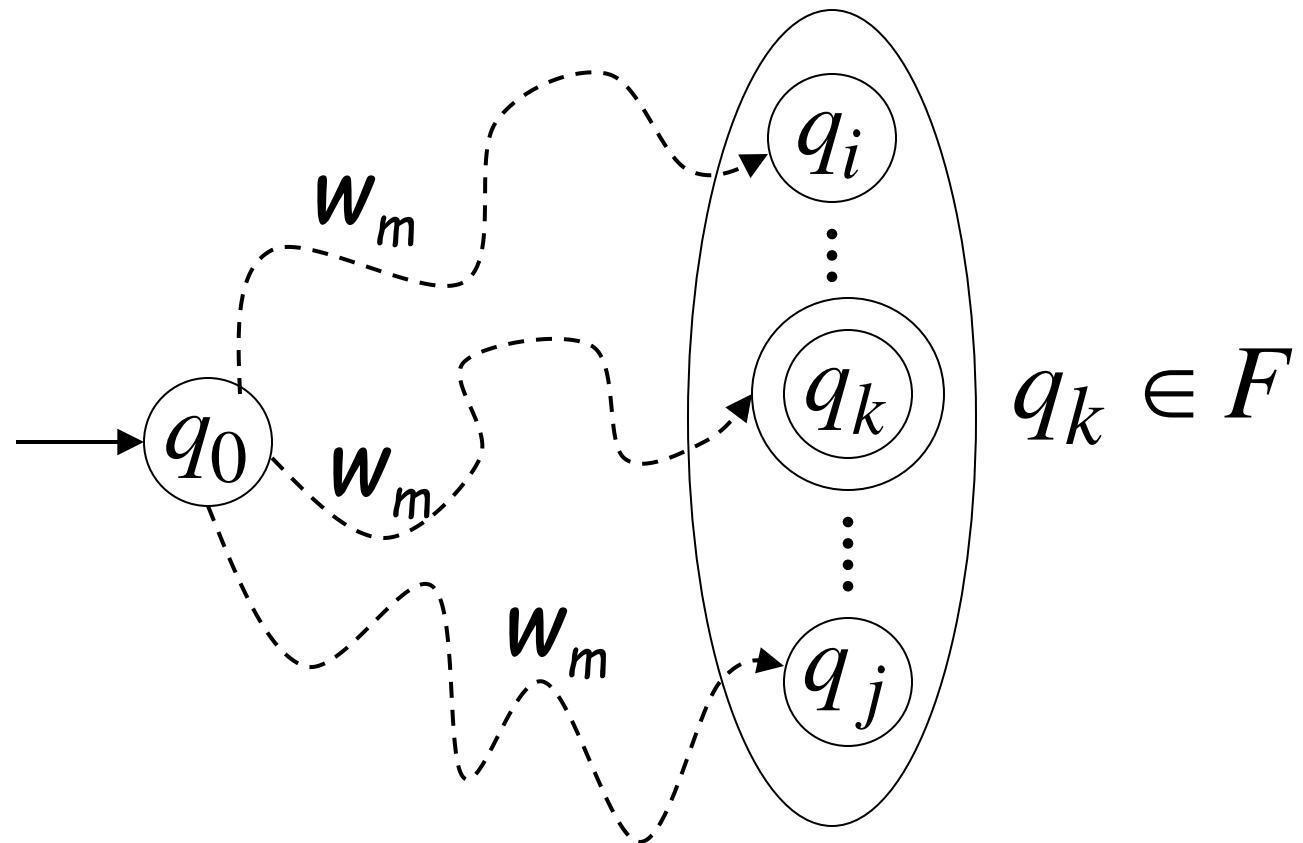
where  $\delta^*(q_0, w_m) = \{q_i, \dots, q_k, \dots, q_j\}$

and there is some  $q_k \in F$  (accepting state)



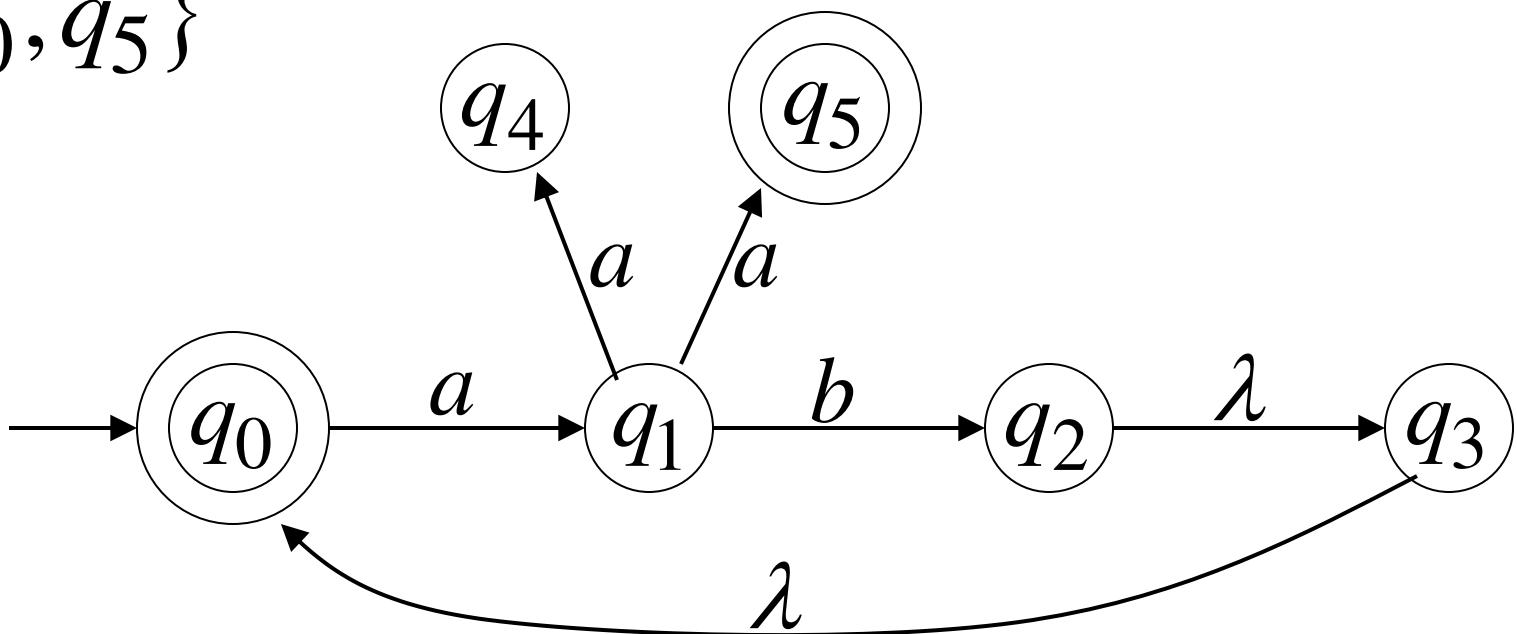
$$w_m \in L(M)$$

$$\delta^*(q_0, w_m)$$



$$q_k \in F$$

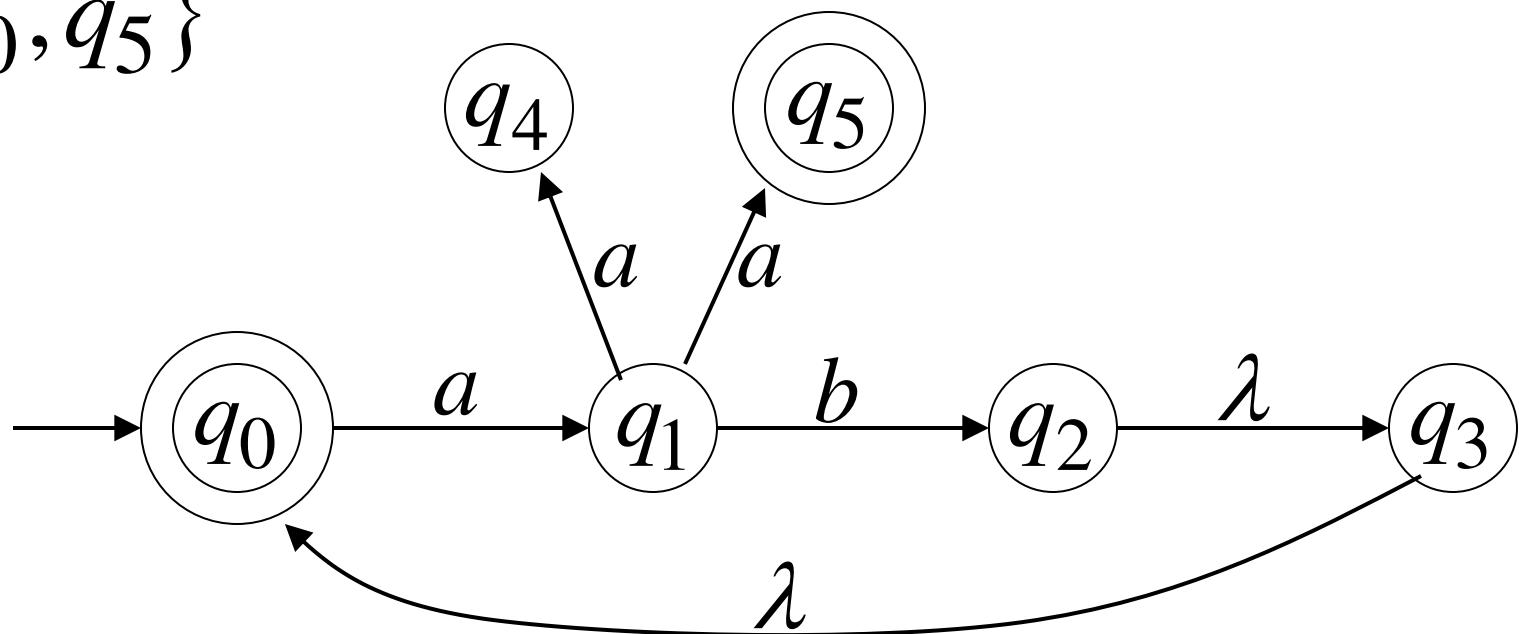
$$F = \{q_0, q_5\}$$



$$\delta^*(q_0, aa) = \{q_4, \underline{q_5}\} \xrightarrow{\quad \text{yellow arrow} \quad} aa \in L(M)$$

$\xrightarrow{\quad \text{yellow arrow} \quad}$

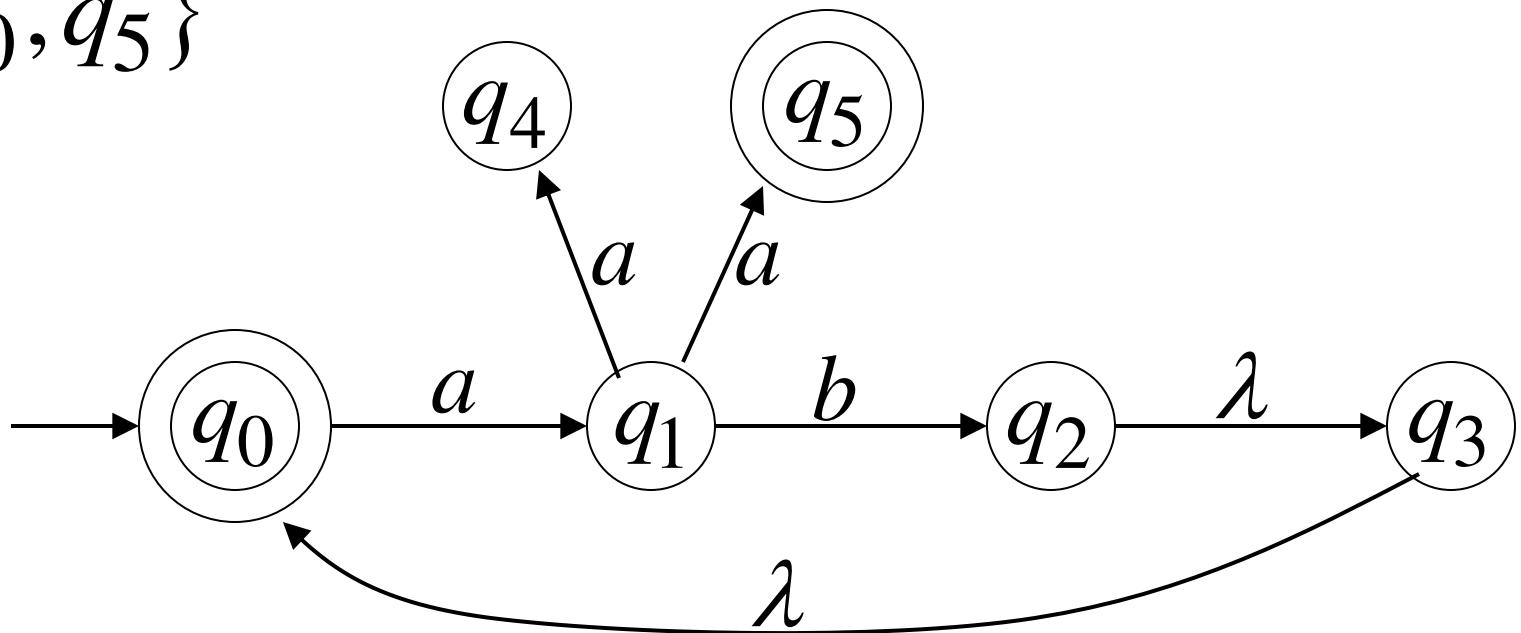
$$F = \{q_0, q_5\}$$



$$\delta^*(q_0, ab) = \{q_2, q_3, \underline{q_0}\} \xrightarrow{\quad} ab \in L(M)$$

$\xrightarrow{\quad} \in F$

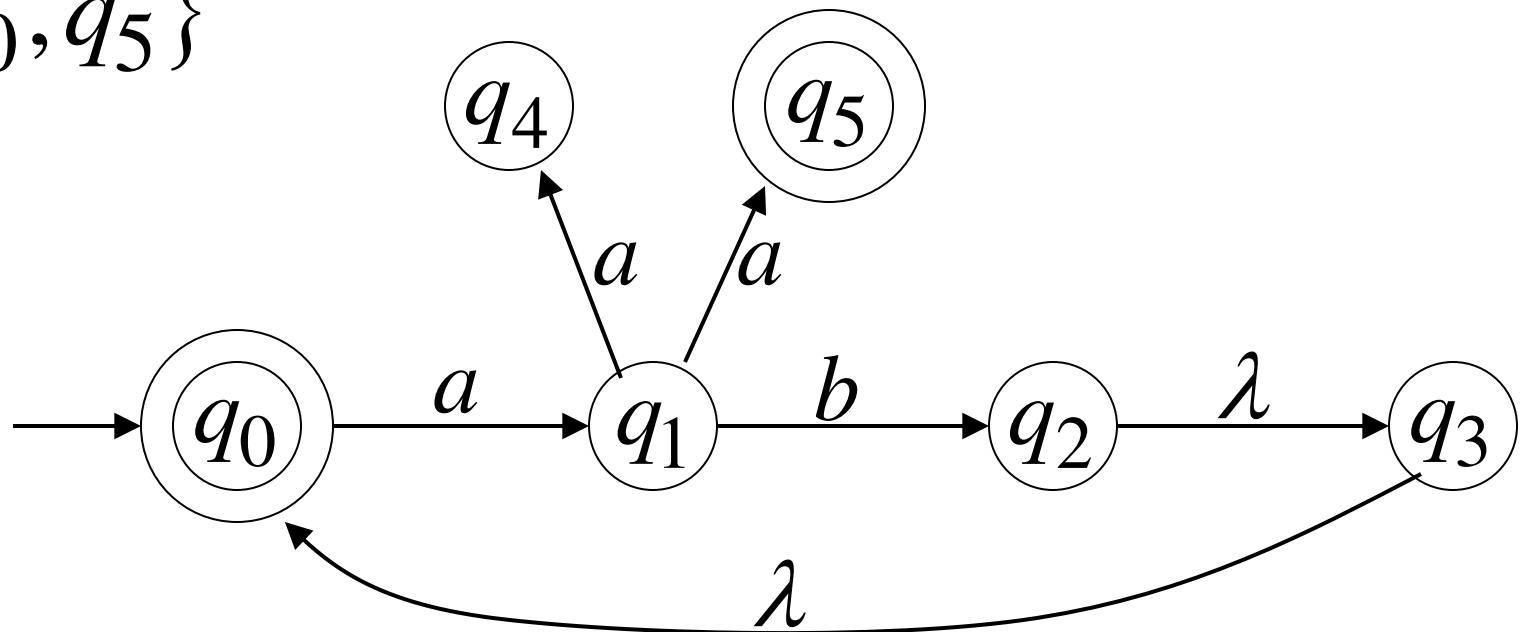
$$F = \{q_0, q_5\}$$



$$\delta^*(q_0, aba) = \{q_4, \underline{q_5}\} \xrightarrow{\quad} aaba \in L(M)$$

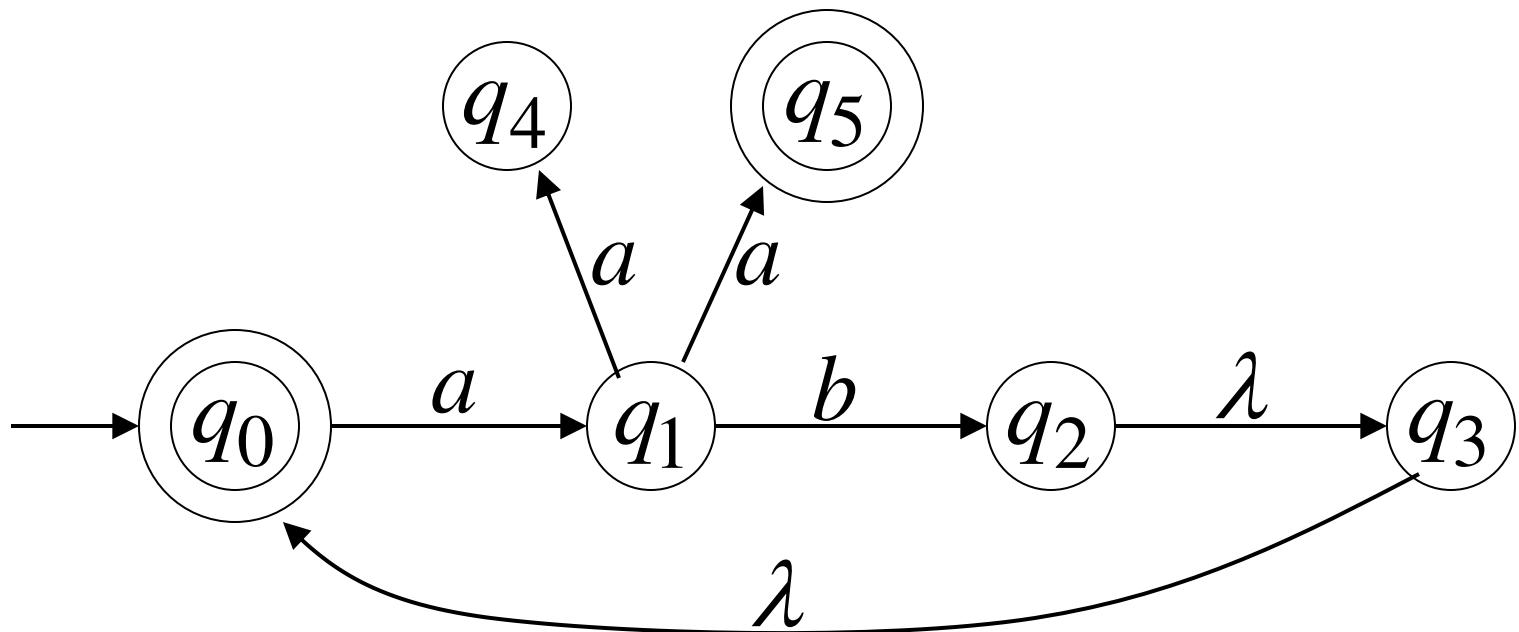
$\xrightarrow{\quad} \in F$

$$F = \{q_0, q_5\}$$



$$\delta^*(q_0, aba) = \{q_1\} \quad \longrightarrow \quad aba \notin L(M)$$

$\not\in F$



$$L(M) = \{ab\}^* \cup \{ab\}^* \{aa\}$$

NFAs accept the Regular Languages

# Equivalence of Machines

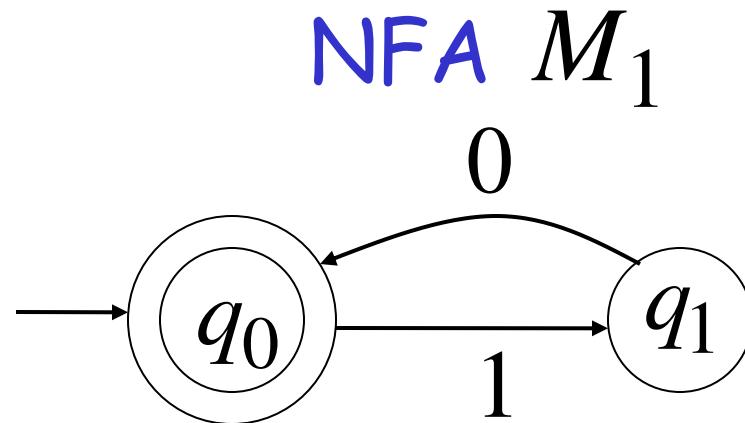
Definition:

Machine  $M_1$  is equivalent to machine  $M_2$

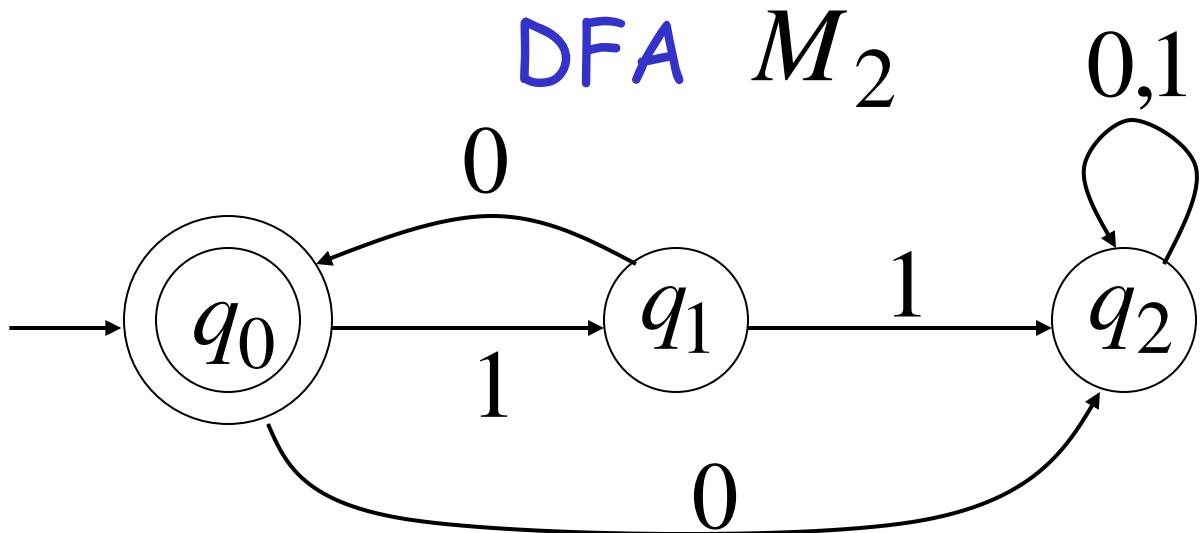
if  $L(M_1) = L(M_2)$

# Example of equivalent machines

$$L(M_1) = \{10\}^*$$



$$L(M_2) = \{10\}^*$$



# Theorem:

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{accepted} \\ \text{by NFAs} \end{array} \right\} = \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

Languages  
accepted  
by DFAs

NFAs and DFAs have the same computation power,  
accept the same set of languages

**Proof:** we only need to show

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{accepted} \\ \text{by NFAs} \end{array} \right\} \supseteq \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

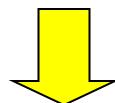
AND

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{accepted} \\ \text{by NFAs} \end{array} \right\} \subseteq \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

## Proof-Step 1

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{accepted} \\ \text{by NFAs} \end{array} \right\} \equiv \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

Every DFA is trivially an NFA

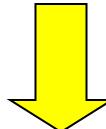


Any language  $L$  accepted by a DFA  
is also accepted by an NFA

## Proof-Step 2

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{accepted} \\ \text{by NFAs} \end{array} \right\} \subseteq \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

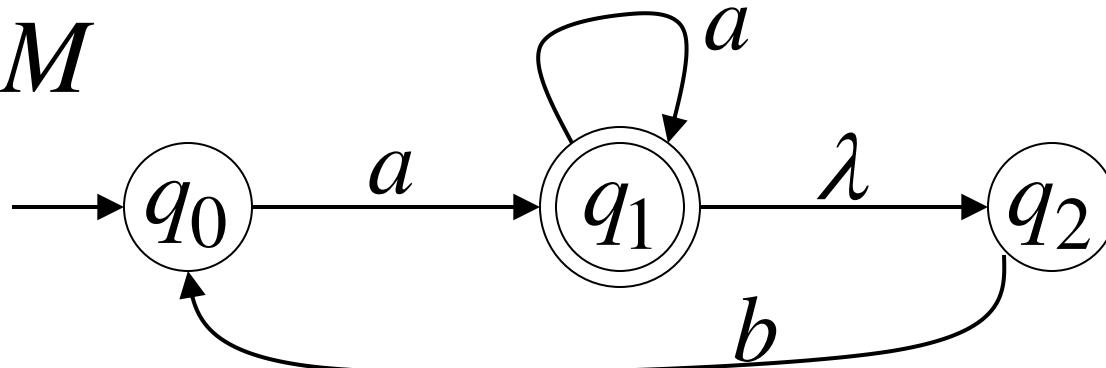
Any NFA can be converted to an equivalent DFA



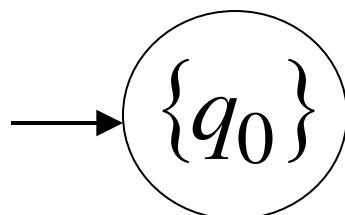
Any language  $L$  accepted by an NFA is also accepted by a DFA

# Conversion NFA to DFA

NFA  $M$

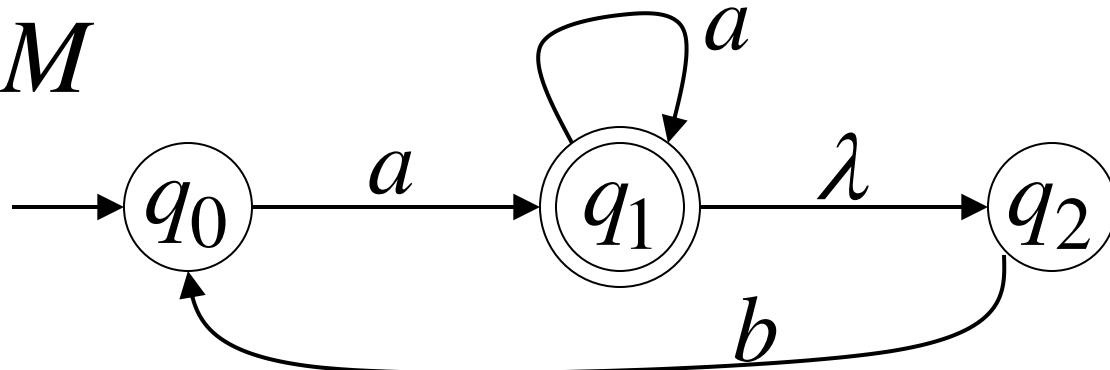


DFA  $M'$

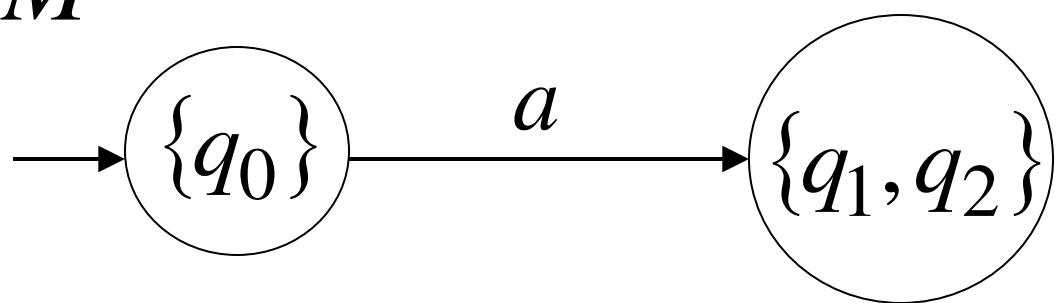


$$\delta^*(q_0, a) = \{q_1, q_2\}$$

NFA  $M$

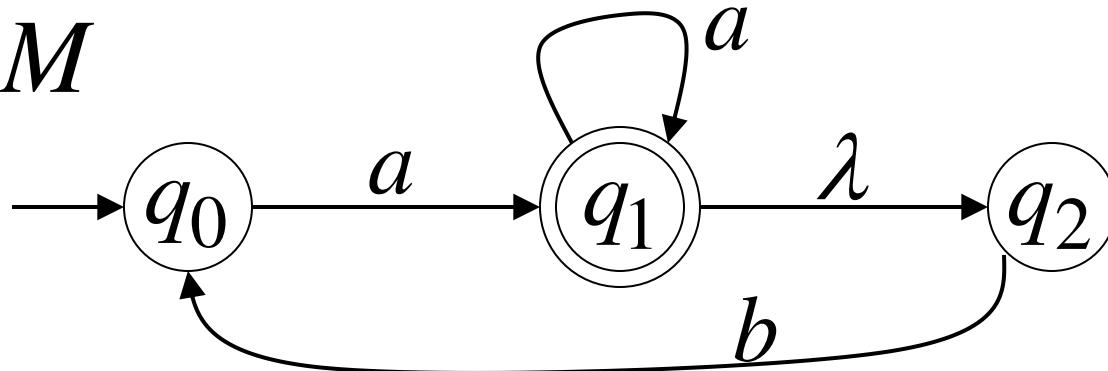


DFA  $M'$

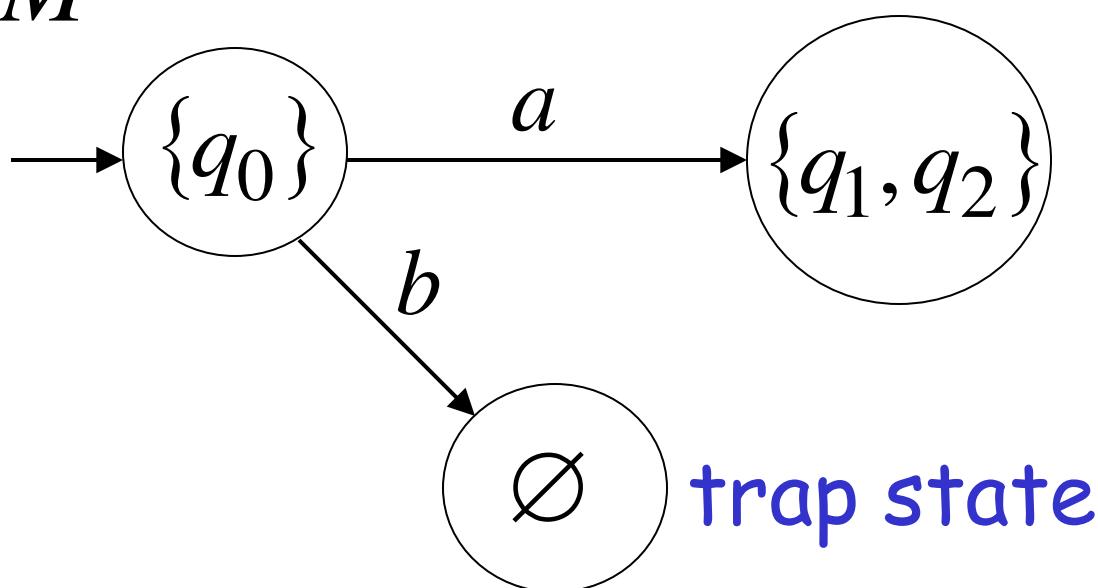


$$\delta^*(q_0, b) = \emptyset \quad \text{empty set}$$

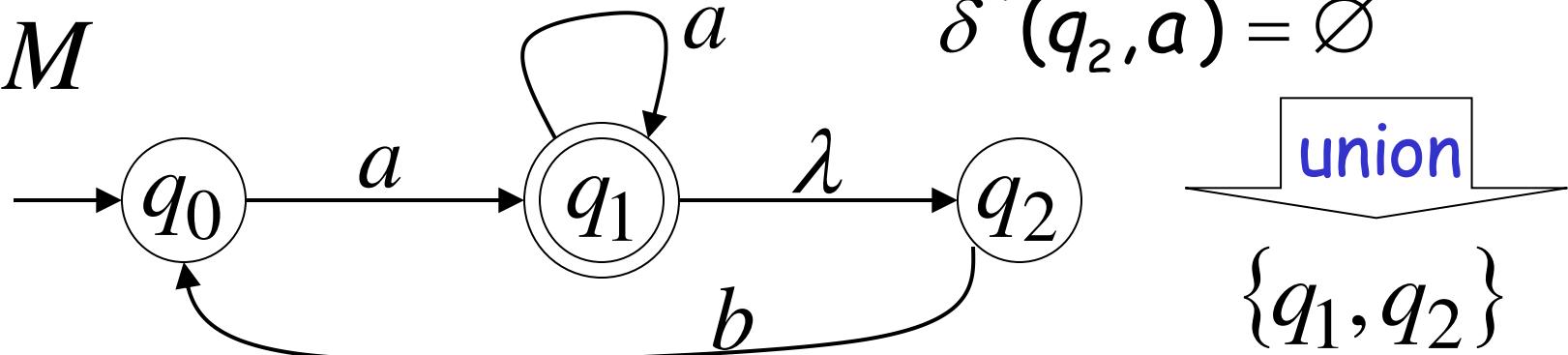
NFA  $M$



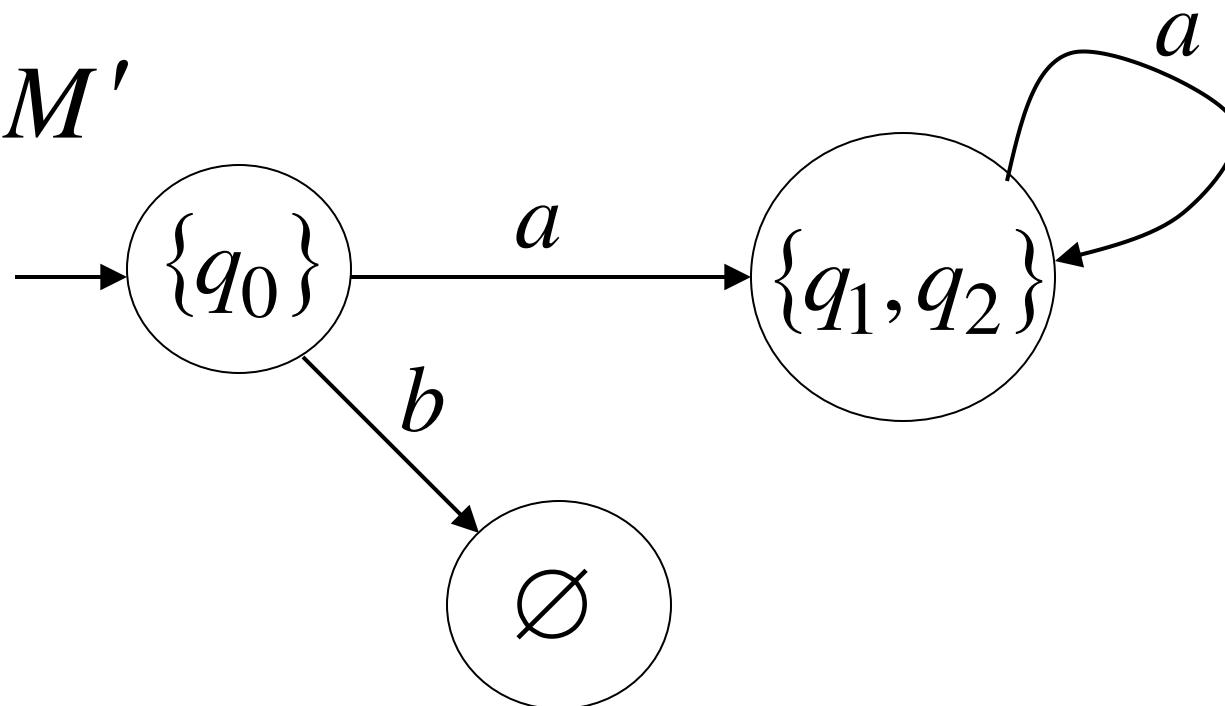
DFA  $M'$



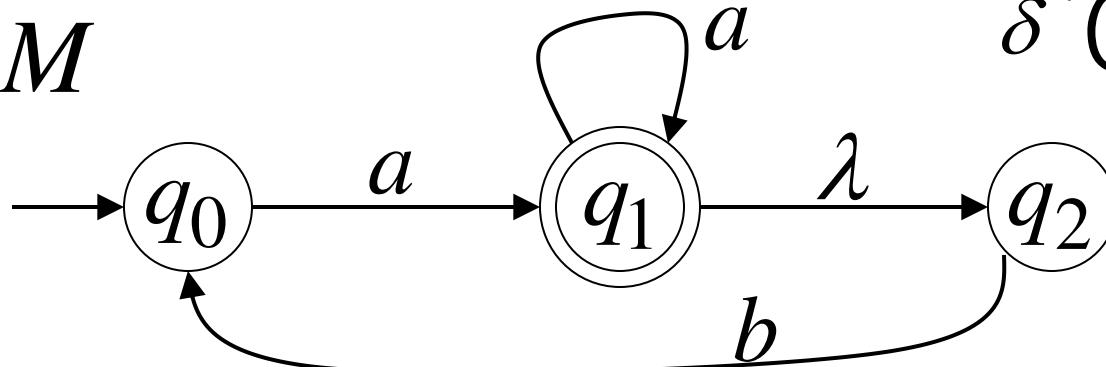
NFA  $M$



DFA  $M'$



NFA  $M$

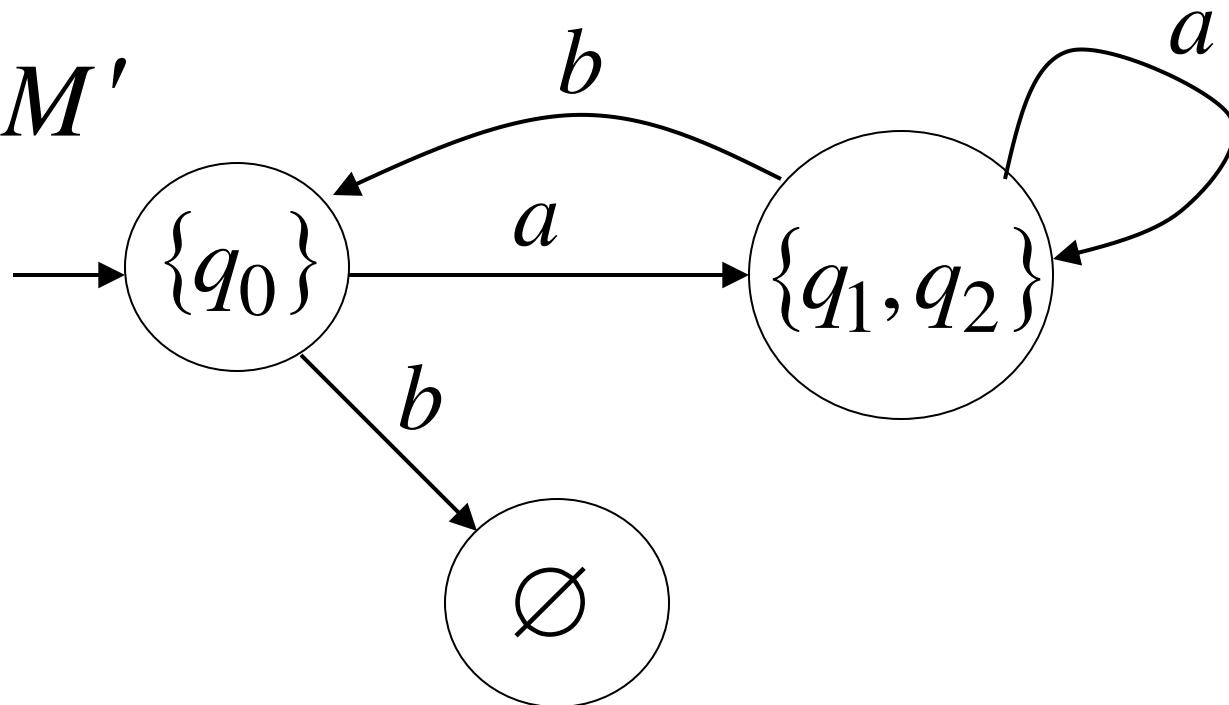


$$\delta^*(q_1, b) = \{q_0\}$$
$$\delta^*(q_2, b) = \{q_0\}$$

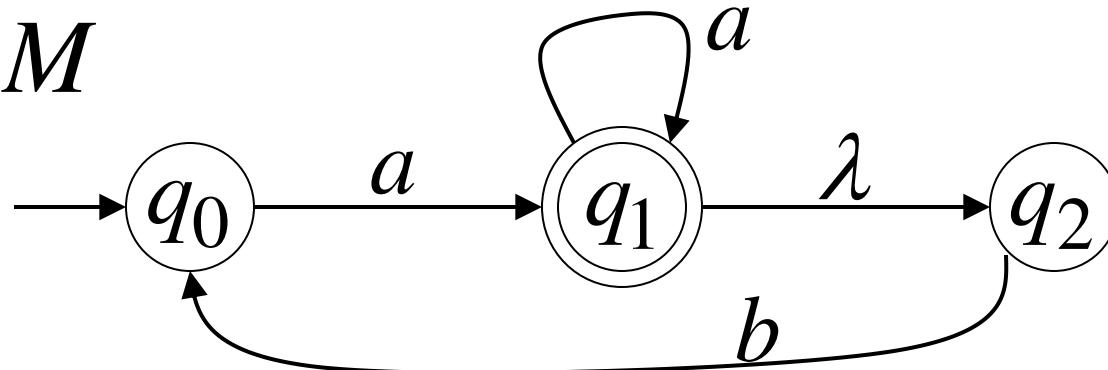
union

$\{q_0\}$

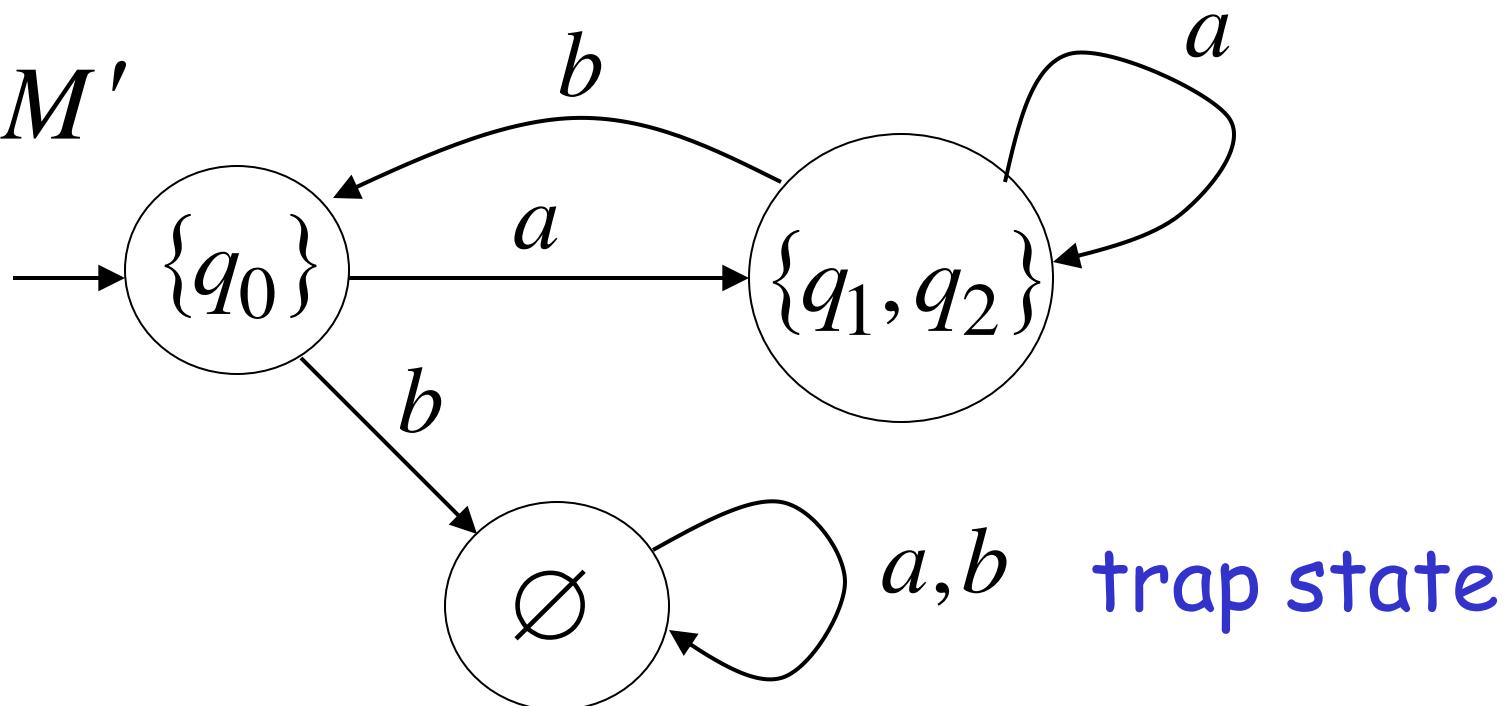
DFA  $M'$



NFA  $M$



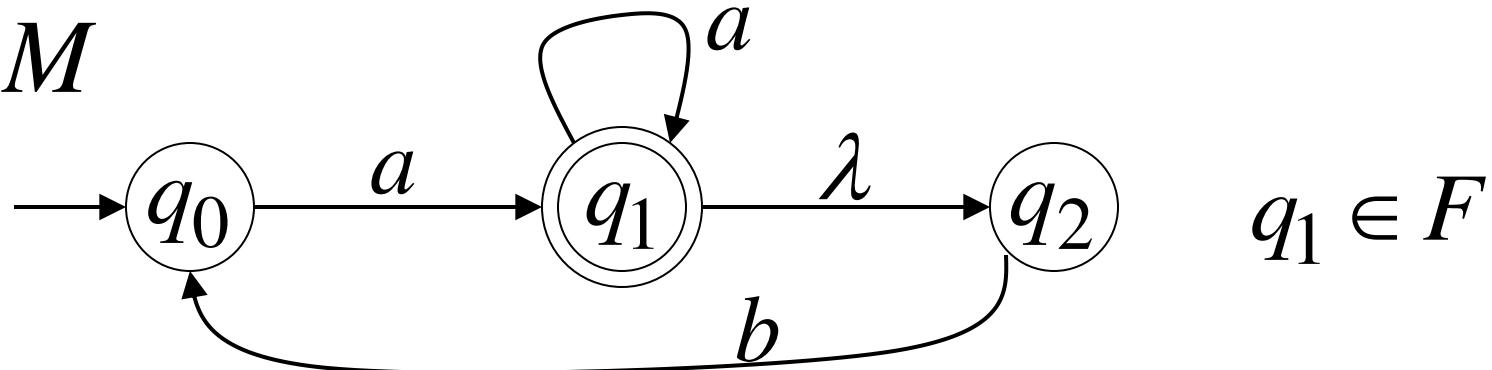
DFA  $M'$



$a, b$  trap state

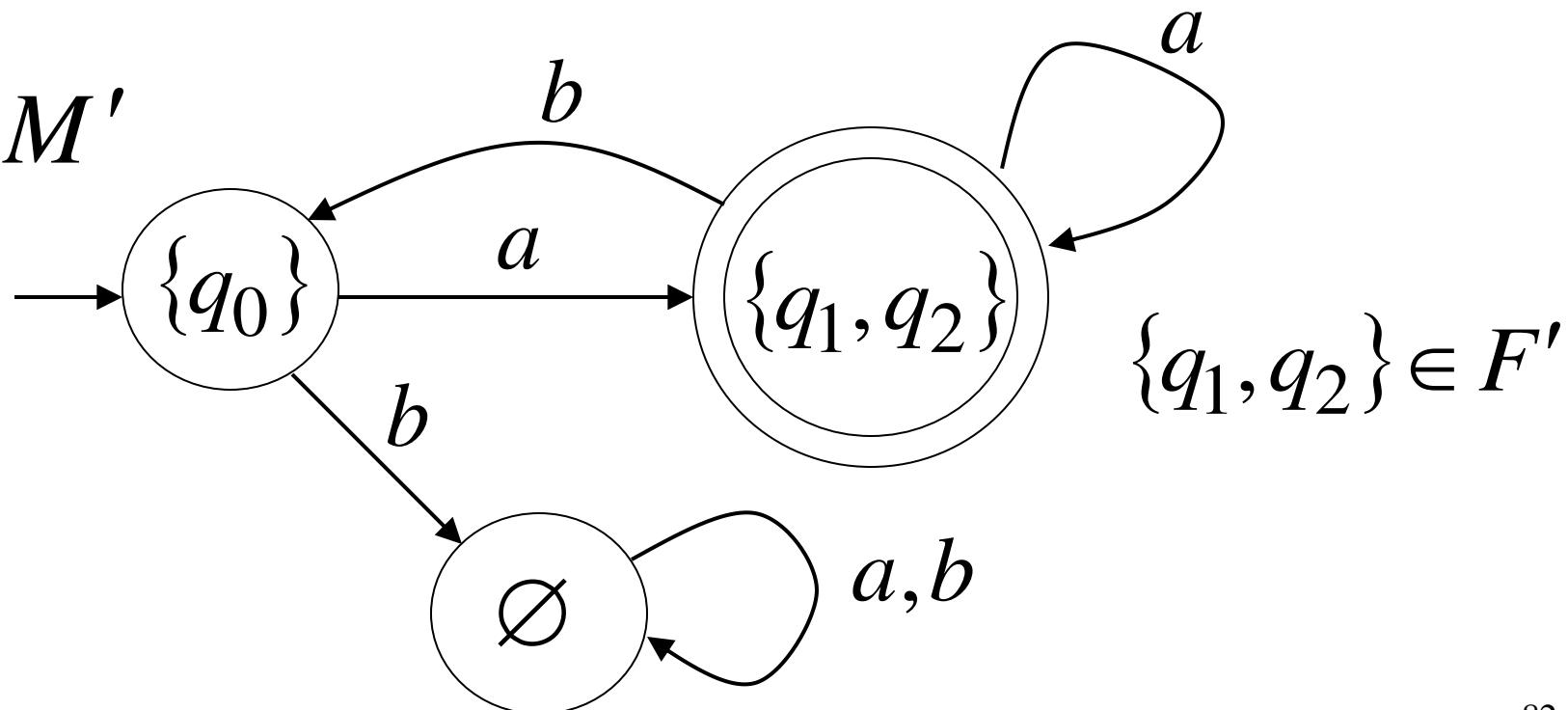
# END OF CONSTRUCTION

NFA  $M$



$$q_1 \in F$$

DFA  $M'$



$$\{q_1, q_2\} \in F'$$

# General Conversion Procedure

Input: an NFA  $M$

Output: an equivalent DFA  $M'$   
with  $L(M) = L(M')$

The NFA has states  $q_0, q_1, q_2, \dots$

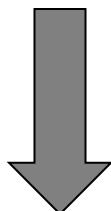
The DFA has states from the power set

$\emptyset, \{q_0\}, \{q_1\}, \{q_0, q_1\}, \{q_1, q_2, q_3\}, \dots$

# Conversion Procedure Steps

step

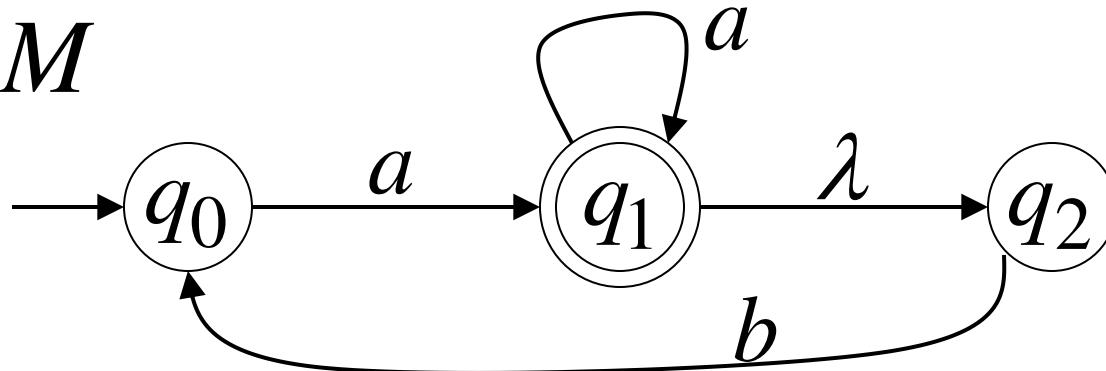
1. Initial state of NFA:  $q_0$



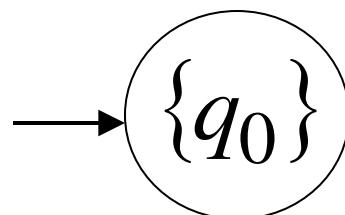
Initial state of DFA:  $\{q_0\}$

# Example

NFA  $M$



DFA  $M'$



step

2. For every DFA's state  $\{q_i, q_j, \dots, q_m\}$

compute in the NFA

$$\left. \begin{array}{l} \delta^*(q_i, a) \\ \cup \delta^*(q_j, a) \\ \dots \\ \cup \delta^*(q_m, a) \end{array} \right\} = \{q'_k, q'_l, \dots, q'_n\}$$

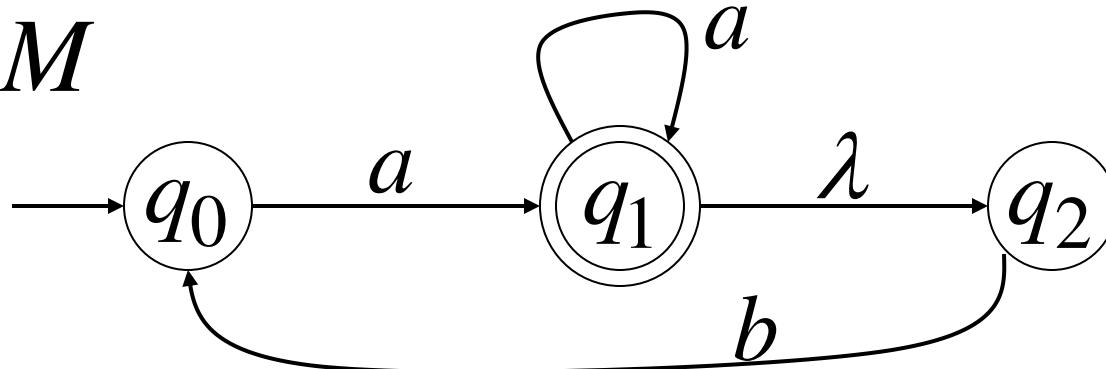
Union

add transition to DFA

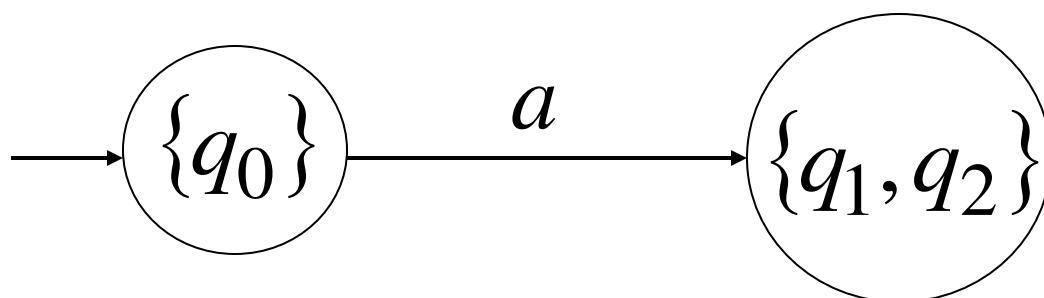
$$\delta(\{q_i, q_j, \dots, q_m\}, a) = \{q'_k, q'_l, \dots, q'_n\}$$

**Example**  $\delta^*(q_0, a) = \{q_1, q_2\}$

**NFA**  $M$



**DFA**  $M'$   $\delta(\{q_0\}, a) = \{q_1, q_2\}$

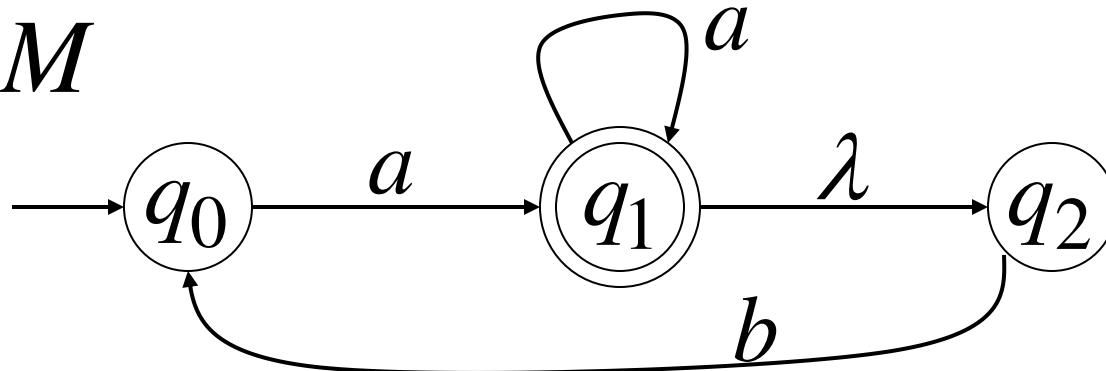


step

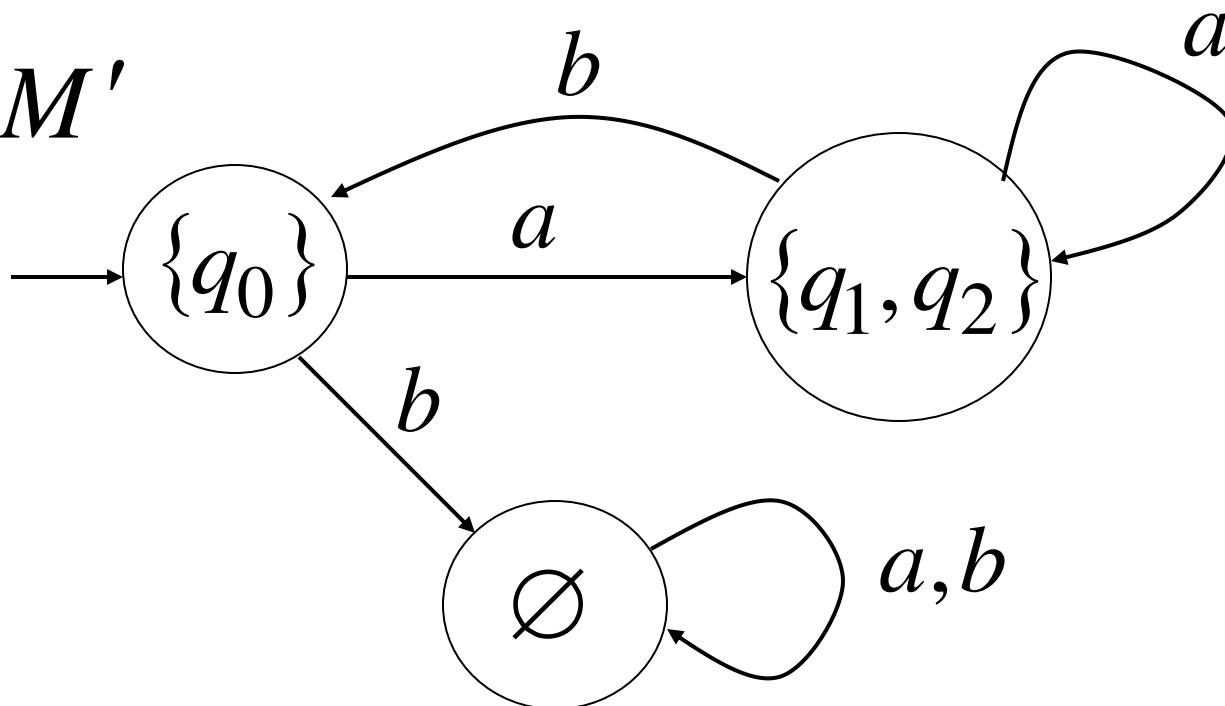
3. Repeat Step 2 for every state in DFA and symbols in alphabet until no more states can be added in the DFA

# Example

NFA  $M$



DFA  $M'$



step

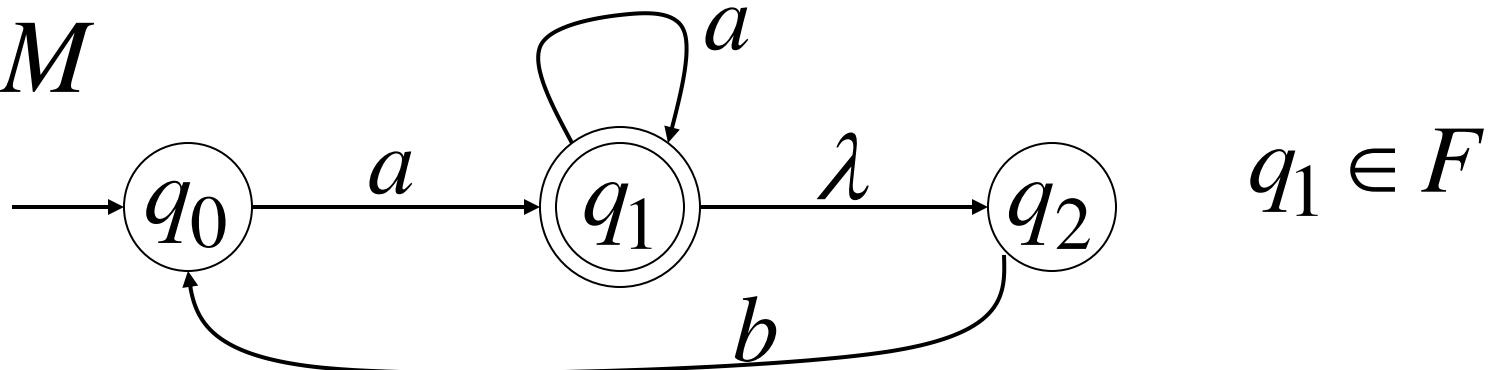
4. For any DFA state  $\{q_i, q_j, \dots, q_m\}$

if some  $q_j$  is accepting state in NFA

Then,  $\{q_i, q_j, \dots, q_m\}$   
is accepting state in DFA

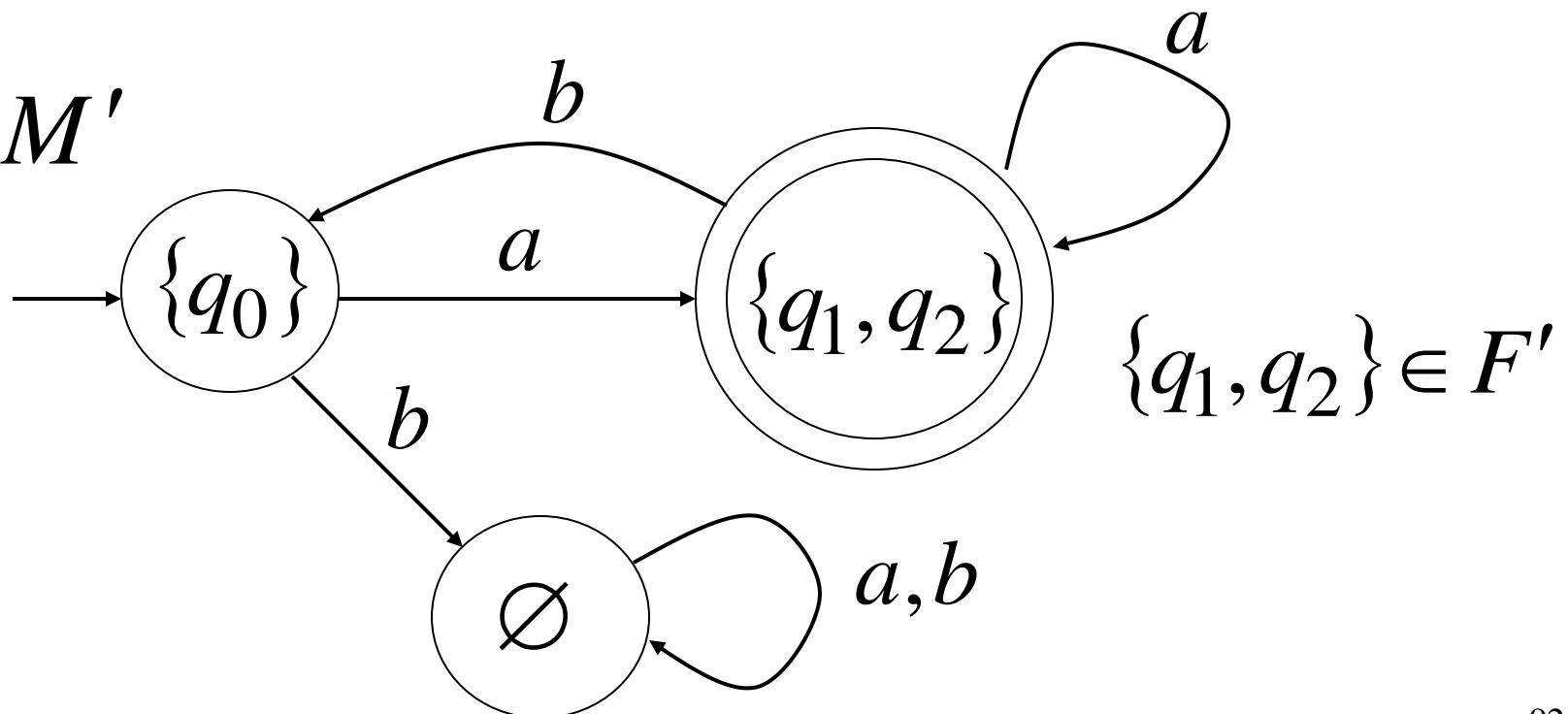
# Example

NFA  $M$



$$q_1 \in F$$

DFA  $M'$



**Lemma:**

If we convert NFA  $M$  to DFA  $M'$   
then the two automata are equivalent:

$$L(M) = L(M')$$

**Proof:**

We only need to show:  $L(M) \subseteq L(M')$

AND

$$L(M) \supseteq L(M')$$

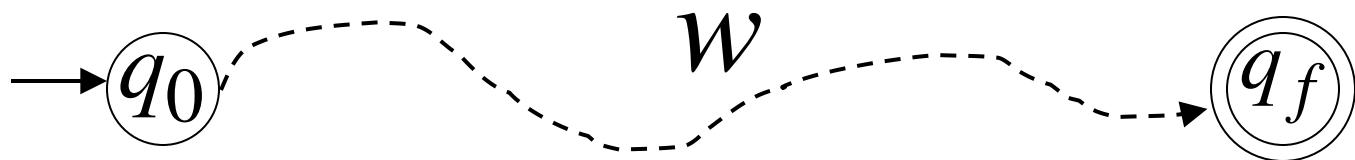
First we show:  $L(M) \subseteq L(M')$

We only need to prove:

$$w \in L(M) \quad \longrightarrow \quad w \in L(M')$$

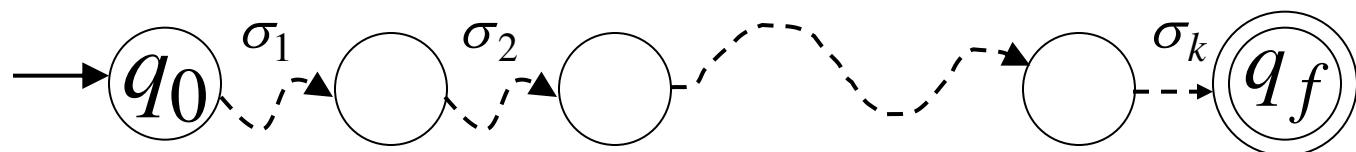
NFA

Consider  $w \in L(M)$



symbols

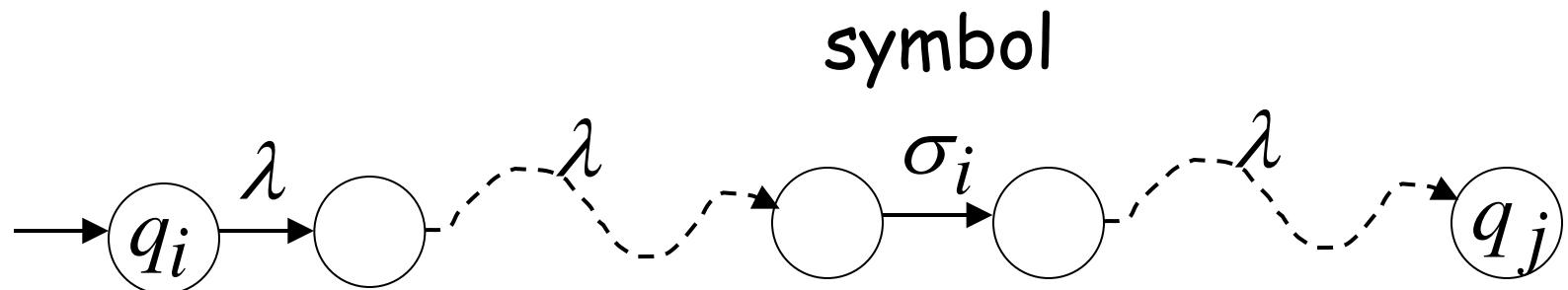
$$w = \sigma_1 \sigma_2 \cdots \sigma_k$$



symbol

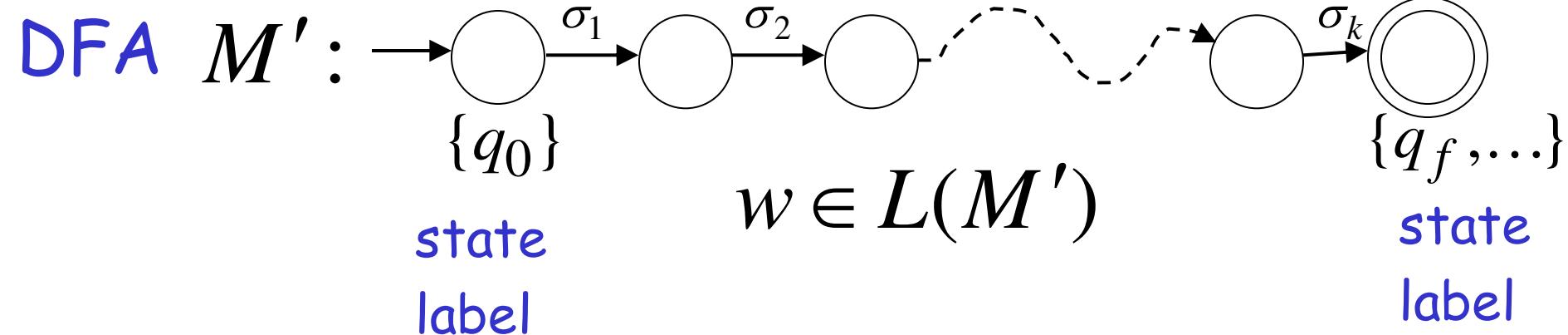
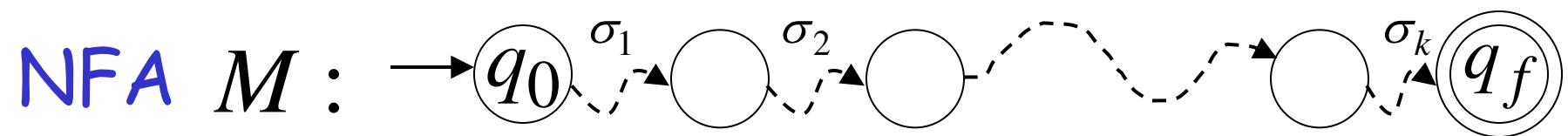


denotes a possible sub-path like



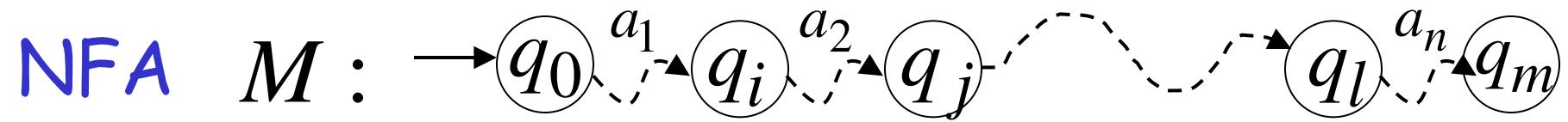
We will show that if  $w \in L(M)$

$$w = \sigma_1 \sigma_2 \cdots \sigma_k$$

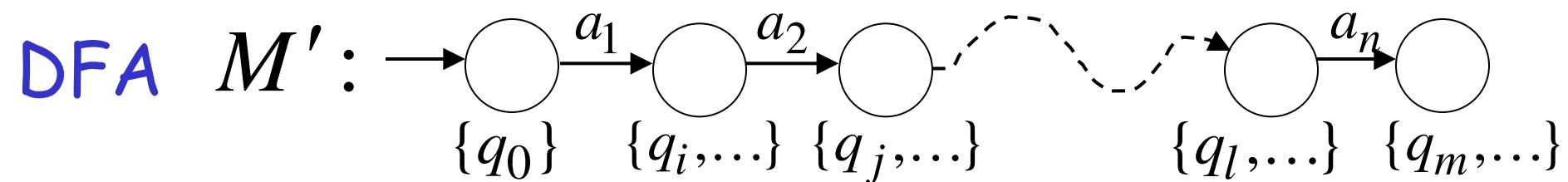


More generally, we will show that if in  $M$ :

(arbitrary string)  $v = a_1 a_2 \cdots a_n$

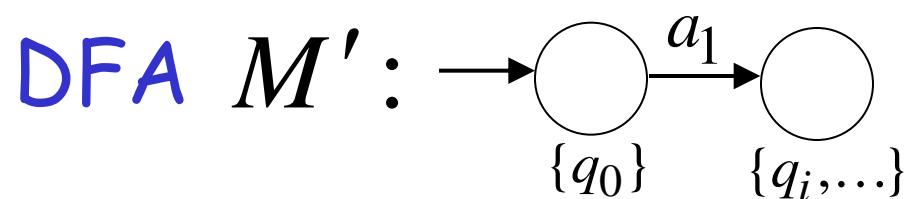
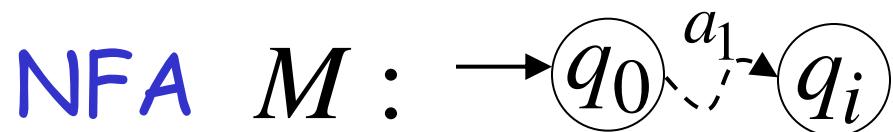


then



# Proof by induction on $|v|$

Induction Basis:  $|v| = 1 \quad v = a_1$

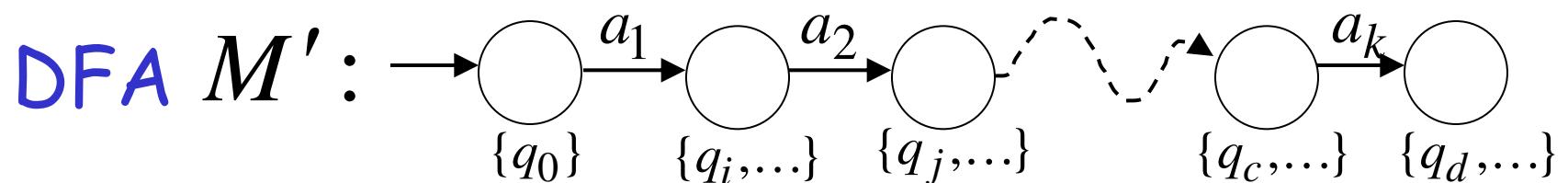
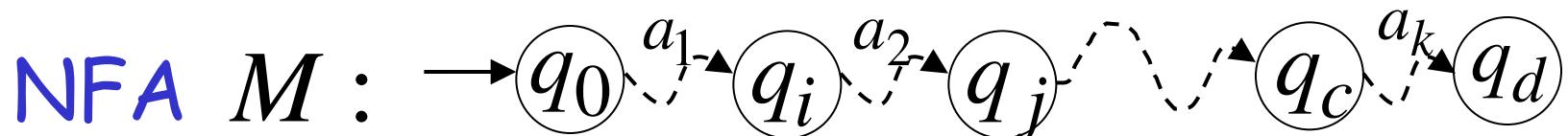


is true by construction of  $M'$

Induction hypothesis:  $1 \leq |v| \leq k$

$$v = a_1 a_2 \cdots a_k$$

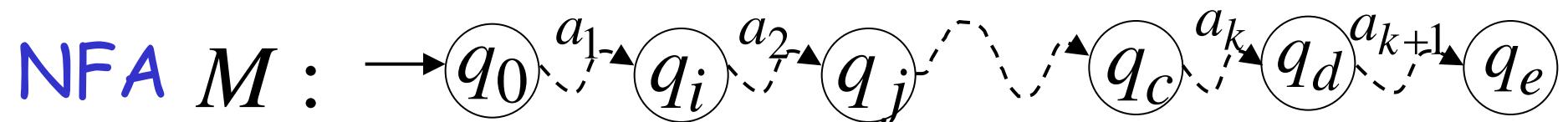
Suppose that the following hold



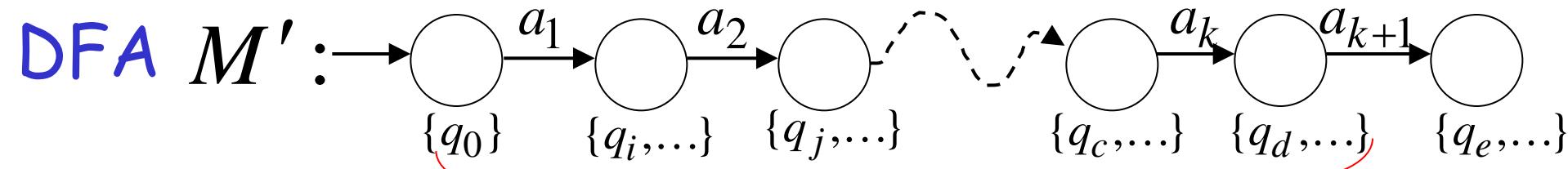
Induction Step:  $|v| = k + 1$

$$v = \underbrace{a_1 a_2 \cdots a_k}_{v'} a_{k+1} = v' a_{k+1}$$

Then this is true by construction of  $M'$



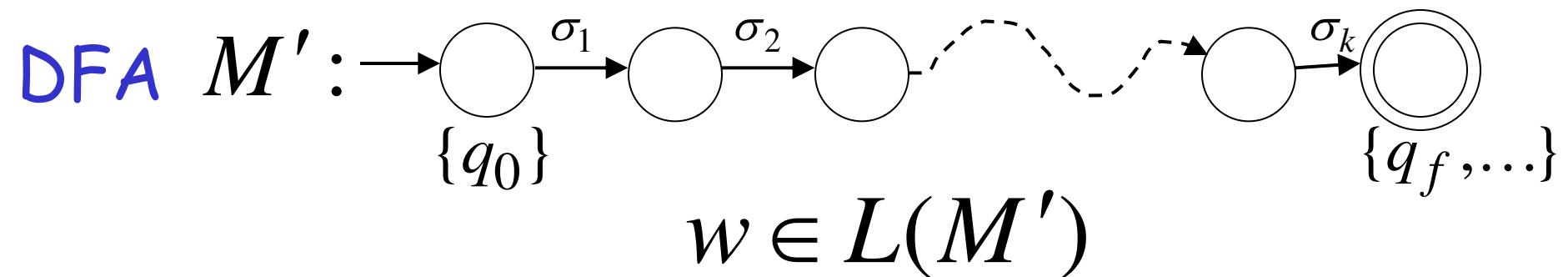
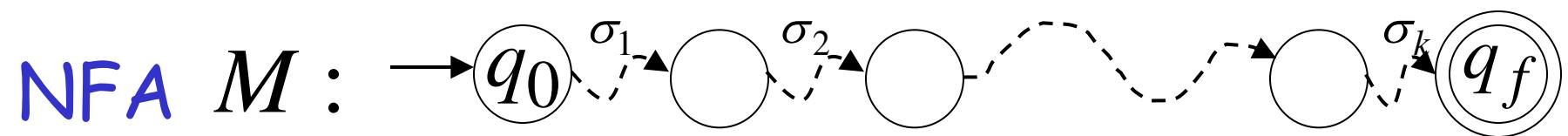
$v'$



$v'$

Therefore if  $w \in L(M)$

$$w = \sigma_1 \sigma_2 \cdots \sigma_k$$



$$w \in L(M')$$

We have shown:  $L(M) \subseteq L(M')$

With a similar proof

we can show:  $L(M) \supseteq L(M')$

Therefore:  $L(M) = L(M')$

END OF LEMMA PROOF