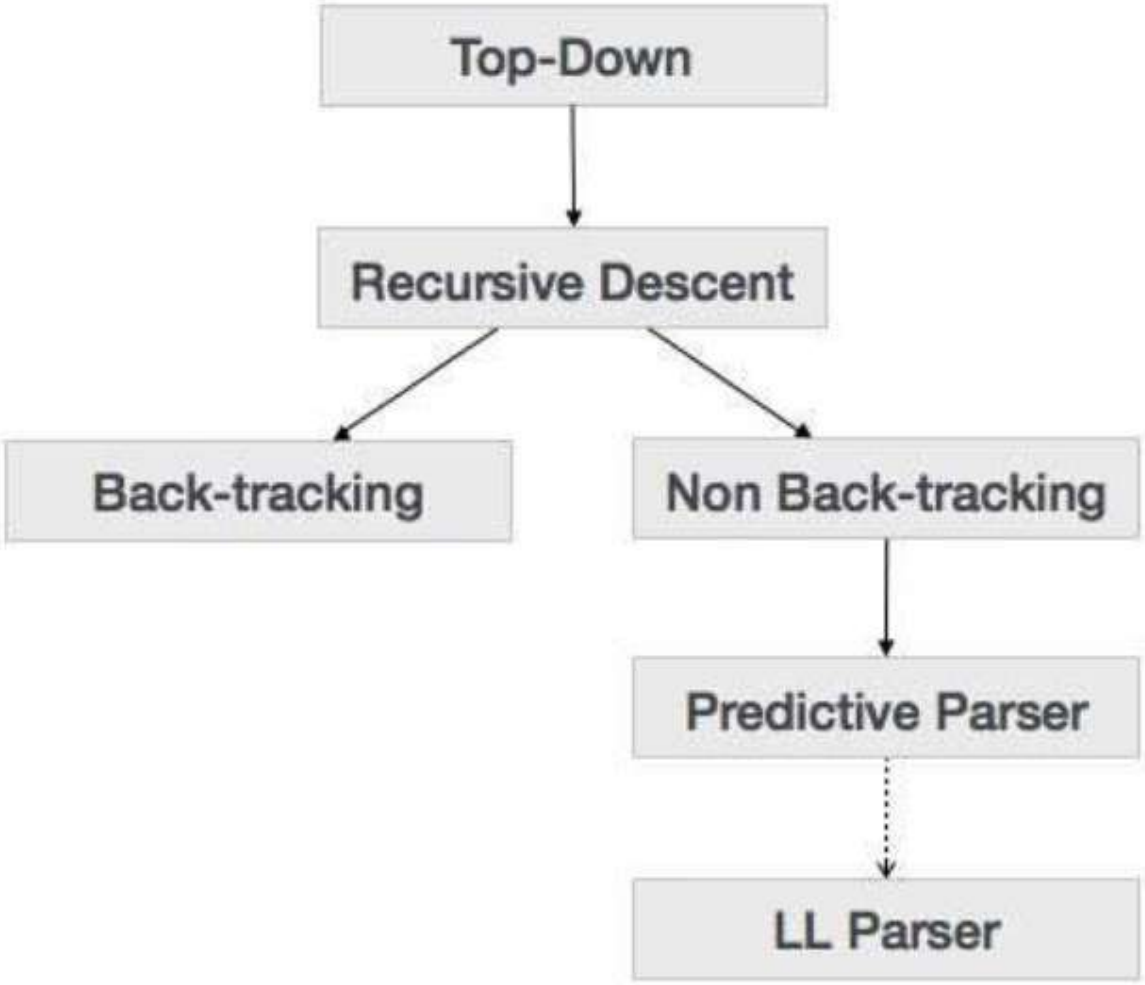
Parsing Methods

- Parsing is the process of determining if a string of token can be generated by a grammar.
- Mainly 2 parsing approaches: Top Down and Bottom Up
- In **top down parsing**, parse tree is constructed from top (root) to the bottom (leaves).
- In **bottom up parsing**, parse tree is constructed from bottom (leaves) to the top (root).

Top Down Parsing

- Parsing is the process of determining if a string of token can be generated by a grammar.
- In **top down parsing**, parse tree is constructed from top (root) to the bottom (leaves).

Top Down Parsing



Procedure-RD Parser

```
void A() {
        Choose an A-production, A → X1 X2 ··· Xk;
        for ( i = 1 to k ) {
                if ( Xi is a nonterminal )
                        call procedure Xi();
                else if ( Xi equals the current input symbol a )
                        advance the input to the next symbol;
                else /* an error has occurred */;
        }
}
```

Recursive Descent Parsing

- It is the most general form of top-down parsing.
- It may involve **backtracking**, that is making repeated scans of input, to obtain the correct expansion of the leftmost non-terminal. Unless the grammar is ambiguous or left-recursive, it finds a suitable parse tree

EXAMPLE

Consider the grammar:

S ⟶ cAd

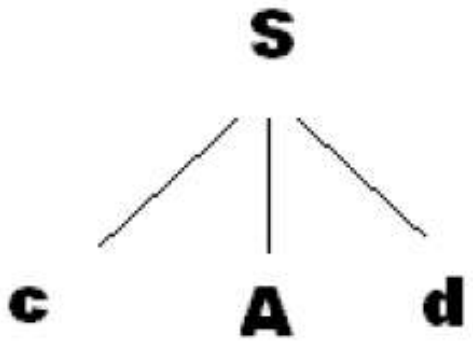
A ⟶ ab | a

and the input string w = cad.

Recursive Descent Parsing

- To construct a parse tree for this string top down, we initially create a tree consisting of a single node labelled **S**.

- An input pointer points to **c**, the first symbol of w. **S** has only one production, so we use it to expand **S** and obtain the tree as:



Recursive Descent Parsing

- The leftmost leaf, labeled **c**, matches the first symbol of input w, so we advance the input pointer to **a**, the second symbol of **w**, and consider the next leaf, labeled **A**.

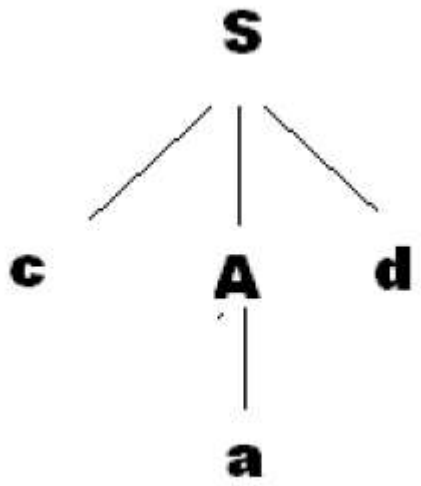
- Now expand **A** using the first alternative **A → ab**
- We have a match for the second input symbol, **a**, so we advance the input pointer to **d**, the third input symbol, and compare d against the next leaf, labeled **b**.

- Since **b** does not match **d**, we report failure and go back to **A** to see whether there is another alternative for **A** that has not been tried, but that might produce a match.

- In going back to **A**, we must reset the input pointer to position 2 , the position it had when we first came to **A**, which means that the procedure for **A** must store the input pointer in a local variable.

- The second alternative for **A** produces the correct Parse Tree

Recursive Descent Parsing



Recursive Descent Parsing (Recursive)

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E' \mid \epsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T' \mid \epsilon$

$F \rightarrow (\ E\ ) \mid id$

```
procedure E( );
begin
    T( );
    EPRIME( )
end;

procedure EPRIME( );
if input-symbol = '+' then
    begin
        ADVANCE( );
        T( );
        EPRIME( )
    end;

procedure T( );
begin
    F( );
    TPRIME( )
end;
```

Recursive Descent Parsing (Recursive) contd…

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

```
procedure T( );
begin
    F( );
    TPRIME( )
end;

procedure TPRIME( );
if input-symbol = '*' then
    begin
        ADVANCE( );
        F( );
        TPRIME( )
    end;
```

Recursive Descent Parsing (Recursive) contd…

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E' \mid \epsilon$

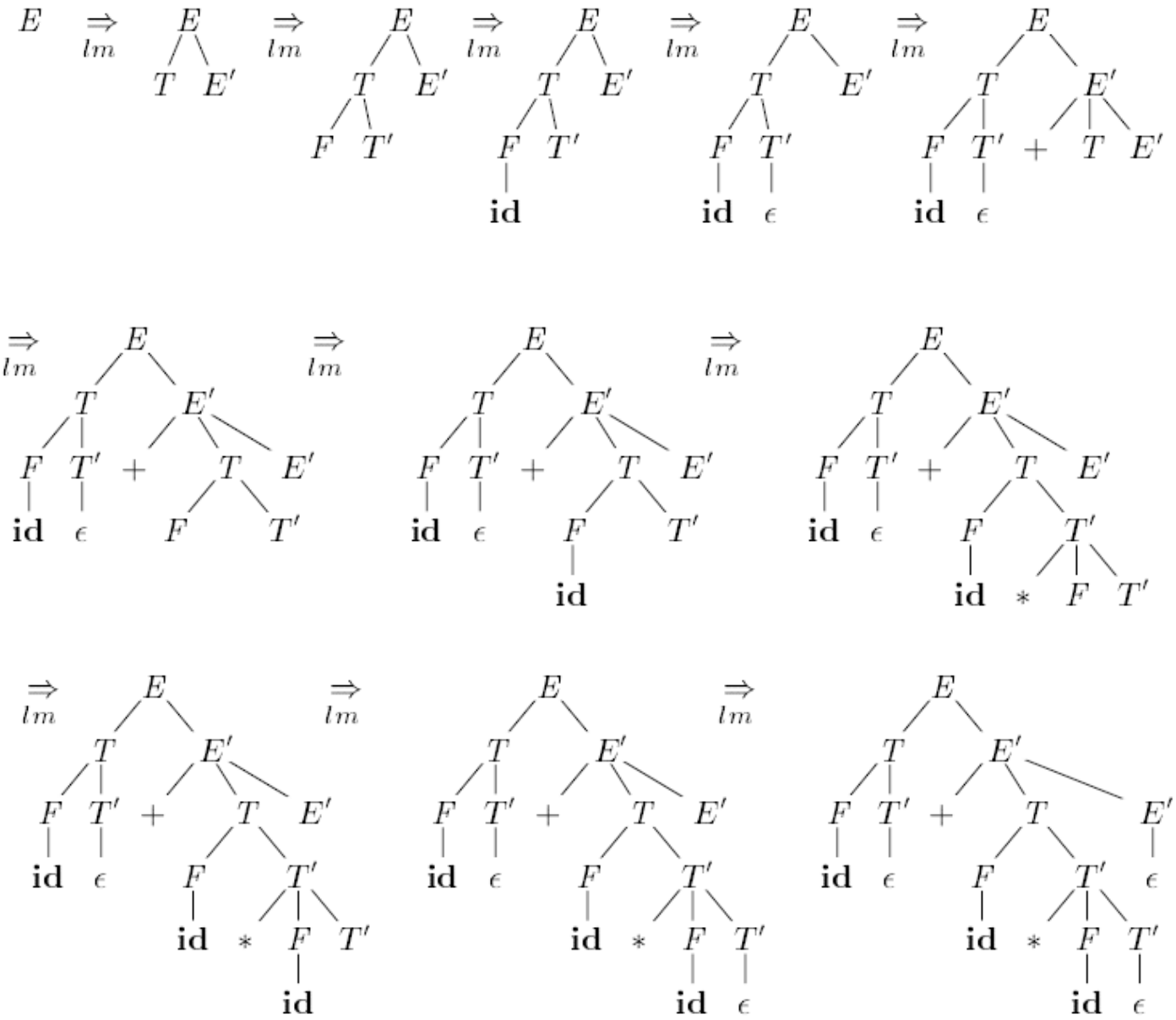
$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T' \mid \epsilon$

$F \rightarrow (\ E\ ) \mid id$

```
procedure F( );
if input-symbol = 'id' then
    ADVANCE( )
else if input-symbol = '(' then
    begin
        ADVANCE( );
        E( );
        if input-symbol = ')' then
            ADVANCE( )
        else ERROR( )
    end
else ERROR( )
```

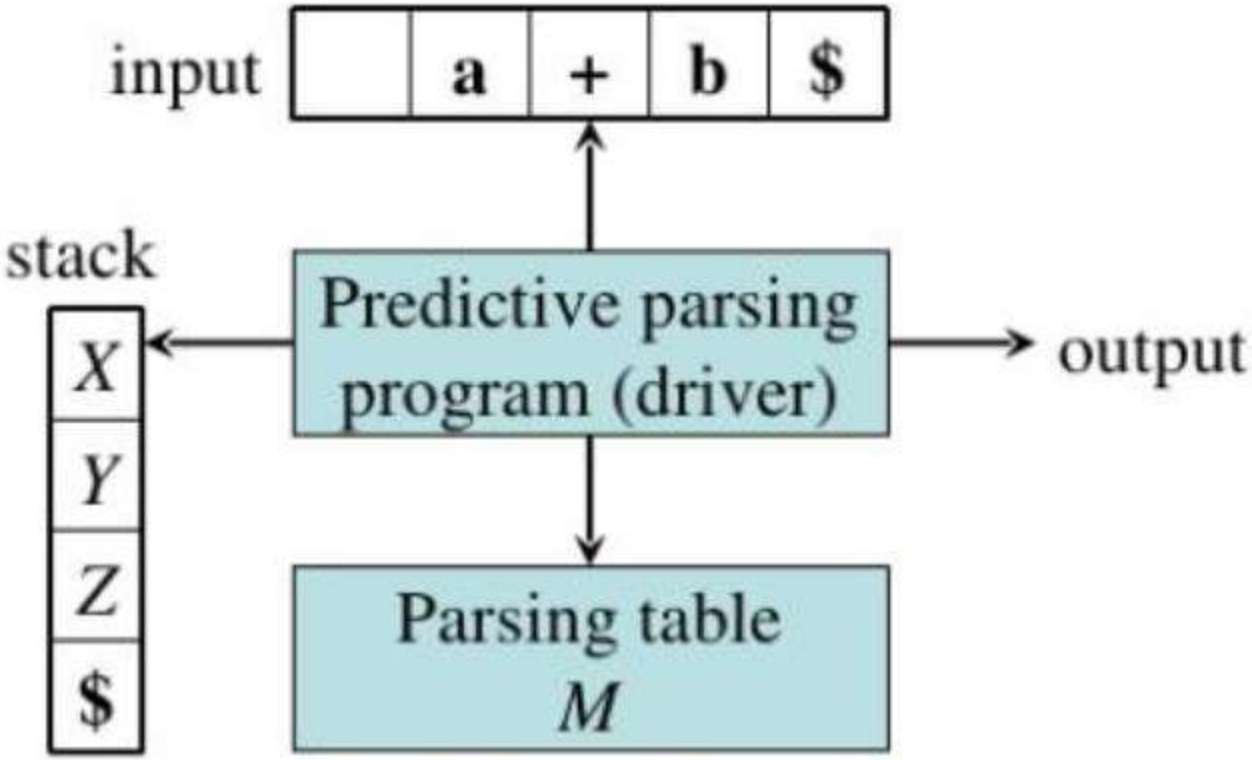
Example id + id *id



Predictive Parser

- It is possible to build a predictive parser(non recursive) by maintaining a stack explicitly, rather than implicitly via recursive calls.
- The key problem during predictive parsing is that of determining the production to be applied for a nonterminal.
- The non recursive parser in looks up the production to be applied in a parsing table

Model


input
a
+
b
$
stack
X
Y
Z
$
Predictive parsing program (driver)
output
Parsing table M

First

- If 'α' is any string of grammar symbols, then FIRST(α) be the set of terminals that begin the string derived from α . If α=*>ϵ then add ϵ to FIRST(α).First is defined for both terminals and non terminals.
- To Compute First Set

1. If **X** is **a** terminal , then FIRST(**X**) is {**X**}

2. If **X→ϵ** then add **ϵ** to FIRST(**X**)

3. If **X** is a non terminal and **X→$Y_1Y_2Y_3$...Yn** , then put '**a**' in FIRST(**X**) if for some **i**, **a** is in FIRST(**Yi**) and ϵ is in all of FIRST(Y_1),...FIRST(Y_{i-1}).

Example

Consider Grammar:

E → T E′

E' → +T E' | Є

T → F T'

T' →* F T' | Є

F → (E) | id

1. FIRST(F) = FIRST(T) = FIRST(E) = {(, **id**}. To see why, note that the two productions for F have bodies that start with these two terminal symbols, **id** and the left parenthesis. T has only one production, and its body starts with F. Since F does not derive ϵ, FIRST(T) must be the same as FIRST(F). The same argument covers FIRST(E).

2. FIRST$(E') = \{+, \epsilon\}$. The reason is that one of the two productions for E' has a body that begins with terminal +, and the other's body is ϵ. Whenever a nonterminal derives ϵ, we place ϵ in FIRST for that nonterminal.

3. FIRST$(T') = \{*, \epsilon\}$. The reasoning is analogous to that for FIRST(E').

Follow

- FOLLOW is defined only for non-terminals of the grammar **G**.
- It can be defined as the set of terminals of grammar **G** , which can immediately follow the non-terminal in a production rule from start symbol.
- In other words, if **A** is a nonterminal, then FOLLOW(**A**) is the set of terminals '**a**' that can appear immediately to the right of **A** in some sentential form.
- Rules to Compute Follow Set

1. If S is the start symbol, then add $ to the FOLLOW(S).

2. If there is a production rule A→αBβ then everything in FIRST(β) except ϵ is placed in FOLLOW(B).

3. If there is a production A→αB , or a production A→αBβ where FIRST(β) contains ϵ then everything in FOLLOW(A) is in FOLLOW(B).

Example

Consider Grammar:

E → T E′

E' → +T E' | Є

T → F T'

T' →* F T' | Є

F → (E) | id

FOLLOW(E) = FOLLOW(E') = {), \$}. Since E is the start symbol, FOLLOW(E) must contain \$. The production body (E) explains why the right parenthesis is in FOLLOW(E). For E', note that this nonterminal appears only at the ends of bodies of E-productions. Thus, FOLLOW(E') must be the same as FOLLOW(E).

FOLLOW(T) = FOLLOW(T') = {+,), \$}. Notice that T appears in bodies only followed by E'. Thus, everything except ϵ that is in FIRST(E') must be in FOLLOW(T); that explains the symbol +. However, since FIRST(E') contains ϵ (i.e., $E' \overset{*}{\Rightarrow} \epsilon$), and E' is the entire string following T in the bodies of the E-productions, everything in FOLLOW(E) must also be in FOLLOW(T). That explains the symbols \$ and the right parenthesis. As for T', since it appears only at the ends of the T-productions, it must be that FOLLOW(T') = FOLLOW(T).

FOLLOW(F) = {+, *,), \$}. The reasoning is analogous to that for T

First and Follow

Non-terminal	FIRST	FOLLOW
E	(, id	), $
E'	+, ϵ	), $
T	(, id	+,), $
T'	*, ϵ	+,), $
F	(, id	+, *,), $

Algorithm to Construct Predictive Parsing Table

1. For each production **A** → α of the grammar, do steps 2 and 3.

2. For each terminal **a** in FIRST(α), add ***A*** → α to **M|A, a|**

3. If ϵ is in FIRST(α), add **A** → α to **M|A, b |** for each terminal *b* in FOLLOW (**A**). If ϵ is in FIRST (α) and **$** is in FOLLOW(**A**), add A → α to ***M |A, $|***

4. Make each undefined entry of *M* be error

Parsing Table

E → T E′

E' → +T E' | Є

T → F T'

T' →* F T' | Є

F → (E) | id

Non-terminal	FIRST	FOLLOW
E	(, id	), $
E'	+, Є	), $
T	(, id	+,), $
T'	*, Є	+,), $
F	(, id	+, *,), $

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**			$F \to (E)$		

Predictive Parsing Method

INPUT: A string **w** and a parsing table ***M*** for grammar ***G***.

OUTPUT: If **w** is in **L (G)** , a leftmost derivation of **w**; otherwise, an error indication.

METHOD: Initially, the parser is in a configuration in which it has **$S** on the stack with **S**, the start symbol of **G** on top, and **w$** in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is shown below.

set ip to point to the first symbol of **w$**;

Predictive Parsing (contd…)

let a be the first symbol of w;
let X be the top stack symbol;
while ($X \neq \$$) { /* stack is not empty */
    if ($X = a$) pop the stack and **let** a be the next symbol of w;
    else if (X is a terminal) *error*();
    else if ($M[X, a]$ is an error entry) *error*();
    else if ($M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_k$) {
        output the production $X \rightarrow Y_1 Y_2 \cdots Y_k$;
        pop the stack;
        push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with Y_1 on top;
    }
    let X be the top stack symbol;
}

Predictive Parsing Actions

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+\ TE'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	output $E' \to +\ TE'$
$\mathbf{id}\ +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id}\ +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id}\ +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$*\ FT'E'\$$	$* \mathbf{id}\$$	output $T' \to *\ FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Exercise

Find First and Follow

1) E → E A E | (E) | - E | id

   A → + | *

2)

   S → SS | AB

   A → Aa | a

   B → Bb | b

Example

$$
\begin{aligned}
S &\rightarrow iEtSS' \mid a \\
S' &\rightarrow eS \mid \epsilon \\
E &\rightarrow b
\end{aligned}
$$

First

S = { i , a }

S' = { e, ϵ }

E = { b }

Follow

S = { e , $ }

S' = { e , $ }

E = { t }

Example (contd.)

NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	$\$$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

Multiple entries are there for M[S', e] . So the grammar is not LL(1)

LL(1) Grammar Conditions

For A → α | β

1. For no terminal a do α and β derive strings beginning with a
2. At most one of α and β derive empty string
3. If β =*> ϵ, then α does not derive any string beginning with a terminal in **FOLLOW(A)** or vice versa

Panic mode Error Recovery

- In case of an error like: a=b + c // no semi-colon
- The compiler will discard all subsequent tokens till a semi-colon (**synchronizing token**) is encountered.
- This method often skips a considerable amount of input without checking it for additional errors, it has an advantage of **simplicity** and **not to go in infinite loop.**
- In situations where multiple errors in the same statements are rare, this method may be quite adequate.

Panic mode Error Recovery(contd.)

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E \to +TE'$			$E \to \epsilon$	$E \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**	synch	synch	$F \to (E)$	synch	synch

Panic mode Error Recovery(contd.)

STACK	INPUT	REMARK
$E\,\$$	$)\ \mathbf{id} * + \mathbf{id}\,\$$	error, skip)
$E\,\$$	$\mathbf{id} * + \mathbf{id}\,\$$	**id** is in FIRST(E)
$TE'\,\$$	$\mathbf{id} * + \mathbf{id}\,\$$	
$FT'E'\,\$$	$\mathbf{id} * + \mathbf{id}\,\$$	
$\mathbf{id}\ T'E'\$$	$\mathbf{id} * + \mathbf{id}\,\$$	
$T'E'\,\$$	$* + \mathbf{id}\,\$$	
$* FT'E'\,\$$	$* + \mathbf{id}\,\$$	
$FT'E'\,\$$	$+ \mathbf{id}\,\$$	error, $M[F, +] =$ synch
$T'E'\,\$$	$+ \mathbf{id}\,\$$	F has been popped
$E'\,\$$	$+ \mathbf{id}\,\$$	
$+ TE'\,\$$	$+ \mathbf{id}\,\$$	
$TE'\,\$$	$\mathbf{id}\,\$$	
$FT'E'\,\$$	$\mathbf{id}\,\$$	
$\mathbf{id}\ T'E'\,\$$	$\mathbf{id}\,\$$	
$T'E'\,\$$	$\$$	
$E'\,\$$	$\$$	
$\$$	$\$$	

Phrase Level Recovery

- Filling blank entries by pointers to error routines
- On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue.
- For example, in case of an error like replacing a coma by a semicolon, it will report the error, generate the “;” and continue.

Error Production

- If we have an idea of common errors that might occur, we can include the errors in the grammar at hand.

Eg: E → + E| - E | * A | /A

In the grammar last two productions are error productions

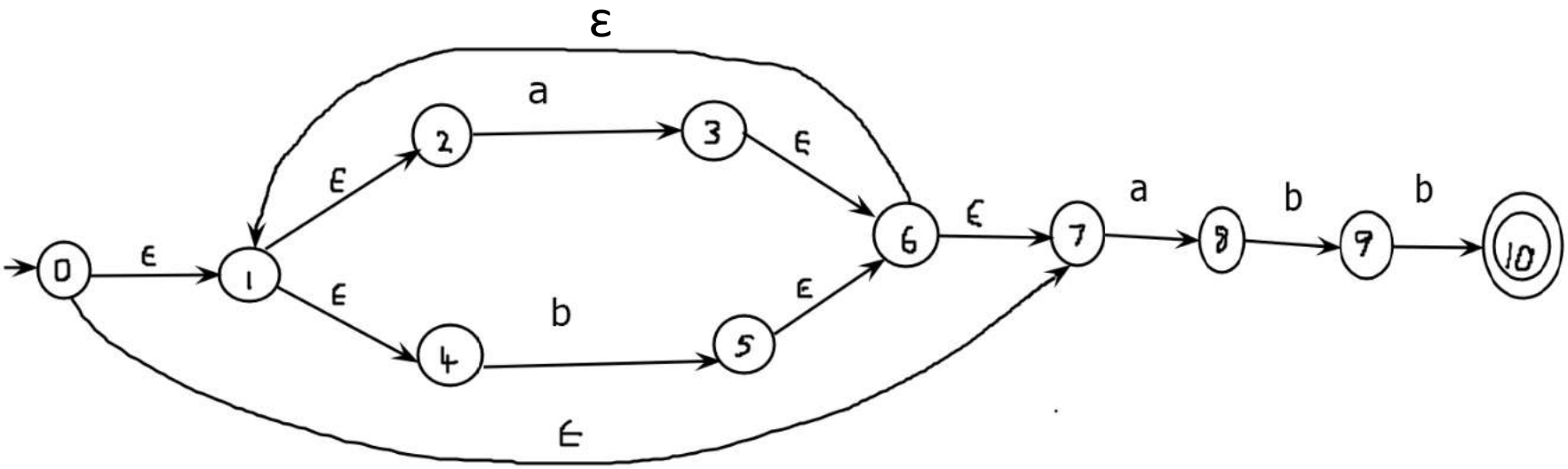
Global Correction

- There are algorithms for choosing a minimal amount of changes to obtain a globally least-cost correction.
- The parser examines the whole program and tries to find out the closest match for it which is error-free.
- When an erroneous input (statement) X is found, it creates a parse tree for some closest error-free statement Y.
- The closest match program has less number of insertions, deletions, and changes of tokens to recover from erroneous input.
- High time and space complexity

NFA_to_DFA Construction

NFA

R= (a+b)*abb



ε-Closure

- ε-transition or $\wedge$ - transition means present state can goto other state without any input.

ε-closure(s)

1. C = { s}

2. Take ε-transitions on each state in C and update C with those states

3. Repeat step 2 until no new state to be added to C

4. Return C

Steps for converting ε-NFA to DFA
(subset construction method)

1. Take the ε -closure for the start state of NFA as a start state of DFA and initialize D(set of states for DFA)

2. Take an unmarked state from D and find union of transitions for each input symbol that can be traversed (state T is identified for each input symbol) and take ε-closure(T)

3. If found a new state, add to D

4. Update transition table for DFA

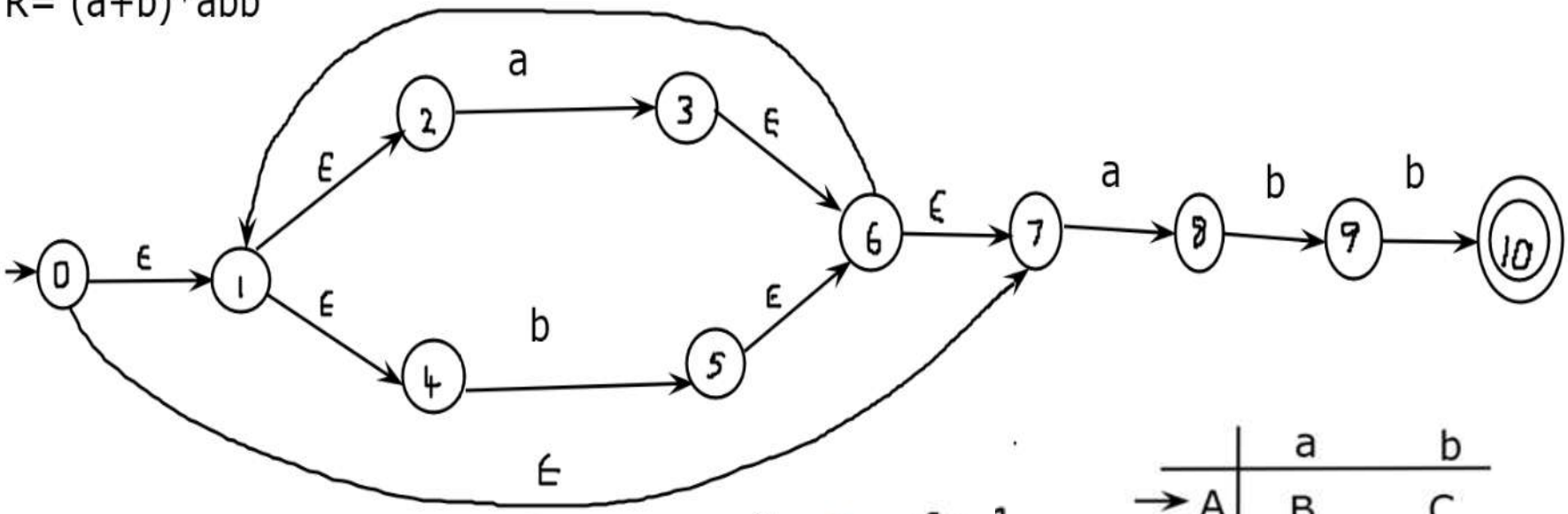
5. Repeat step 2, 3 & 4 until all state of D are marked.(ie no new state is present in the transition table of DFA)

6. Mark the states of DFA as a final state which contains the final state of NFA.

Illustration

ε

R= (a+b)*abb



	a	b
→ A	B	C

A = ε - closure(0) = {0, 1, 2, 4, 7} S = {A}

a on A

T = {3, 8} , ε - closure(T) = {3, 6, 7, 1, 2, 4, 8} = B

b on A

T = {5} , ε - closure(T) = {5, 6, 7, 1, 2, 4} = C

Illustration (Contd.)

S = { A,B,C }

	a	b
→ A	B	C
B	B	D

a on B

T= {3,8} , ∈-closure (T) = B

b on B

T = {5,9} , ∈- closure (T) = {5,6,7,1,2,4,9} = D

a on C

T = {3,8}, ∈- closure(T) = B

b on C

T = {5}, ∈- closure(T) = C

	a	b
→ A	B	C
B	B	D
C	B	C

S = { A,B,C,D}

Illustration (Contd.)

	a	b
→ A	B	C
B	B	D
C	B	C
D	B	E

a on D

T = {3, 8}, ϵ- closure(T) = B

b on D

T = {5, 10}, ϵ- closure(T) = {5, 6, 7, 1, 2, 4, 10} = E (final state)

a on E

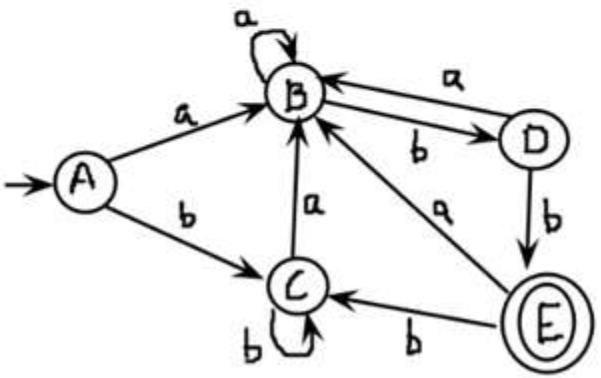
T = {3, 8}, ϵ- closure(T) = B

b on E

T = {5}, ϵ- clousre (T) = C

S = {A,B.C.D.E}

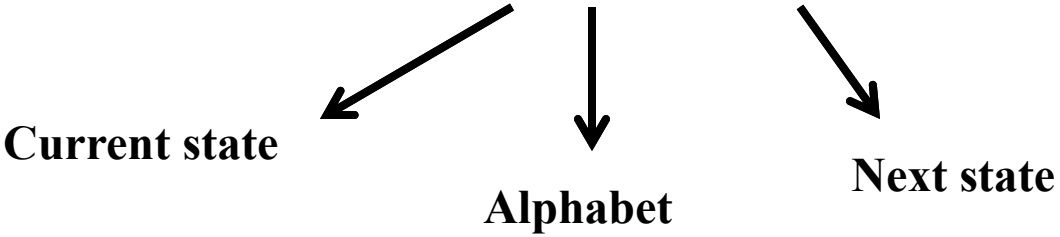
	a	b
→ A	B	C
B	B	D
C	B	C
D	B	E
(E)	B	C



Transition function

- Function that defines the transitions of a state in a system
- Represented by δ
- The transition function δ, receives 2 arguments : Current state $q \in Q$ and input alphabet $a \in \sum$

$$\delta (q,a) = p$$



Properties of Transition Function of DFA

- Property 1

$$\delta(q, \varepsilon) = q$$

- Property 2 (Extended transition function)

a) $\hat{\delta}(q, aw) = \hat{\delta}(\delta(q, a), w)$

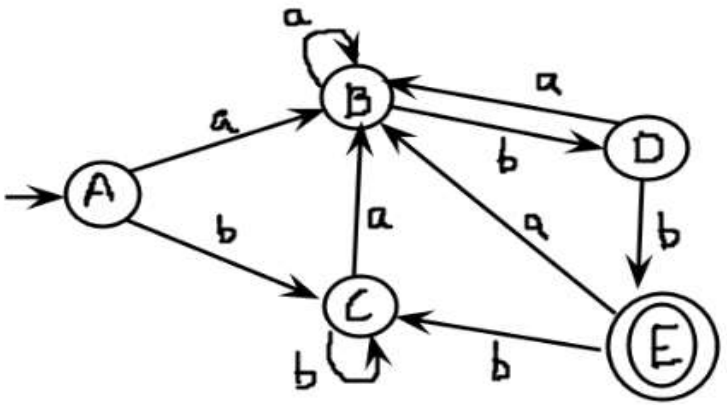
b) $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$

Eg: Let Q={p,q,r}, ∑= {0,1}, $\delta(q, 0) = p$, $\delta(p, 1) = r$, w=01

$\hat{\delta}(q, 01) = \delta(\delta(q, 0), 1) = \delta(p, 1) = r$

Acceptance of DFA

	a	b
→ A	B	C
B	B	D
C	B	C
D	B	E
(E)	B	C



Q= {A,B,C,D,E}, ∑= {a,b}, q_0 = A, F = {E}

Q x ∑ →Q

δ(A, a) = B, δ(A, b) = C similarly for all states

Acceptance

If $\delta(q_0, w) = q$ for some $q \in F$ ie reachability to any final state from the initial state or start state on string w

Acceptance of DFA (contd.)

Input ababb, take it as aw and apply property 2a

$\delta(A, ababb) = \delta(\delta(A, a), babb) = \delta(B, w)$

where w = babb then perform δ(B, babb)

Ie δ(A, ababb) = δ(B, babb) = δ(D, abb) = δ(B, bb)

= δ(D, b) = δ(E, ε) = E - accepted

Input bab

δ(A,bab) = δ(C,ab) = δ(B,b) = δ(D, ε) = D - not accepted

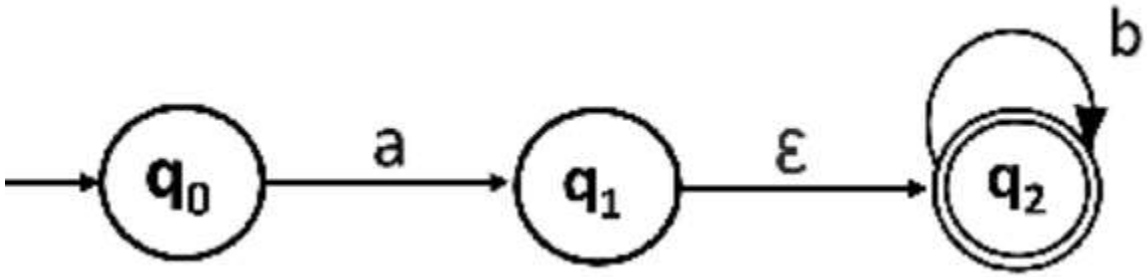
Eliminating ε Transitions

1. Find out all the ε transitions from each state from Q. That will be called as ε-closure(q_i) where $q_i \in Q$
2. Then δ' transitions can be obtained for each input symbol. The δ' transitions mean ε-closure on δ moves and it won't contain any ε-transitions

$$\delta'(q_i,a) = \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q_i), a))$$ where $q_i \in Q$, $a \in \sum$

3. Using the resultant transitions according to δ', the transition table for equivalent NFA without ε can be built.

ε - NFA



states	a	b	ε
→ q0	{q1}	Ø	Ø
q1	Ø	Ø	{q2}
(q2)	Ø	{q2}	Ø

Elimination

ε-closure(q0) = {q0}
ε-closure(q1) = {q1, q2}
ε-closure(q2) = {q2}

δ'(q0, a) = ε-closure(δ(ε-closure(q0),a))
= ε-closure(δ({q0}, a))
= ε-closure({q1})
= {q1, q2}

δ'(q0, b) = ε-closure(δ(ε-closure(q0),b))
= ε-closure(δ({q0}, b))
= ε-closure(Φ)
= Φ

Elimination (contd.)

$\delta'(q1, a) = \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q1), a))$

$= \varepsilon\text{-closure}(\delta(\{q1, q2\}, a))$

$= \varepsilon\text{-closure}(\delta(q1, a) \cup \delta(q2, a))$

$= \varepsilon\text{-closure}(\Phi \cup \Phi)$

$= \Phi$

$\delta'(q1, b) = \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q1), b))$

$= \varepsilon\text{-closure}(\delta(\{q1, q2\}, b))$

$= \varepsilon\text{-closure}(\delta(q1, b) \cup \delta(q2, b))$

$= \varepsilon\text{-closure}(\Phi \cup \{q2\})$

$= \{q2\}$

Elimination (contd.)

$\delta'(q2, a) = \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q2), a))$

$= \varepsilon\text{-closure}(\delta(\{q2\}, a))$

$= \varepsilon\text{-closure}(\Phi)$

$= \Phi$

$\delta'(q2, b) = \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q2), b))$

$= \varepsilon\text{-closure}(\delta(\{q2\}, b))$

$= \varepsilon\text{-closure}(\{q2\})$

$= \{q2\}$

Elimination (contd.)

Summarized

$\delta'(q0, a) = \{q1, q2\}$

$\delta'(q0, b) = \Phi$

$\delta'(q1, a) = \Phi$

$\delta'(q1, b) = \{q2\}$

$\delta'(q2, a) = \Phi$

$\delta'(q2, b) = \{q2\}$

$F = \{ q1, q2\}$

Since ε-closure of q1 and q2 contain the final state q2

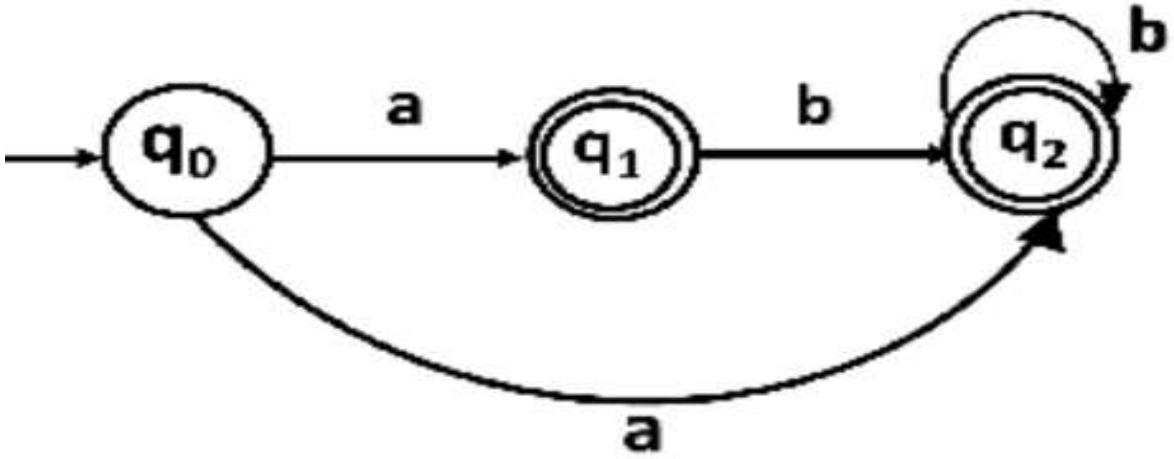
Elimination (contd.)

Transition Table for NFA without ε

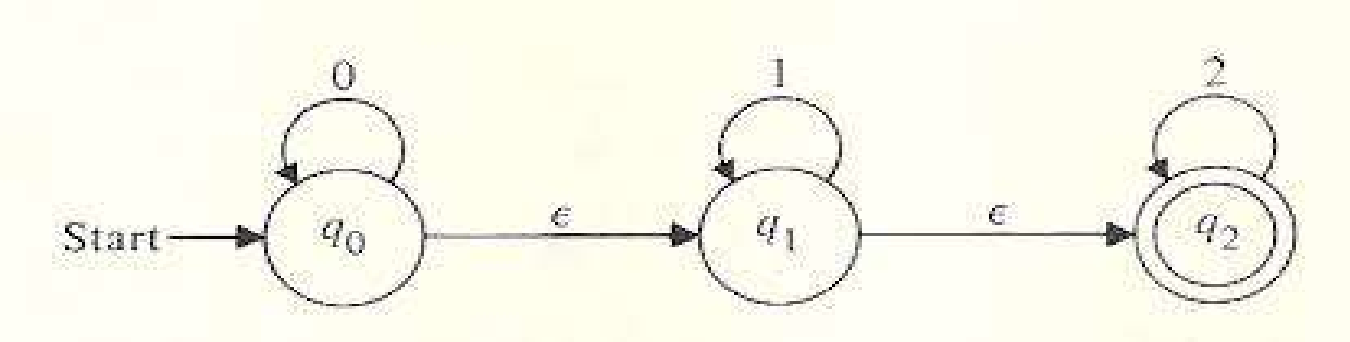
states	a	b
→ q0	{q1,q2}	Ø
q1	Ø	{q2}
q2	Ø	{q2}

Elimination (contd.)

Transition diagram for NFA without ε



Exercise



ε-closure(q0) = {q0,q1,q2}

ε-closure(q1) = {q1, q2}

ε-closure(q2) = {q2}

F={q0,q1,q2}

$\delta'(q_i,a)$ = ε-closure(δ(ε-closure (q_i) , a)) where $q_i \in Q$, $a \in \sum$

ie $\delta'(q_0,0)$ = ε-closure(δ(ε-closure (q_0) , 0))

= ε-closure(δ({q0,q1,q2} , 0)) = ε-closure({q0}) = {q0,q1,q2}

Steps for converting NFA without ε to DFA

1. Start state of NFA as a start state of DFA and initialize D(set of states for DFA)

2. Take an unmarked state from D and find union of transitions for each input symbol that can be traversed(state T is identified for each input symbol)

3. If found a new state, add to D

4. Update transition table for DFA

5. Repeat step 2, 3 & 4 until all state of D are marked.(ie no new state is present in the transition table of DFA)

6. Mark the states of DFA as a final state which contains the final state of NFA.

NFA and DFA Equivalence

Theorem If $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ is the DFA constructed from NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ by the subset construction, then $L(D) = L(N)$.

PROOF: What we actually prove first, by induction on $|w|$, is that

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$$

BASIS: Let $|w| = 0$; that is, $w = \epsilon$. By the basis definitions of $\hat{\delta}$ for DFA's and NFA's, both $\hat{\delta}_D(\{q_0\}, \epsilon)$ and $\hat{\delta}_N(q_0, \epsilon)$ are $\{q_0\}$.

NFA and DFA Equivalence(contd…)

INDUCTION: Let w be of length $n+1$, and assume the statement for length n. Break w up as $w = xa$, where a is the final symbol of w. By the inductive hypothesis, $\hat{\delta}_D(\{q_0\}, x) = \hat{\delta}_N(q_0, x)$.

$$\hat{\delta}_D(\{q_0\}, x) = \{p_1, p_2, \ldots, p_k\} \quad (1)$$

The inductive part of the definition of $\hat{\delta}$ for NFA's tells us

$$\hat{\delta}_N(q_0, w) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \quad (2)$$

The subset construction, on the other hand, tells us that

$$\delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \quad (3)$$

NFA and DFA Equivalence(contd…)

Using Equations (1) and (3)

inductive part of the definition of $\hat{\delta}$ for DFA's:

$$\hat{\delta}_D(\{q_0\}, w) = \delta_D\big(\hat{\delta}_D(\{q_0\}, x), a\big) = \delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \qquad (4)$$

Thus, Equations (2) and (4) demonstrate that $\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$. When we observe that D and N both accept w if and only if $\hat{\delta}_D(\{q_0\}, w)$ or $\hat{\delta}_N(q_0, w)$, respectively, contain a state in F_N, we have a complete proof that $L(D) = L(N)$. $\square$

Regular Grammar

If all productions are of the form $A \to wB$ or $A \to w$, where A and B are variables and w is a (possibly empty) string of terminals, then we say the grammar is *right-linear*. If all productions are of the form $A \to Bw$ or $A \to w$, we call it *left-linear*. A right- or left-linear grammar is called a *regular grammar*.

Example 1 The language $\mathbf{0(10)^*}$ is generated by the right-linear grammar

$$S \to 0A$$

$$A \to 10A \mid \varepsilon$$

and by the left-linear grammar

$$S \to S10 \mid 0$$

Equivalence of Regular grammars and Finite Automata

Theorem If L has a regular grammar, then L is a regular set.

Proof First, suppose $L = L(G)$ for some right-linear grammar $G = (V, T, P, S)$. We construct an NFA with ϵ-moves, $M = (Q, T, \delta, [S], \{[\epsilon]\})$ that simulates derivations in G.

Q consists of the symbols $[\alpha]$ such that α is S or a (not necessarily proper) suffix of some right-hand side of a production in P.

We define δ by:

1) If A is a variable, then $\delta([A], \epsilon) = \{[\alpha] \mid A \to \alpha \text{ is a production}\}$.
2) If a is in T and α in $T^* \cup T^*V$, then $\delta([a\alpha], a) = \{[\alpha]\}$.

Then an easy induction on the length of a derivation or move sequence shows that $\delta([S], w)$ contains $[\alpha]$ if and only if $S \overset{*}{\Rightarrow} xA \Rightarrow xy\alpha$, where $A \to y\alpha$ is a production and $xy = w$, or if $\alpha = S$ and $w = \epsilon$. As $[\epsilon]$ is the unique final state, M accepts w if and only if $S \overset{*}{\Rightarrow} xA \Rightarrow w$.

Proof(contd…)

Let G= (V, T,P,S) be a left linear grammar and G'= (V, T,P',S) be the grammar with productions reversed

$$P' = \{A \to \alpha \mid A \to \alpha^R \text{ is in } P\}.$$

If we reverse the productions of a left-linear grammar we get a right-linear grammar, and vice versa. Thus G' is a right-linear grammar, and it is easy to show that $L(G') = L(G)^R$. By the preceding paragraph, $L(G')$ is a regular set. But the regular sets are closed under reversal

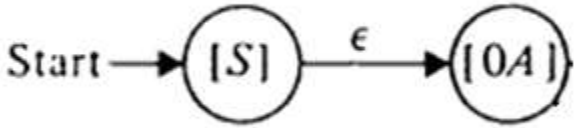
so $L(G')^R = L(G)$ is also a regular set.

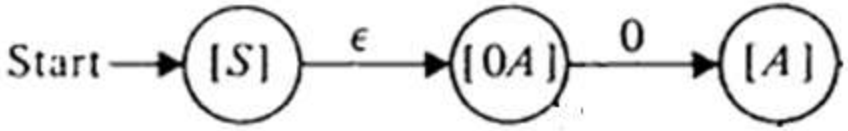
Thus every right- or left-linear grammar defines a regular set. □

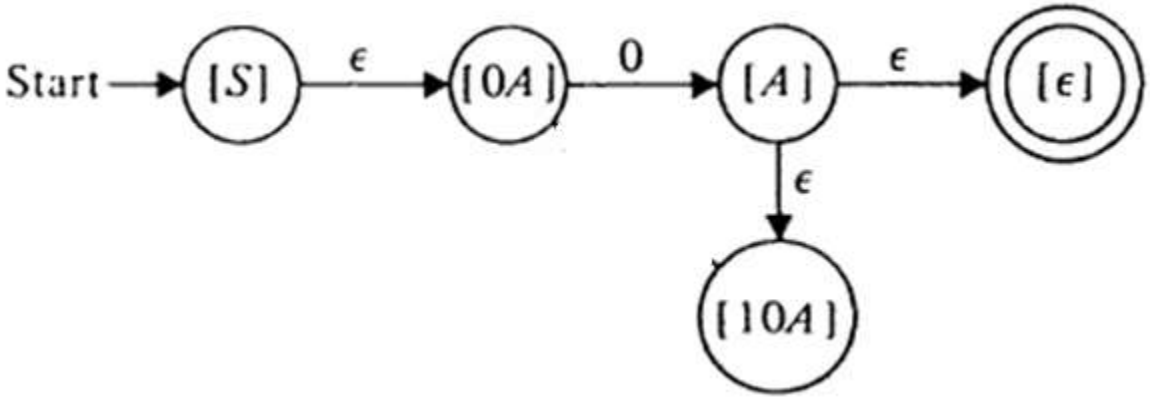
Example 1

$S \rightarrow 0A$

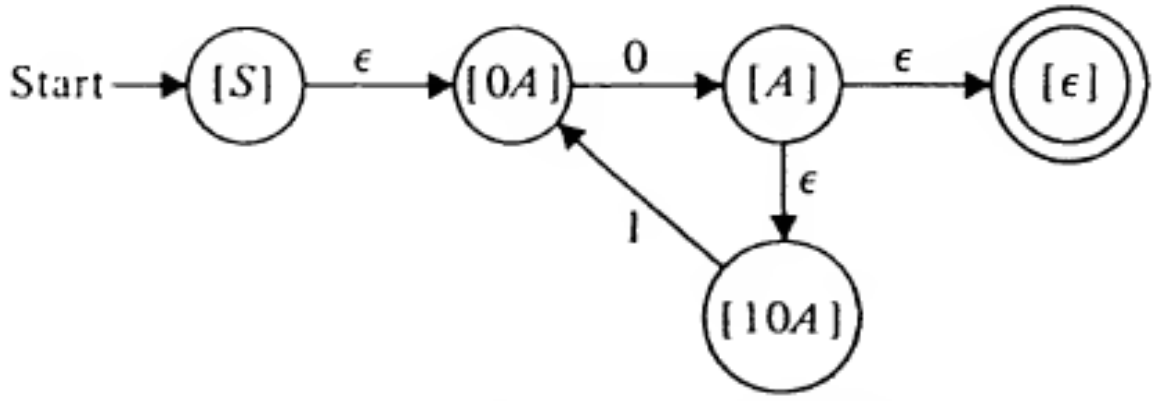
$A \rightarrow 10A \mid \varepsilon$







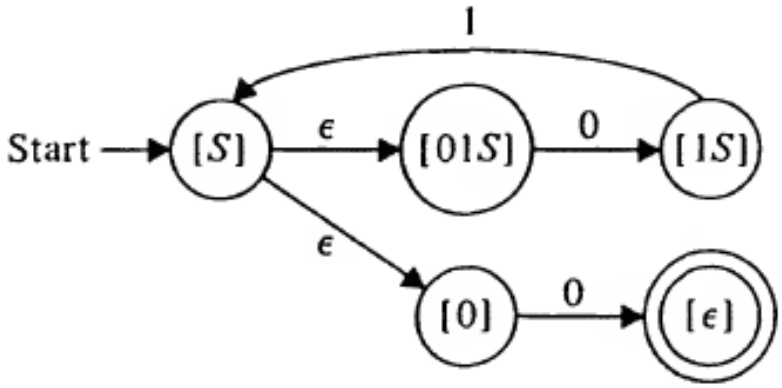
Example 1 (contd…)



Find the DFA for above automata

Example 2

S → 01S|0



Find the DFA for above automata