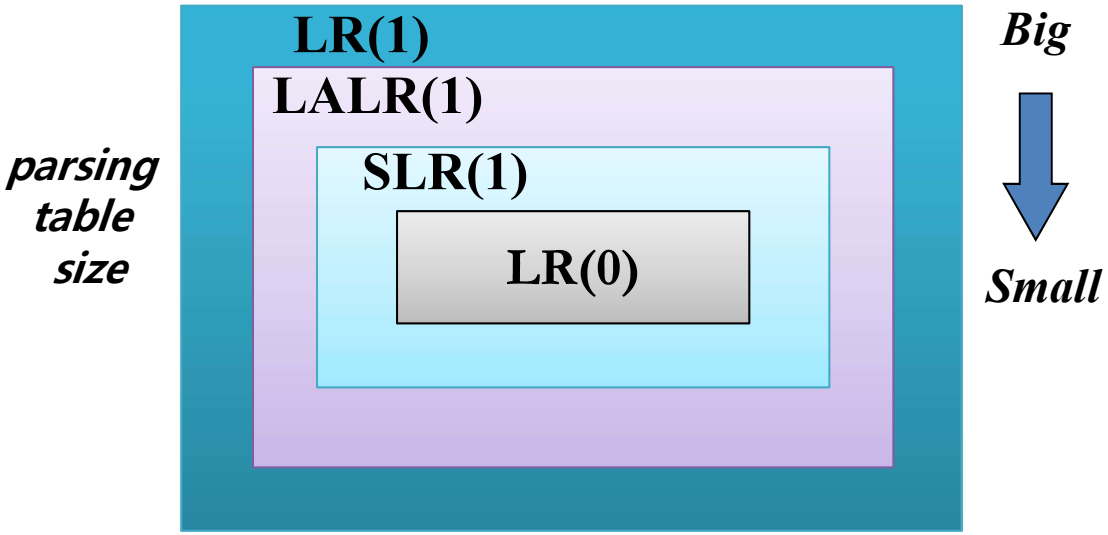
Chapter 6

상향식 파싱 알고리즘 - **Part I**

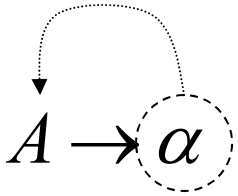
Table of Contents

Overview

- **LR(0) : No *look-ahead***
- **SLR(1) : *Simple* LR(1)**
- **LALR(1) : *Lookahead* LR(1)**
  - More powerful than SLR(1) but less complex than LR(1)
- **LR(1) (또는 Canonical LR(1))**



LR parser

- **Bottom-Up 방식(R means *rightmost* derivation)**
- ***Shift-Reduce* Parsing**
- **Production rule is applied *backwards*.** 

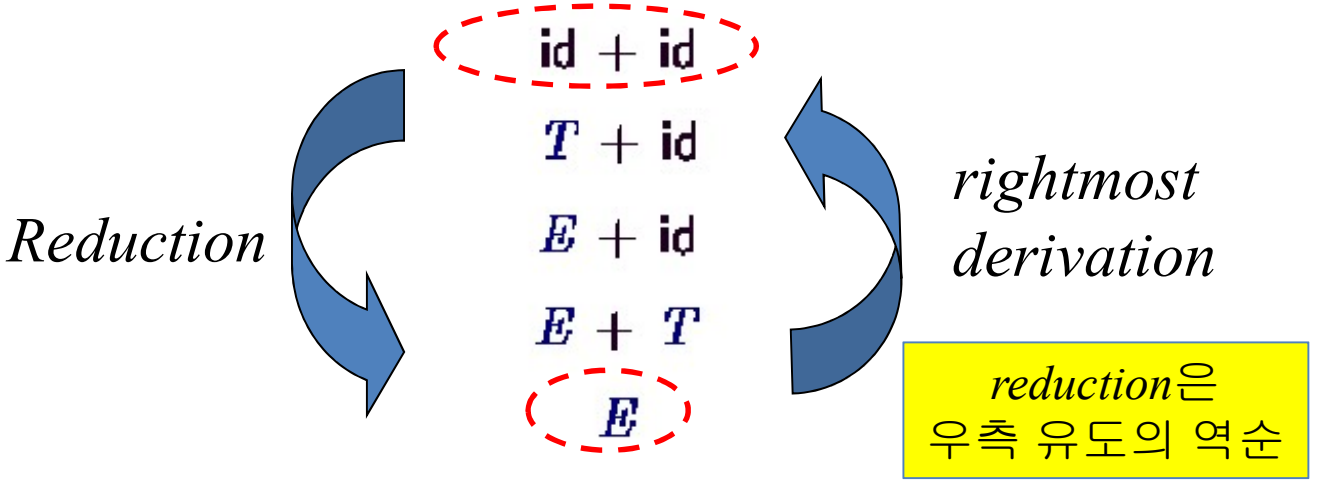
  - 텅 빈 stack에서 시작
  - One or more input symbols are *moved* onto the stack (***shift***)
  - stack 에 쌓인 기호(α)를 nonterminal A 로 바꿈(***reduce***)
    - **Pop α → Push A**
    - 문법에 맞는 문장일 경우
      - (parsing이 끝나면) stack 에 *start symbol* S 만 남게 됨.

Bottom-Up Parsing (1/2)

Given a stream of tokens w, *reduce* it to the start symbol.

E	$\longrightarrow$	$E+T$
E	$\longrightarrow$	T
T	$\longrightarrow$	id

Parse input stream: id + id:

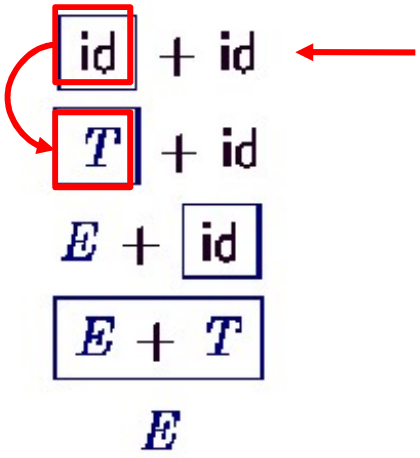


Bottom-Up Parsing (2/2)

A structure that furnishes a means to perform reductions.

E	$\longrightarrow$	$E+T$
E	$\longrightarrow$	T
T	$\longrightarrow$	id

Parse input stream: id + id:



두 개의 *id* 중 왜 왼쪽 *id*를 선택했을까?

Do we need preprocessing ?

- **Definitely, YES!**
- **LL parsing algorithm을 적용하기 위해서는**
  - 주어진 문법에 대한 전처리 과정이 필요
    - 예: left-recursive rule을 right recursive rule로 변환
- **LR parsing algorithm에서도 전처리 과정이 필요**
  - Single production rule 추가: `S'` → `S`
    - Start symbol이 `S`에서 `S'`으로 바뀜
  - 이렇게 바뀐 문법을 확장 문법(augmented grammar)이라 부름

예 1

■ 확장 문법: S′ → S
S → (S) S | ε

Parsing Stack	Input	Action
$	()$	shift
$(	)$	reduce S→ ε
$(S	)$	shift
$(S)	$	reduce S→ ε
$(S)S	$	reduce S→(S)S
$S	$	reduce S′→S
$S′	$	accept

- 같은 S이지만, 각각 다른 생성 규칙을 적용하여 reduce함.
- 어떤 rule을 적용할지 판단하기 위해 스택에 저장된 기호를 참조 (*stack look-ahead*)

예 2

■ **확장 문법:** E′ → E

E → E + n | n

	Parsing Stack	Input	Action
1	$	n+n$	shift
2	$n	+n$	reduce E → n
3	$E	+n$	*shift*
4	$E+	n$	shift
5	$E+n	$	reduce E→ E + n
6	$E	$	*reduce* E′ → E
7	$E′	$	accept

- 스택에 똑같이 E(단계 3,6)가 있지만, 전혀 다른 action이 발생
  - *input lookahead*가 다르기 때문 : + (단계 3) 와 $ (단계 6)

General Observations (1/2)

- **A shift-reduce parser traces out a rightmost derivation of the input string**
  - But the steps of the derivation occur in *reverse order*.
    - 예1: S′ ⇒ S ⇒ (S) S ⇒ (S) ⇒ ( )
    - 예2: E′ ⇒ **E** ⇒ **E + n** ⇒ **n + n**

- ***right sentential form*** **(우 문장 형태)**
  - 우측 유도 과정에서 만들어지는 문자열
  - parsing stack의 문자열과 입력 버퍼에 남은 문자열로 구분

    - **E**‖**+n** (step 3, ‖는 stack 과 input buffer간 구분 기호)
    - **E+**‖**n** (step 4)

General Observations (2/2)

- ***viable prefix*** : stack 에 있는 문자열(=*handle*)
  - **E**, **E +**, **E + n** 은 우 문장 형태 **E + n** 의 *viable prefix*

Parsing stack	Input
$	n + n $
$ n	+ n $
$ E	+ n $
$ E +	n $
$ E + n	$

ε ‖ n + n
n ‖ + n
E ‖ + n

  - ε 과 n + 는 *viable prefix*인가?

viable : 실행 가능한. (태아,신생아가) 살아갈 수 있는, 생명력이 있는

예 3: 우단 유도와 handle 찾기

- 입력 문장 **id + id * id**에 대해 우단 유도 과정을 보이고 핸들을 찾아보자.

① E → E + T　　② E → T　　③ T → T * F

④ T → F　　⑤ F → (E)　　⑥ F → id

우문장 형태	핸들	감축에 사용되는 생성 규칙
id1 + id2 * id3	id1	F → id
F + id2 * id3	F	T → F
T + id2 * id3	T	E → T
E + id2 * id3	id2	F → id
E + F * id3	F	T → F
E + T * id3	id3	F → id
E + T * F	T * F	T → T * F
E + T	E + T	E → E + T
E		

LR parsing algorithm 개요(1/2)

어떻게 **handle** 을 찾을 수 있나요?

$$A \rightarrow \alpha$$

생성규칙의 **RHS** (*Right Hand Side*)
에 해당하는 문자열 α를 찾는다.

shift 할지 ***reduce*** 할지 어떻게 구분하나요?

$$A \rightarrow X_1 X_2 \cdots X_i X_{i+1} \cdots X_n \; (\alpha = X_1 X_2 \cdots X_n)$$

생성규칙의 **RHS** 에서
일부만 찾았으면 ***shift*** 하고, 다 찾은 경우에만 ***reduce*** 한다.

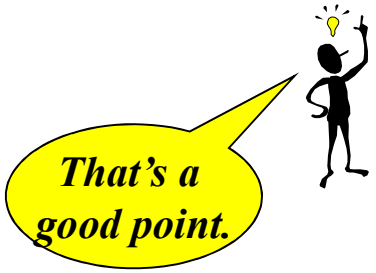
LR 파싱 알고리즘 개요(2/2)

handle 을 다 찾았는지 못 찾았는지는 어떻게 알 수 있나요?

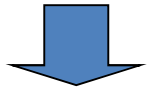
어디까지 찾았는지 ***marking*** 한다.

$$A \rightarrow \bullet X_1 X_2 \cdots X_i X_{i+1} \cdots X_n \, (initial, shift)$$
$$A \rightarrow X_1 X_2 \cdots X_i \bullet X_{i+1} \cdots X_n \, (shift)$$
$$A \rightarrow X_1 X_2 \cdots X_i X_{i+1} \cdots X_n \bullet (reduce)$$

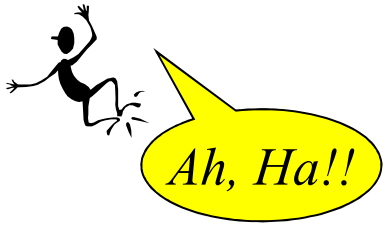
모든 생성 규칙에 대해 일일이 ***marking*** 하면 그 수가 너무 많아지지 않나요?



비슷한 성질을 갖는 것끼리 묶으면 된다.



state(상태)



LR(0) Items (1/2)

- **LR (0) item : rule의 RHS 에 dot (.)를 갖는 생성 규칙**
  - 숫자 '0'의 의미 : 입력을 참조하지 않음
    - "입력을 참조한다"
      - 입력 버퍼에서 token(=*lookahead*)을 읽어 와서 shift/reduce를 결정할 때 이용

A → XYZ 이면, 생성 가능한 LR(0) item은 모두 4개

A → . X Y Z

A → X . Y Z

A → X Y . Z

A → X Y Z .

LR(0) Items (2/2)

- **A → α . β 에서 *dot* 의 의미**

  α : 현재 parsing stack에 저장되어 있음. *reduce* 시킬 대상.

  β : 입력 버퍼에 아직 남아 있음. *shift* 시킬 대상.

  mark symbol : dot 다음에 나오는 기호

  $A \rightarrow \alpha . X\eta$ 에서 mark symbol은 **X**

- **Initial (*closure*) item : *A* → . α**
- **complete (*reduce*) item : *A* → α .**
  - 생성 규칙의 RHS 에 해당하는 문자열 α 를 다 찾은 상태

LR(0) Items : 예 4

- ***예 4(a)***

S′ → S

S → (S) S | ε

S′ → . S	S′ → S .
S → .(S) S	S → (. S) S
S → (S .) S	S → (S) . S
S → (S) S .	S → .

- ***예 4(b)***

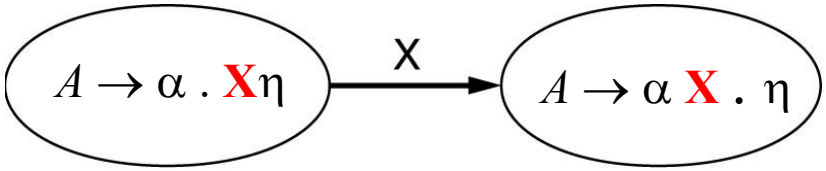
E′ → E

E → E + n | n

E′ → . E	E′ → E .
E → . E + n	E → E . + n
E → E + . n	E → E + n .
E → . n	E → n .

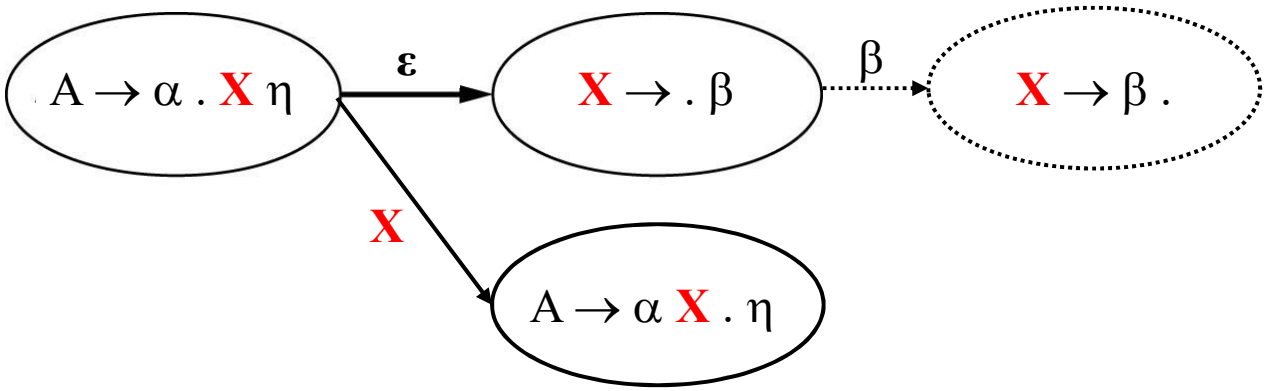
Finite Automata of LR(0) Items (1/3)

- ***state*** = **LR(0) items들의 집합**
- **상태 천이**
  - If **X** is ***token***, push **X** onto the stack



Finite Automata of LR(0) Items (2/3)

- ***state*** = **LR(0) items들의 집합**
- **상태 천이**
  - If **X** is a ***nonterminal ,***
    - Push **X** 를 위해 $X \rightarrow . \beta$ 부터 시작



    - recognition of β ➔ *reduce by* $X \rightarrow \beta$ ➔ **X** 를 stack에 push

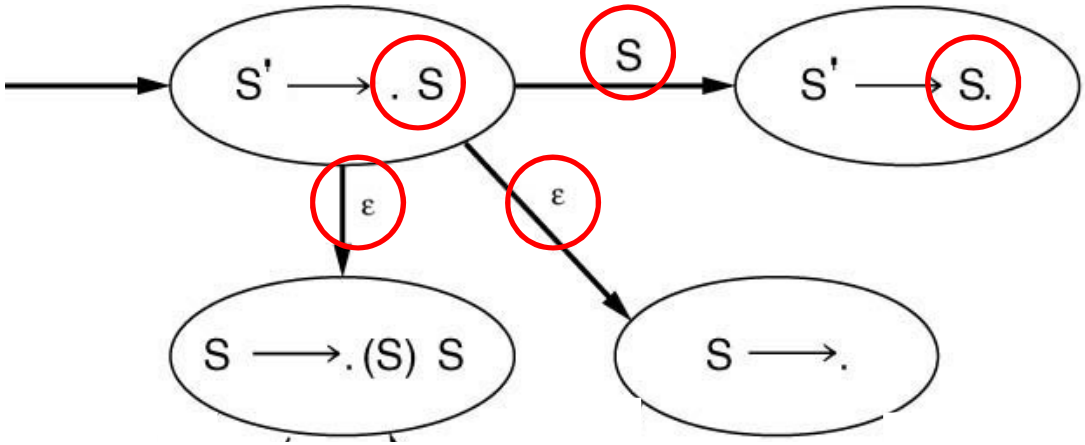
Finite Automata of LR(0) Items (3/3)

- **시작 상태 : S′ → .S**
  - 왜 확장 문법이 필요한가?
    - S → α | β 인 경우

      S → . α 와 S → . β 중 어느 것을 시작 상태로 할까?
    - S′ → S 는 *single* production이므로 시작 상태는 S′ → .S

- **No *accepting* states**
  - 상태 천이를 통해 parsing 과정을 추적해 갈 뿐 인식 여부를 판단하지는 않음
    - Accept 여부는 오토마타가 아니라 **parsing algorithm에서 결정**

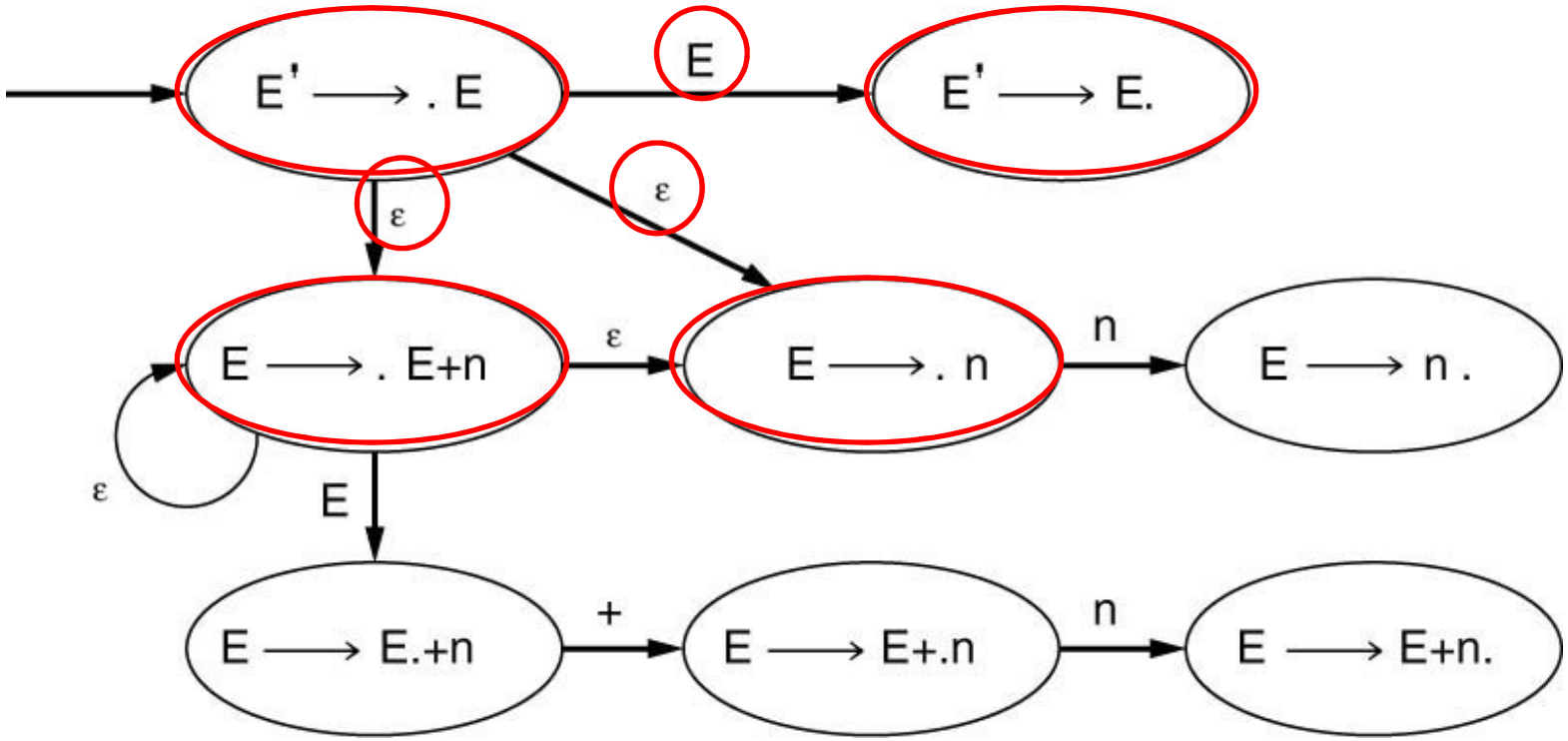
예 5



S′ → S
S → (S) S | ε

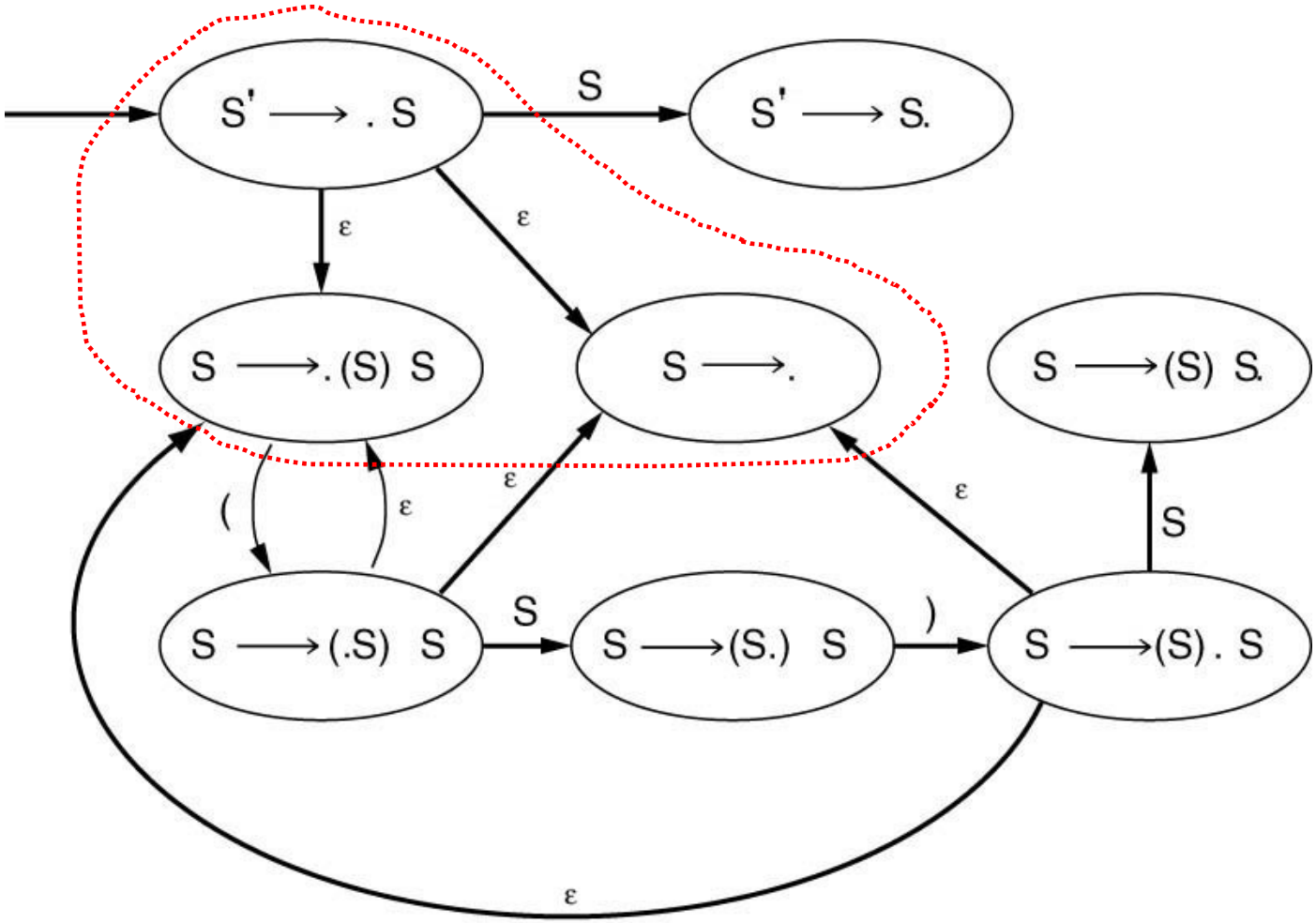
예 6

$$E' \rightarrow E$$
$$E \rightarrow E + n \mid n$$

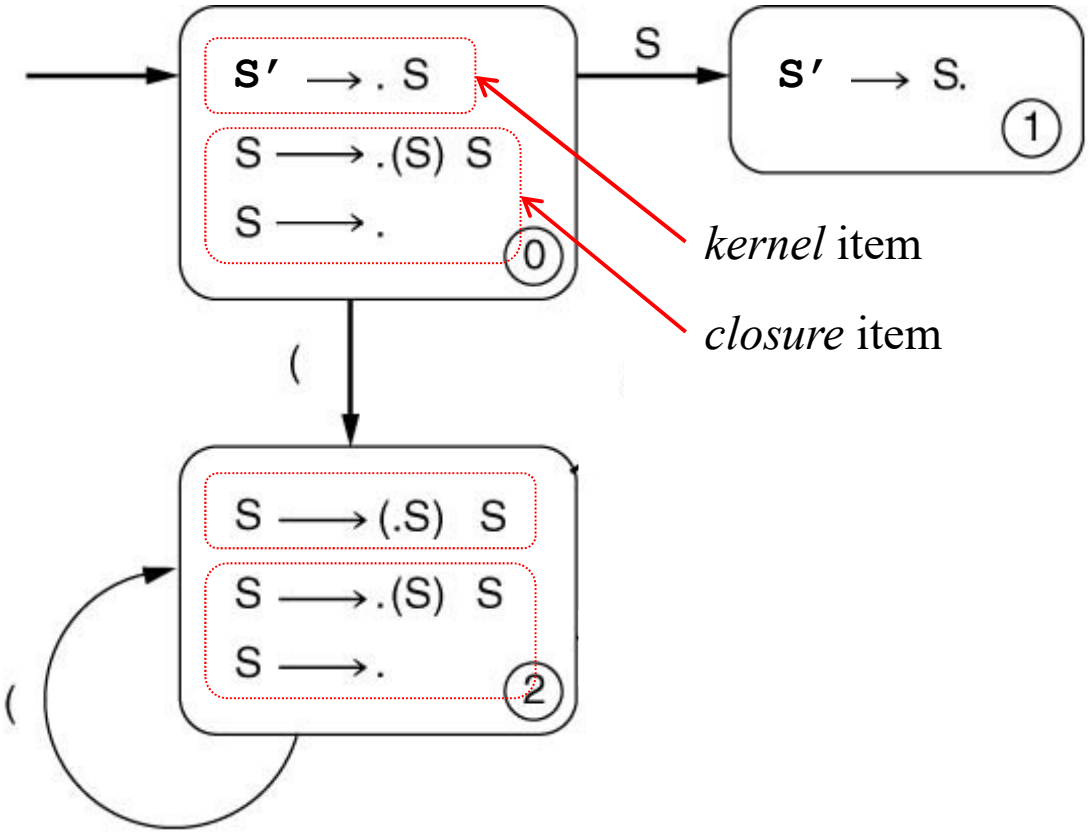


예 7 (1/3): DFA of LR(0) Items

DFA 로 변환 : ε-closure를 구함



예 7 (2/3): DFA of LR(0) Items




```
S′ → S
S  → ( S ) S | ε
```

CLOSURE(i)

- **CLOSURE(i) : i 는 kernel item**
  - **Kernel item** i 는 CLOSURE(i)에 포함
  - [**A → α•Bβ**] ∈ CLOSURE(i) 이고, **B → γ** ∈ P 일 때
    - **[B → .γ] 를** CLOSURE(i)에 추가
    - 새로운 LR(0) item이 CLOSURE(i)에 추가되지 않을 때까지 반복 적용

  - **CLOSURE(i) = CLOSURE(i) ∪**
    {[B → .γ] | [A → α.Bβ] ∈ CLOSURE(i), B→γ ∈P}

예 8

E′ → E **E → E + T | T**

T → T * F | F **F → (E) | id**

- CLOSURE([**E′ →•E**]) =

  {[**E′ →•E**], [E →•E + T}, [E →•T], [T →•T * F], [T →•F], [F →•(E)], [F →•id] }

- CLOSURE([**E → E +•T**]) =

  { [**E → E +•T**], [T →•T * F], [T →•F], [F →•(E)], [F →•id] }

- CLOSURE([**E →•T**]) =

  { [**E →•T**], [T →•T * F], [T →•F], [F →•(E)], [F →•id] }

상태 천이 : GOTO 함수

- $GOTO(I_i, X) = I_j$
  - I_i : 현재 상태 , I_j : 다음 상태, $X \in V = (V_N \bigcup V_T)$

$$GOTO(I_i, X) = CLOSURE(\{ [A \rightarrow \alpha X \bullet \beta] \mid [A \rightarrow \alpha \bullet X\beta] \in I_i \})$$

- **예 9**

E′ → E **E → E + T | T**

T → T * F | F **F → (E) | id**

- I = { [E′ → E•], [E → E• + T] }일 때 GOTO(I, +)를 구하시오.

GOTO(I, +) = CLOSURE([E → E + •T]}

= {[E → E + •T], [T →•T * F], [T →•F], [F →•(E)], [F →•id]}

DFA 상태

DFA 상태 = *kernel item* + *closure item*

kernel item

- 상태 천이 과정에서 생성된 item
- 상태를 정의하고 상태 천이를 결정하는 주체

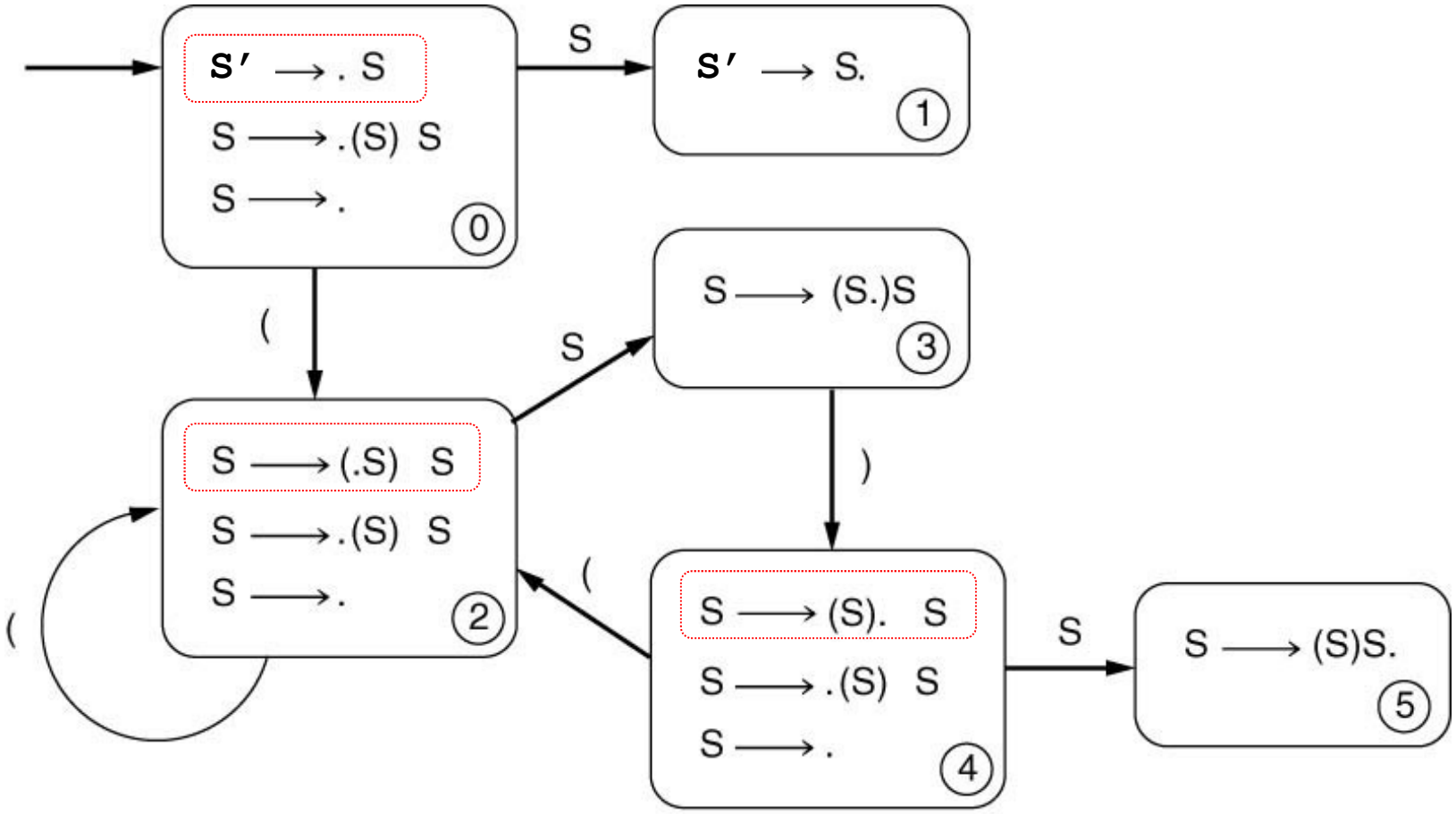
closure item

- ε-transition을 통해 추가된 item
  - *kernel item*에 의해 자연스럽게 추가된 item

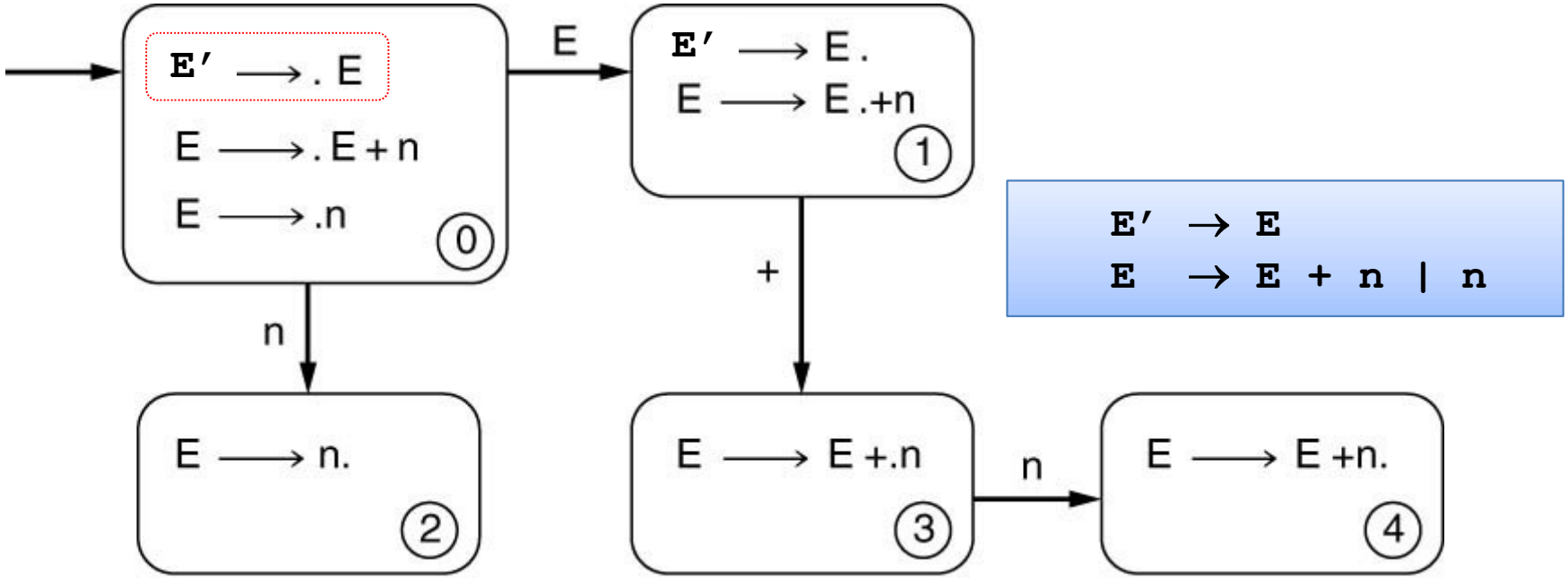
DFA 구성 방법

- **시작 상태 = CLOSURE([S′ → .S])**
  - mark symbol이 nonterminal 일 경우
  - *kernel* item 외에 *closure* item을 함께 포함
- **현재 상태 I 에 속한 LR(0) item [$A \rightarrow \alpha . X\beta$] 에서 mark symbol을 찾는다**
  - 중복되지 않은 mark symbol 개수 만큼 상태 천이가 발생
- **다음 상태 J = GOTO(I, X)**
  - dot를 mark symbol **X**의 오른쪽으로 옮겼을 때
    - **CLOSURE([A → α X.β])** 를 구한다.
  - reduce item이면 다음 상태는 없음
- **더 이상 다음 상태를 구할 수 없으면 종료**

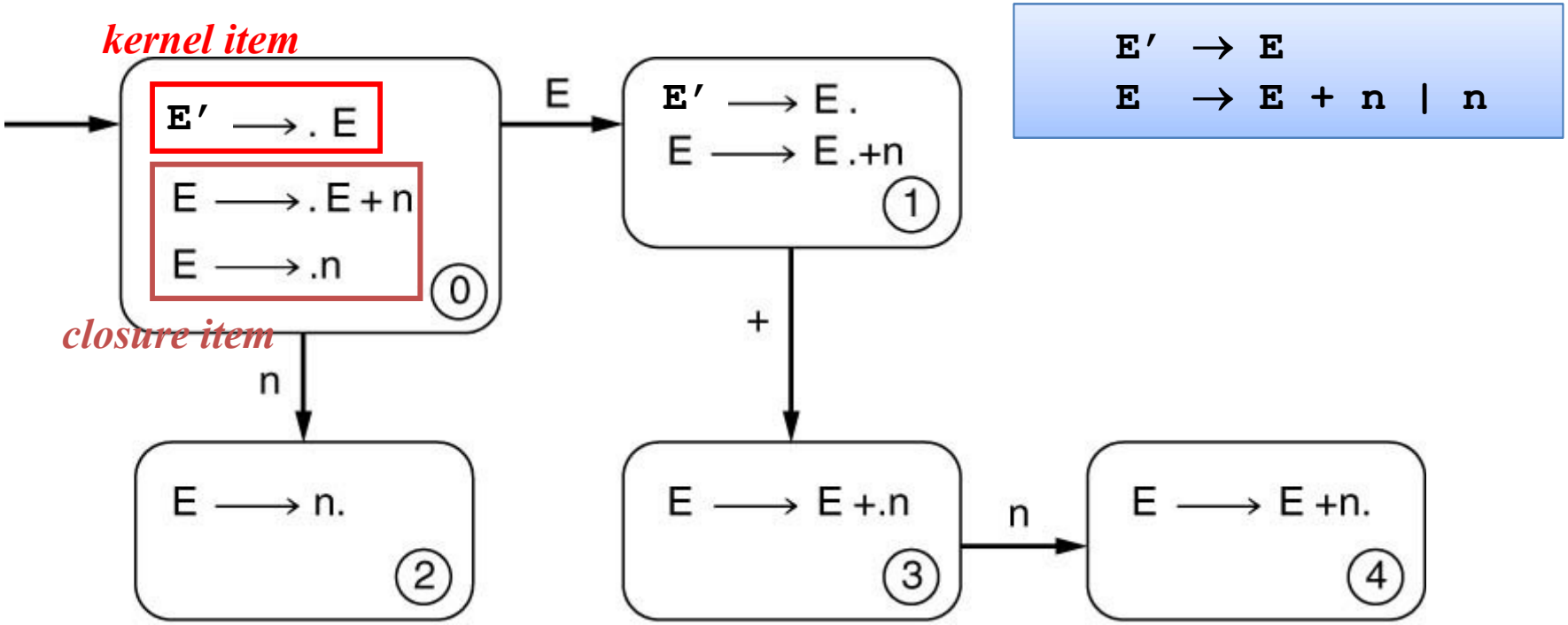
예 7 (3/3): DFA of LR(0) Items



예 10: DFA of LR(0) Items (1/2)



예 10: DFA of LR(0) Items (2/2)



The LR(0) Parsing Algorithm (1/3)

- **Parsing 알고리즘 = DFA의 상태 추적**
  - shift 또는 reduce 여부는
    - 현재 상태에 속한 LR(0) item을 보면 알 수 있기 때문
  - Parsing stack의 내용은
    - symbol 뿐만 아니라 state 번호도 함께 저장
      - (symbol + state번호) 쌍으로 stack에 push
        - » *shift* action 일 때 token 저장
        - » *reduce* action 일 때 nonterminal 저장

- **LR(0) parsing 알고리즘은 DFA의 현재 상태만을 보고 action (*shift* 또는 *reduce*)을 결정**
  - 현재 상태는 항상 stack의 top에 위치

The LR(0) Parsing Algorithm (2/3)

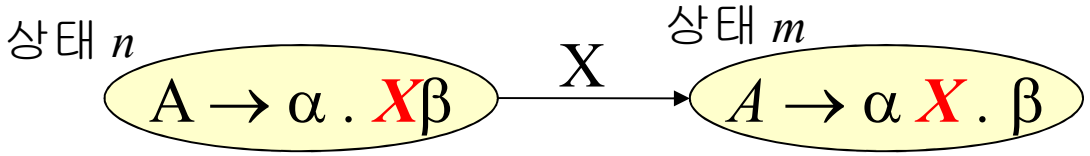
현재 상태 *n* (parsing stack의 top에 위치)이

1. $A \rightarrow \alpha\ .X\beta$ 형태의 item을 갖고 있는 경우

- **X**가 *terminal* 이면

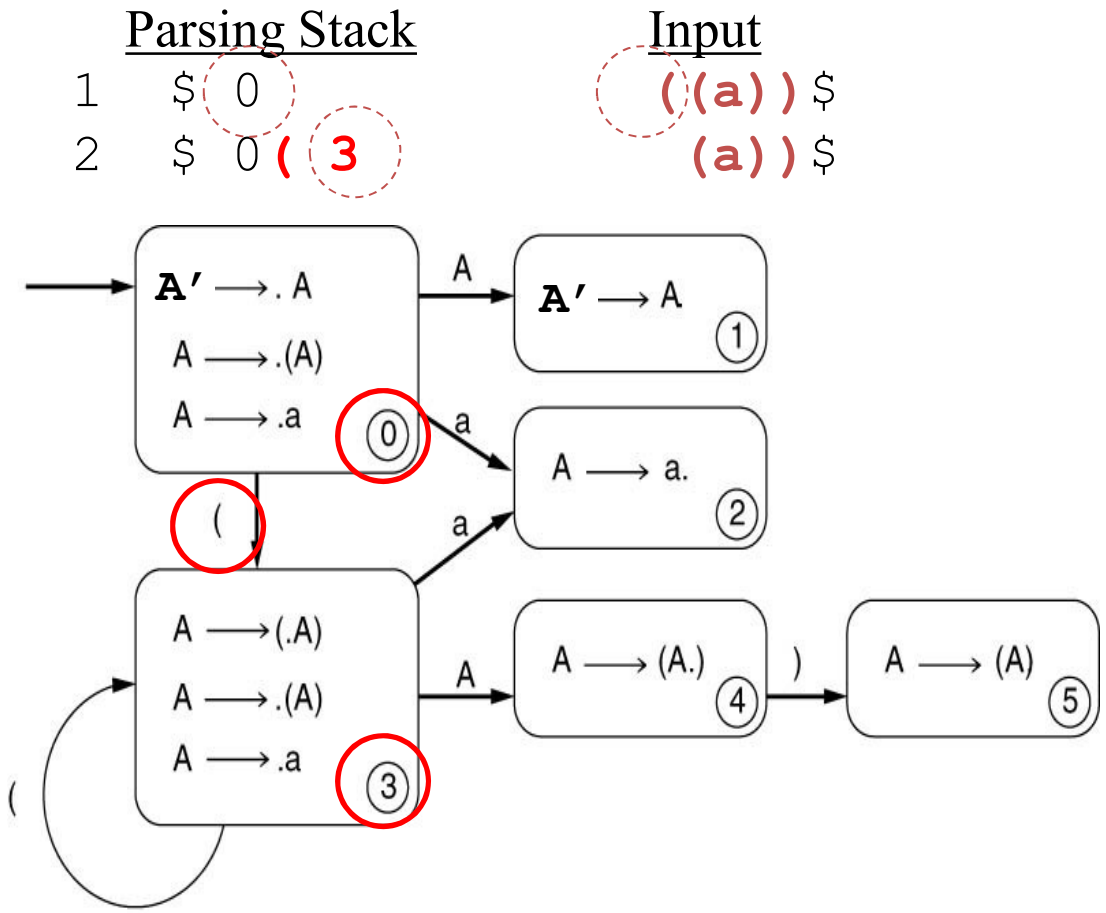
  ***shift* action** : *input lookahead* 를 stack에 push

- 다음 상태는 $A \rightarrow \alpha X\ .\ \beta$ 의 item을 포함



Parsing Stack	Input
$ . . . n	X . . . $
$ . . . n X m	. . . $

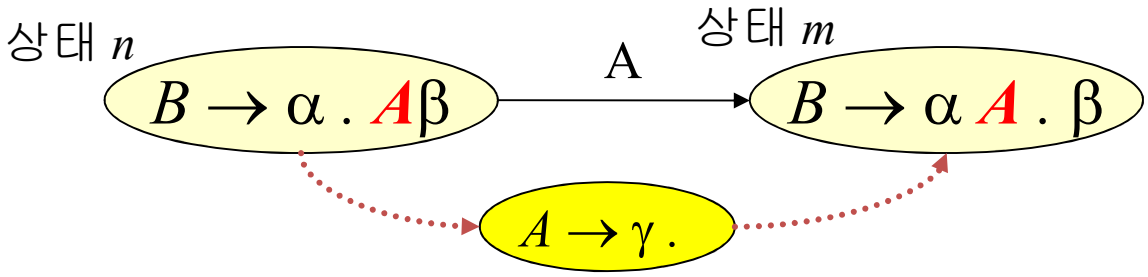
Stepwise Execution of the LR(0) algorithm



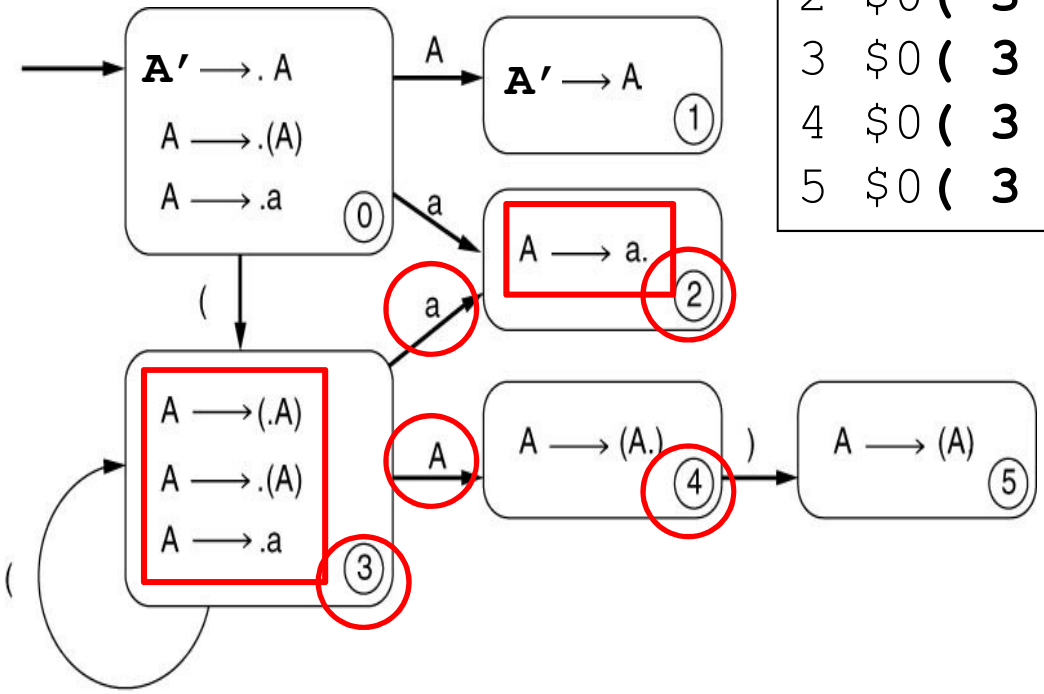
The LR(0) Parsing Algorithm (3/3)

현재 상태 *n* 이
2. $A \rightarrow \gamma$. 형태의 *complete* item을 갖고 있는 경우

- ***reduce* action**: reduction by $A \rightarrow \gamma$
  - Stack에서 기호(γ) 및 상태 번호를 함께 제거(*pop*) 하고 nonterminal *A*를 stack에 push
  - $B \rightarrow \alpha\ A\ .\ \beta$ ($B \rightarrow \alpha.A\beta$ 에서 reduction 직후의 item) 를 포함하는 상태 번호를 push
- $S' \rightarrow S.$ 이고, 입력 버퍼가 비어 있으면 → accept



Stepwise Execution of the LR(0) algorithm



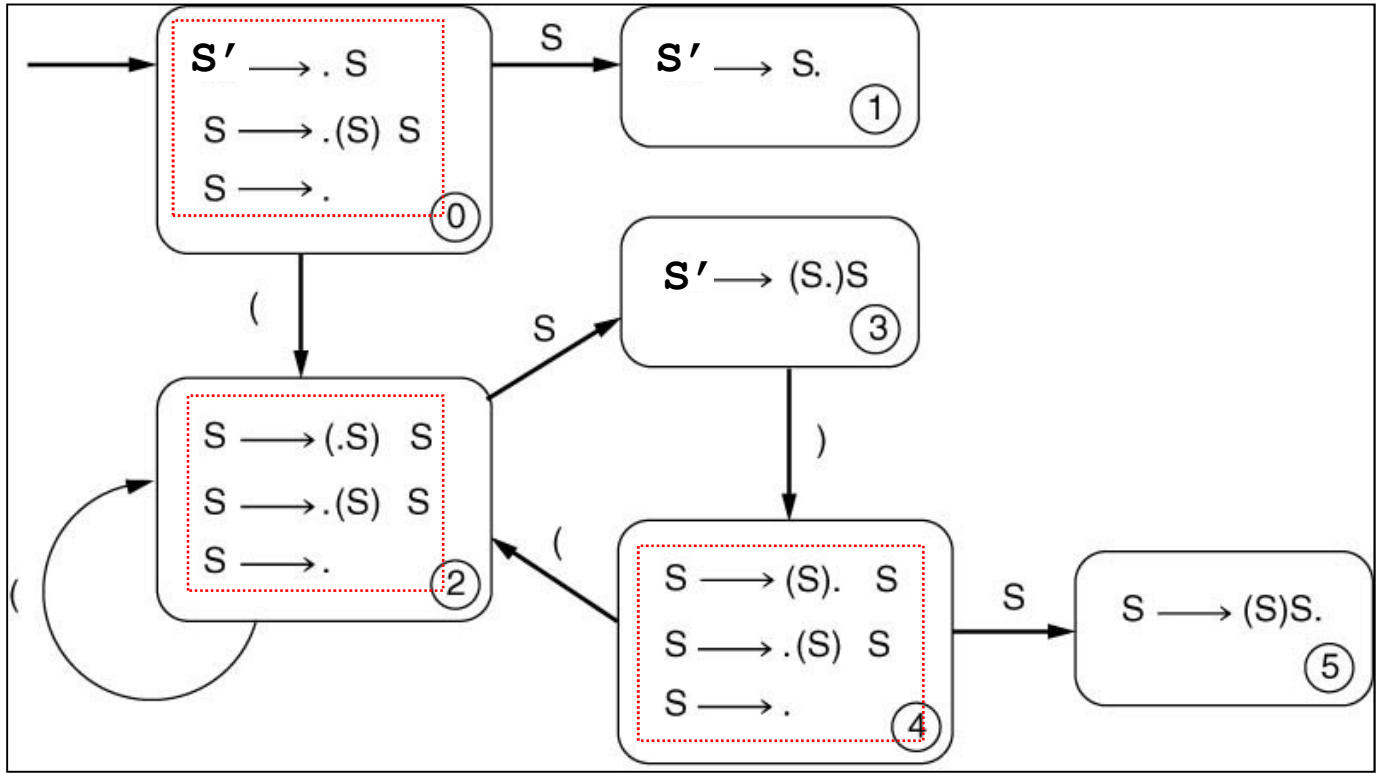
	Parsing Stack		Input
1	$0		((a))$
2	$0 (3		(a))$
3	$0 (3 (3		a))$
4	$0 (3 (3 a 2		))$
5	$0 (3 (3 A 4		))$

LR(0) Grammar

- **A grammar is *LR(0) grammar* *if and only if***
  - Each state is
    - a *shift* state (a state containing only "shift" items) or
    - a *reduce* state containing a *single* complete item
  - *shift* 면 *shift*, *reduce* 면 *reduce*, 하나로 통일된 item만을 갖고 있어야 함

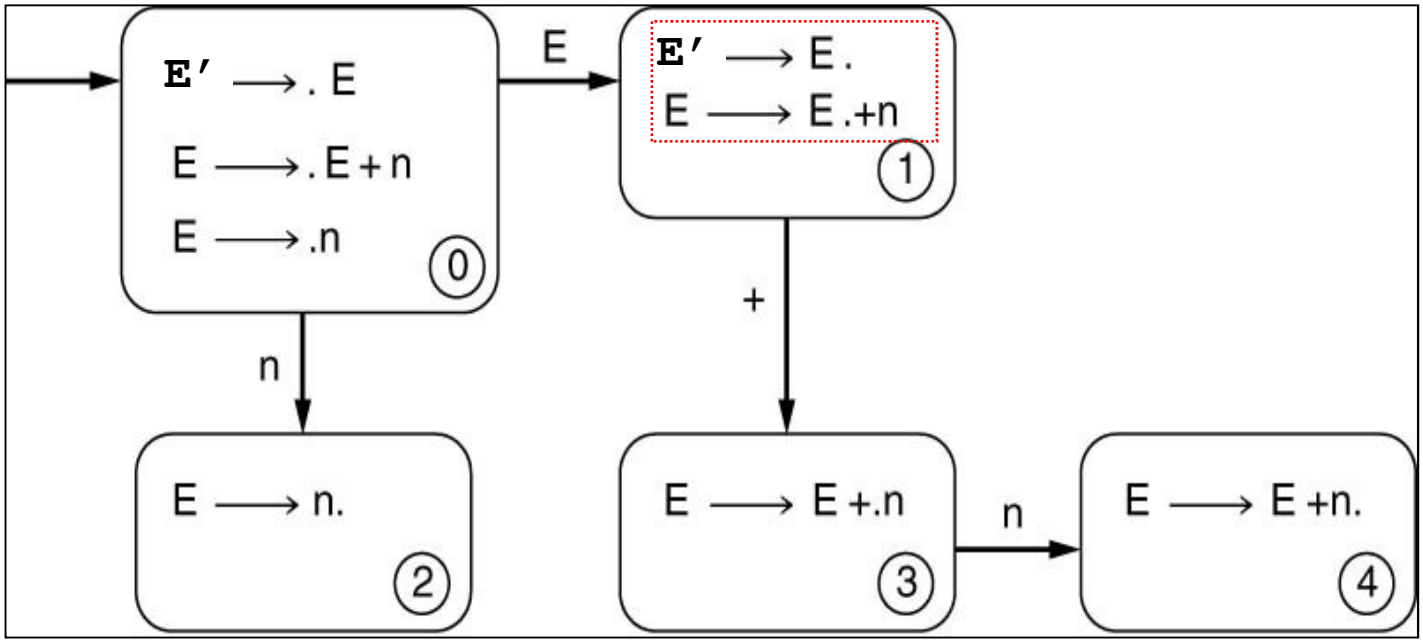
- **Otherwise, an ambiguity arises**
  - ***shift-reduce* conflict**
    - a state contains both $A \rightarrow \alpha \,.$ and $A \rightarrow \alpha \,. X\beta$
      - cannot decide whether *shift* or *reduce*
  - ***reduce-reduce* conflict**
    - a state contains both $A \rightarrow \alpha \,.$ and $B \rightarrow \beta \,.$
      - cannot decide *which production to reduce*

The following grammar is *not* LR(0)



shift-reduce conflicts

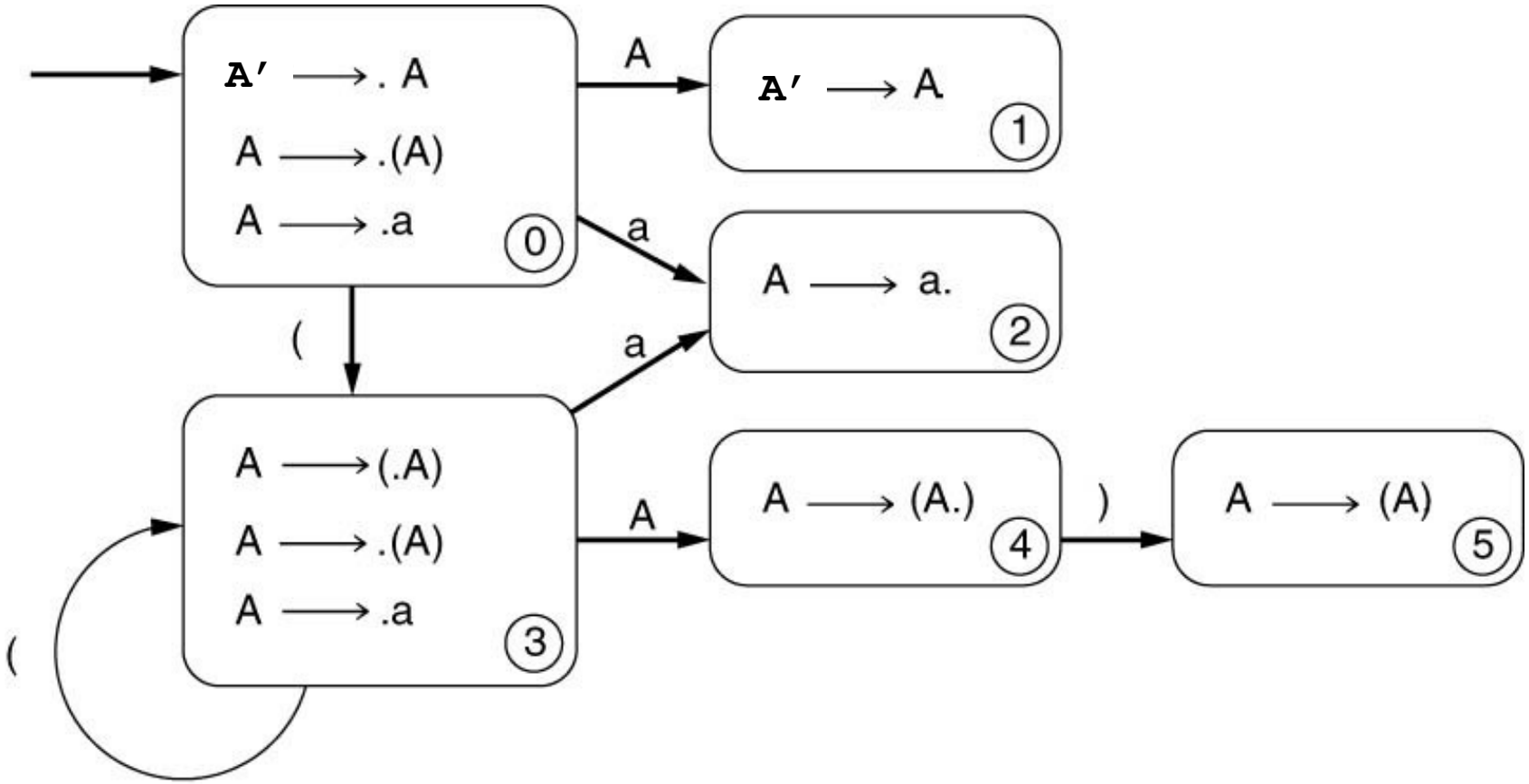
The following grammar is *not* LR(0)



shift-reduce conflict

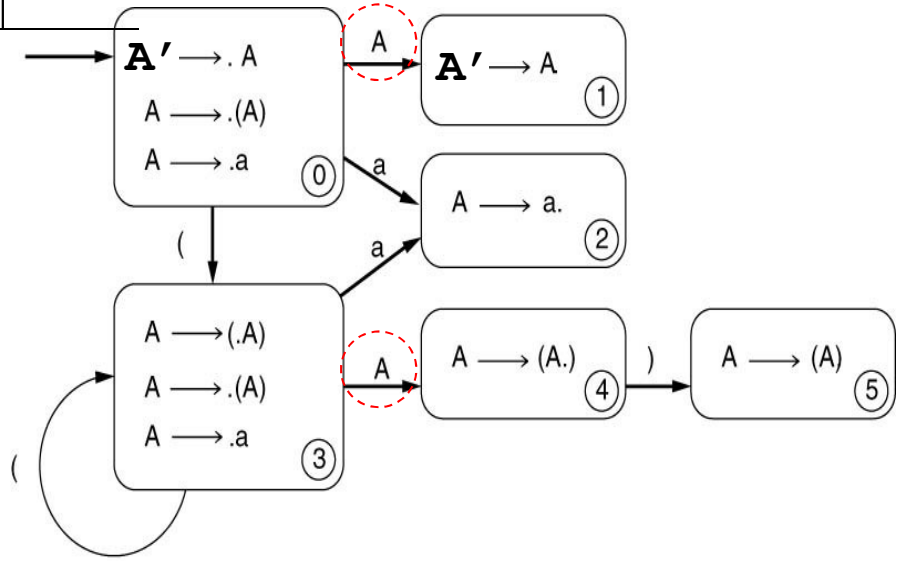
예 11: This is *LR(0) Grammar*

A → (A) | a

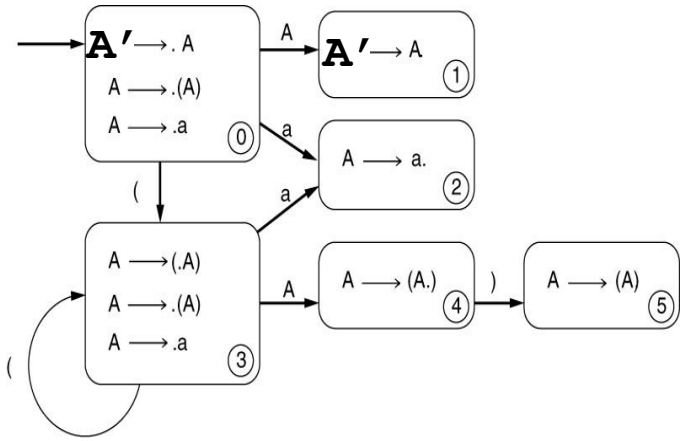


예 11: Parsing Table

State	Action	Rule	Input			Goto
			(	a	)	A
0	shift		3	2		1
1	reduce	A′→A				
2	reduce	A →a				
3	shift		3	2		4
4	shift				5	
5	reduce	A→(A)				



예 11: Parsing Actions



State	Action	Rule	Input			Goto
			(	a	)	A
0	shift		3	2		1
1	reduce	A′→A				
2	reduce	A →a				
3	shift		3	2		4
4	shift				5	
5	reduce	A→(A)				

	Parsing Stack	Input	Action
1	$0	**((a))**$	shift
2	$0**(**3	**(a))**$	shift
3	$0**(**3**(**3	**a))**$	shift
4	$0**(**3**(**3**a**2	**))**$	reduce A→a
5	$0**(**3**(**3A4	**))**$	shift
6	$0**(**3**(**3A4**)**5	**)**$	reduce A→(A)
7	$0**(**3A4	**)**$	shift
8	$0**(**3A4**)**5	$	reduce A→(A)
9	$0A1	$	accept