Compiler (컴파일러)
Syntax Analysis-Bottom Up

2015년 2학기

충남대학교 컴퓨터공학과
조은선

Bottom-Up Parsing

- Bottom-up = 우파스 = 우측유도의 역순
  - terminal 심벌들로부터 시작해서 시작 심벌을 유도
  - 생성규칙의 rhs 와 매치되나 보고 맞으면 lhs 로 바꿔치기 반복

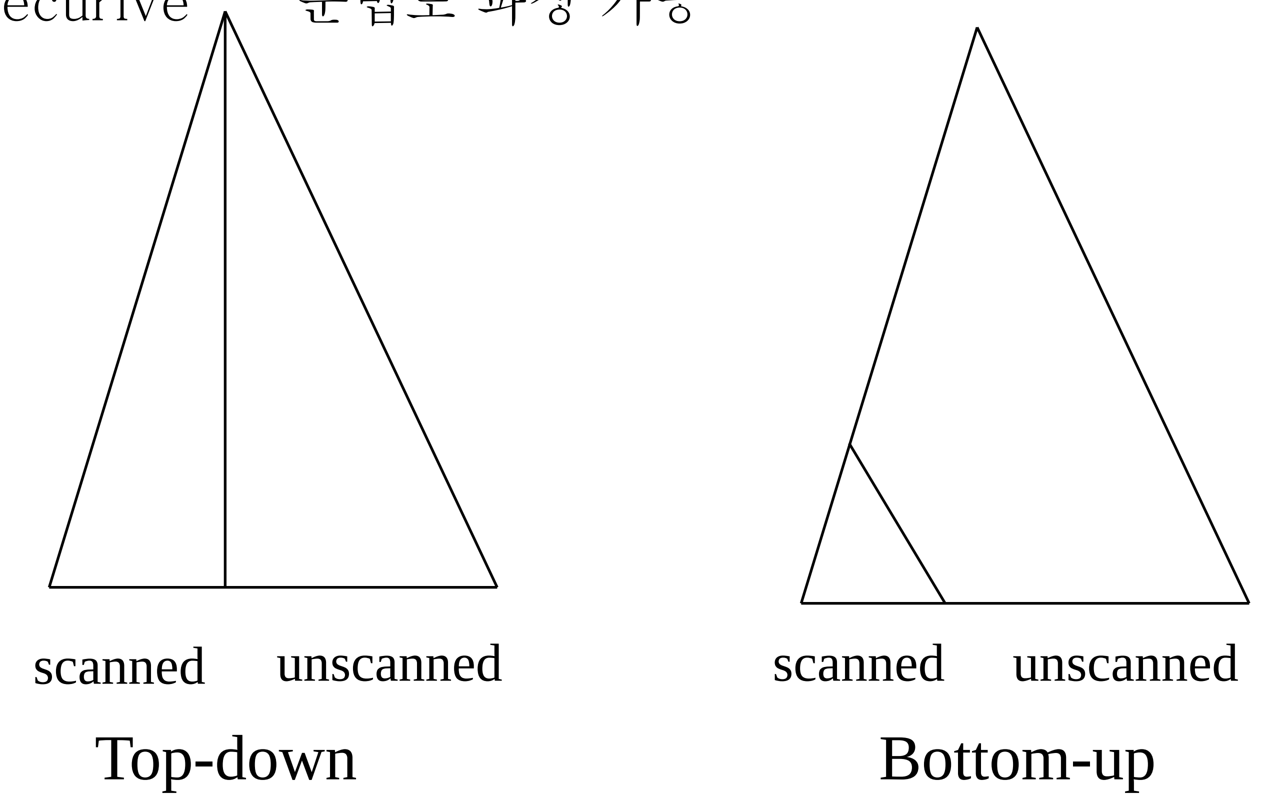
$$E \rightarrow E + T \mid T$$
$$T \rightarrow num \mid (E)$$

(1+2+(3+4))+5 ← (T+2+(3+4))+5
← (E+2+(3+4))+5 ← (E+T+(3+4))+5
← (E+(3+4))+5 ← (E+(T+4))+5 ← (E+(E+4))+5
← (E+(E+T))+5 ← (E+(E))+5 ← (E+T)+5 ← (E)+5
← E+5 ← E+T ← E

Top-down vs. Bottom-up

Bottom-up 이 더 강력

- 생성규칙 선택을 더 많은 token 이 들어올 때 까지 미룰 수 있음
- left-recurive 문법도 파싱 가능



용어 : LL, LR

- LL(k)
  - 입력을 Left-to-right 스캔
  - Left-most 유도
  - k 개의 심벌을 lookahead
  - [Top-down 또는 predictive] parsing 또는 LL parser
  - 파스트리를 pre-order 로 순회 및 생성

- LR(k)
  - 입력을 Left-to-right 스캔
  - Right-most 유도
  - k 심벌을 lookahead
  - [Bottom-up 또는 shift-reduce] parsing 또는 LR parser
  - 파스트리를 post-order 로 순회 및 생성

Reduce 란?

- S => αβω이고 A→ β의 생성규칙이 존재할 때, 문장형태 αβω에서 β를 A로 대치하는 것
- 파싱결과는 시작심벌이 나올때까지 reduce하면 얻어짐
- 예제

1. S → aAcBe 2. A → Ab

3. A → b 4. B → d

(1) reduce sequence

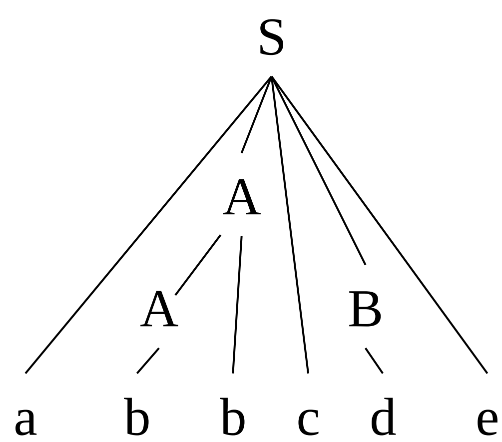
abbcde ⇒ aAbcde (reduce 3)

⇒ aAcde (reduce 2)

⇒ aAcBe (reduce 4)

⇒ S (reduce 1)

(2) 파스트리 ====>



Handle

- 한 문장 형태에서 reduce될 부분
  - S => αAω => αβω의 과정이 있을 때 β을 문장형태 αβω 의 handle이라 함

예제 1. E → E + T 2. E → T 3. T → T * F
4. T → F 5. F → (E) 6. F → a

유도 (파란글씨는 lhs)		
E	⇒ E + T	(1)
	⇒ E + T * F	(13)
	⇒ E + T * a	(136)
	⇒ E + F * a	(1364)
	⇒ E + a * a	(13646)
	⇒ T + a * a	(136462)
	⇒ F + a * a	(1364624)
	⇒ a + a * a	(13646246)

Reduce (밑줄이 handle)		
a + a * a	⇒ F + a * a	(6)
	⇒ T + a * a	(64)
	⇒ E + a * a	(642)
	⇒ E + F * a	(6426)
	⇒ E + T * a	(64264)
	⇒ E + T * F	(642646)
	⇒ E + T	(6426463)
	⇒ E	(64264631)

Handle

- 같은 문장 형태에서 서로 다른 두개 이상의 handle 이 존재할 때? ..... “모호하다”
- 예제) $E \rightarrow E + E \mid E * E \mid (E) \mid id$ 에서

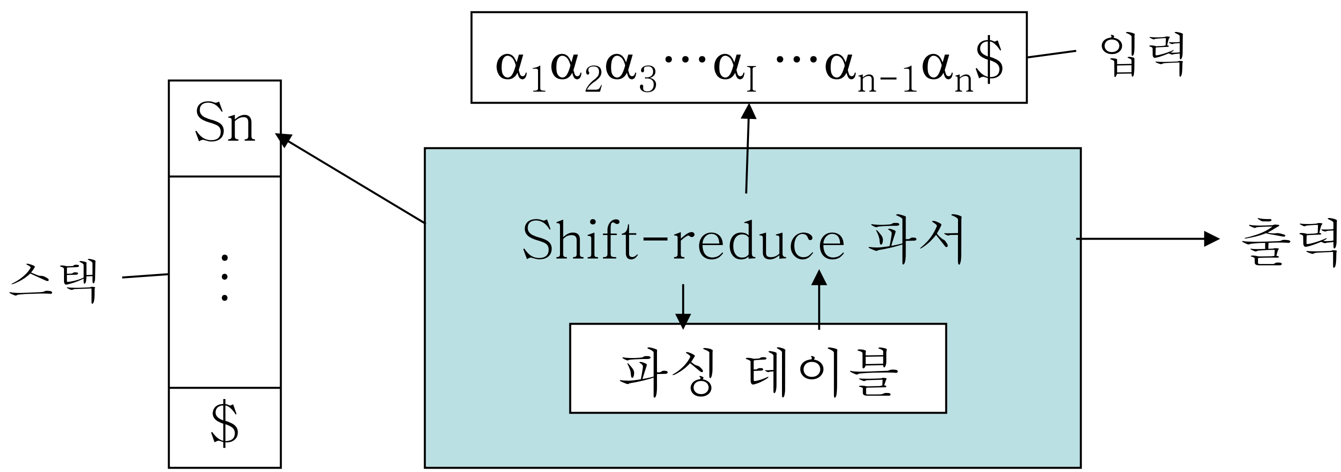
  $id + id * id$

  <아래 우측 유도식을 거꾸로 올라가며 생각하면..>

$E \Rightarrow \underline{E + E}$	$E \Rightarrow \underline{E * E}$
$\Rightarrow E + \underline{E * E}$	$\Rightarrow \underline{E} * id$
$\Rightarrow E + E * \underline{id}$	$\Rightarrow \underline{E + E} * id$
$\Rightarrow E + \underline{id} * id$	$\Rightarrow E + \underline{id} * id$
$\Rightarrow \underline{id} + id * id$	$\Rightarrow \underline{id} + id * id$

Shift-Reduce Parsing

- Overview
  - Shift : 스택의 top에 handle이 나타날 때까지 계속해서 입력심벌을 스택에 옮겨 담기
  - Reduce : handle를 찾으면 생성 규칙을 결정하여 lhs 인 non-terminal 로 바꿔 치기
  - 이것을 문법의 시작 심벌에 이를 때까지 반복, 수행



Actions

- Shift: look-ahead token 하나를 스택에 옮김
  - push a

stack	input	action
(	1+2+(3+4))+5	shift 1
(1	+2+(3+4))+5	

- Reduce: 스택 top에 있는 handle β을 non-terminal 심벌 X로 바꿔치기 (단, 생성규칙은 X→ β)
  - pop β, push X

stack	input	action
(E+T	+(3+4))+5	reduce E → E+ T
(E	+(3+4))+5	

예) Shift-Reduce Parsing

1. $E \rightarrow E + T$ 2. $E \rightarrow T$ 3. $T \rightarrow T * F$
4. $T \rightarrow F$ 5. $F \rightarrow (E)$ 6. $F \rightarrow a$

(1) reduce 과정 :

$$
\begin{array}{lll}
\underline{a} + a * a & \Rightarrow \underline{F} + a * a & (6) \\
 & \Rightarrow \underline{T} + a * a & (64) \\
 & \Rightarrow E + \underline{a} * a & (642) \\
 & \Rightarrow E + \underline{F} * a & (6426) \\
 & \Rightarrow E + T * \underline{a} & (64264) \\
 & \Rightarrow E + \underline{T * F} & (642646) \\
 & \Rightarrow \underline{E + T} & (6426463) \\
 & \Rightarrow E & (64264631)
\end{array}
$$

∴ reduce sequence : 64264631

(2) 구문 분석 과정

스　　택	입력 버퍼	구문분석행동
$	a+a*a$	shift a
$a	+a*a$	reduce F → a
$F	+a*a$	reduce T → F
$T	+a*a$	reduce E → T
$E	+a*a$	shift +
$E+	a*a$	shift a
$E+a	*a$	reduce F → a
$E+F	*a$	reduce T → F
$E+T	*a$	shift *
$E+T*	a$	shift a
$E+T*a	$	reduce F → a
$E+T*F	$	reduce T → T*F
$E+T	$	reduce E → E+T
$E	$	accept

- 초기상태 : 스택에 $, 입력 버퍼에는 입력스트링과 $를 넣음
- accept 상태 : 스택 top에 시작 심벌이 있고 입력은 모두 본 상태

Issues

- Shift 할지 Reduce 할지 어떻게 아는가?
  - 스택에 β 가 있고, 입력 글자가 b일때
    - $\beta = \alpha\gamma$ 이고, $X \rightarrow \gamma$ 가 있다면, 스택을 αX 로 만들것인가?
    - 아니면, b를 스택에 옮겨 βb 로 만들것인가?
- 여러 가지 reduce 방법이 있으면 어떻게 할 것인가?
  - 스택 top중 얼만큼을 떼어 handle 로 볼 것인가?
    - 스택이 $\beta = \alpha\gamma$ 일때 $X \rightarrow \gamma$ 규칙에 의해 reduce 를 한다면, 가능한 γ 의 길이가 여러가지 아닌가?

해결 – LR Parsing Engine

- ‘파서 상태’ 라는 것들을 사용
  - 스택에 입력 심벌 shift 할 때, 관련 ‘파서 상태’도 끼워 넣음

    예) 0 (6 E 10 + 5 ... *파란 글씨는 상태 번호*
  - 이 파서 상태가 무엇인지에 따라서, reduce인지, shift 인지, 얼만큼을 stack top으로 볼것인지 등 결정
    - 애매한 일 없음
- 파싱테이블도 <상태> X <심벌들> 형태
  - 파서는 파싱테이블을 보고
    - 현재 상태, 입력 심벌을 보고 shift 할지 reduce 할지 결정하고 (action table)
    - 파서 상태도 변경시킴 (goto table)

LR Parsing Table

	Terminals	Non-terminals
파서상태	다음 action + 다음 상태	다음 상태
	Action table	Goto table

- 현재 상태 S와 입력 문자 a를 보고
  - M[S,a] = “shift S’ ” 라면 … shift:
    - push(a), push(S’)
  - M[S,a] = “reduce X→ α ”라면… reduce:
    - pop(2*|α|), S’= top(), push(X), push(M[S’,X])

예)

L→L, S
L→S
S→a

- 파싱테이블

"0은 시작상태"

심벌 / 상태	a	,	$	L	S
0	s3			g1	g2
1		s4	accept		
2		L→S	L→S		
3		S→a	S→a		
4	s3				g5
5		L→L,S	L→L,S		

- 파싱 과정: a,a (**좀 짧은버전**)

	$0	a, a$
s3 →	$0 a 3	, a$
S→a g2 →	$0 S 2	, a$
L→S g1 →	$0 L 1	, a$
s4 →	$0 L 1 , 4	a$
s3 →	$0 L 1 , 4 a 3	$
S→a g5 →	$0 L 1 , 4 S 5	$
L→L,S g1 →	$0 L 1	$
accept →	$accept	$

예-다시)

L→L,S
L→S
S→a

- 파싱테이블

심벌 / 상태	a	,	$	L	S
0	s3			g1	g2
1		s4	accept		
2		L→S	L→S		
[illegible]		[illegible]	[illegible]		
[illegible]					g5
5		L→L,S	L→L,S		

그럼, 이런 table 을 만드는 방법은?

- 파싱 과정: a,a (**좀 풀어서**)

	$0	a, a$
shift 3 →	$0 a 3	, a$
reduce S→a →	$0 S	, a$
goto 2 →	$0 S 2	, a$
reduce L→S →	$0 L	, a$

이하, 각자 채워보시오..

LR 파서

- LR(k)
  - Left-to-right 스캔, 우측 유도, k개의 lookahead 문자
  - 기본적으로 : LR(0), LR(1)
  - 사용을 위한 변형 : SLR,... LALR(1)
- LR(0) 파서부터 하자.
  - lookahead 없이 파싱하고, shift-reduce 형태
- LR(0) 파싱 테이블 만들기
  - 파서 상태가 될만한 것들이 무엇인지 정하고
    - LR(0) 아이템, Closure 등을 이용
  - 각 파서 상태들 간의 상태 전이도 (DFA: Deterministic Finite Automata) 를 만들어서
    - GOTO 를 이용
  - 이것을 parsing table 에 담아내야함

LR(0) 아이템

- LR(0) 아이템
  - rhs에 점('.') 심벌을 가진 생성 규칙
  - 생성 규칙의 형태가 A→XYZ일 때,
    - [A→.XYZ], [A→X.YZ], [A→XY.Z], [A→XYZ.] 등은 모두 LR(0)아이템
  - 생성 규칙이 A → ε 일때,
    - [A→.] 는 LR(0) 아이템

예) E → num | (S) 에서 등장할 수 있는 LR(0) Item?

[E→.num] [E→num.]

[E→.(S)] [E→(.S)] [E→(S.)] [E→(S).]

closure

- closure란?

$$\text{closure}(I) = I \cup \{[B \rightarrow .\gamma\ ] \mid [A \rightarrow \alpha.B\beta\ ] \in \text{closure}(I),\ B \rightarrow \gamma \in P\}$$

  - t 가 terminal 심벌일때

    $\text{closure}([A \rightarrow \alpha.t\beta]) = \{[A \rightarrow \alpha.t\beta]\}$

  - X가 non-terminal 심벌일 때,

    $\text{closure}([A \rightarrow \alpha.X\beta]) = \{[A \rightarrow \alpha.X\beta\ ],\ [X \rightarrow .\gamma\ ]\}$

    for all production $X \rightarrow \gamma$, recursively.

closure 예

(1)

S' → G
G → E = E | f
E → E + T | T
T → T * f | f

- closure({[S' →.G]}) = { [S' →.G], [G →.E=E],[G →.f], [E →.E+ T],[E →.T], [T →.T*f], [T →.f] }
- closure({[E →E.+ T]}) = {[E →E.+ T]}

(2)

S'→S
S → (L) | id
L → S | L,S

- closure({[S' → . S]}) = {[S' → . S], [S → . (L)], [S → . id]}

goto

- goto(I,X)

  $= closure(\{[A \rightarrow \alpha X.\beta] \mid [A \rightarrow \alpha .X\beta] \in I\})$

- 예) 앞의 예 (1)의 문법에서
  - I = {[G →E.=E], [E →E.+ T]} 일 때,

    goto(I, +) = closure({[E →E+ .T]})

    = {[E →E+ .T],[T →.T*f], [T →.f]}

  - I = {[E →.T], [T →.T*f], [T →.f]} 일 때,

    goto(I,T) = closure({[E →T.], [T →T.*f]})

    = {[E →T.], [T →T.*f]}

Note: A→ ε 에 대한 goto 는 없다.

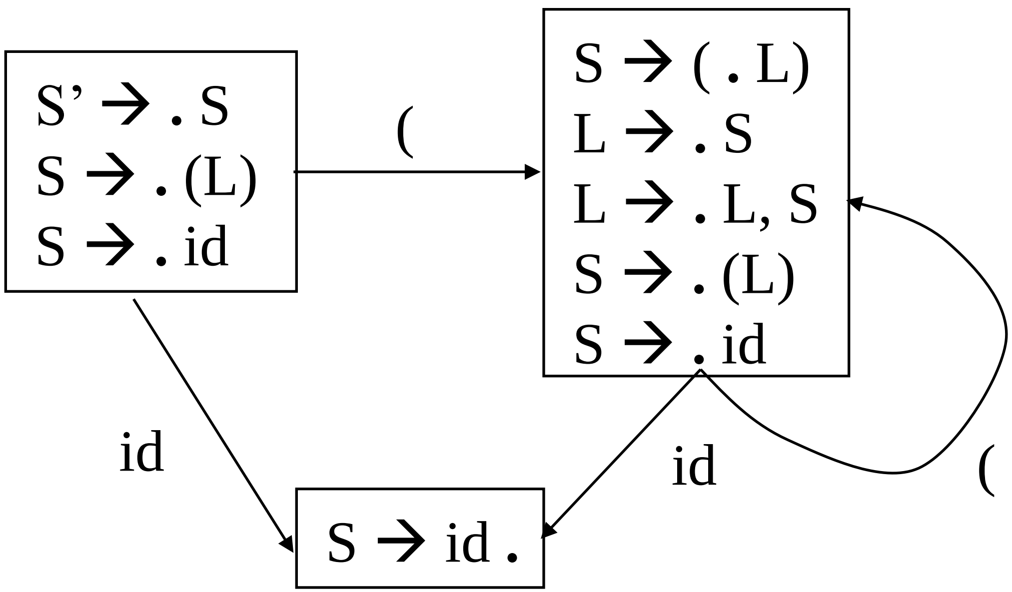
Class Problem

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

1. $I = \{\ [E' \rightarrow .\ E]\}$ 일때, closure(I) 는?

2. $I = \{\ [E' \rightarrow E\ .\ ],\ [E \rightarrow E\ .\ +\ \ T]\ \}$ 일때, goto(I,+) 는?

goto의 예: Terminal Symbols



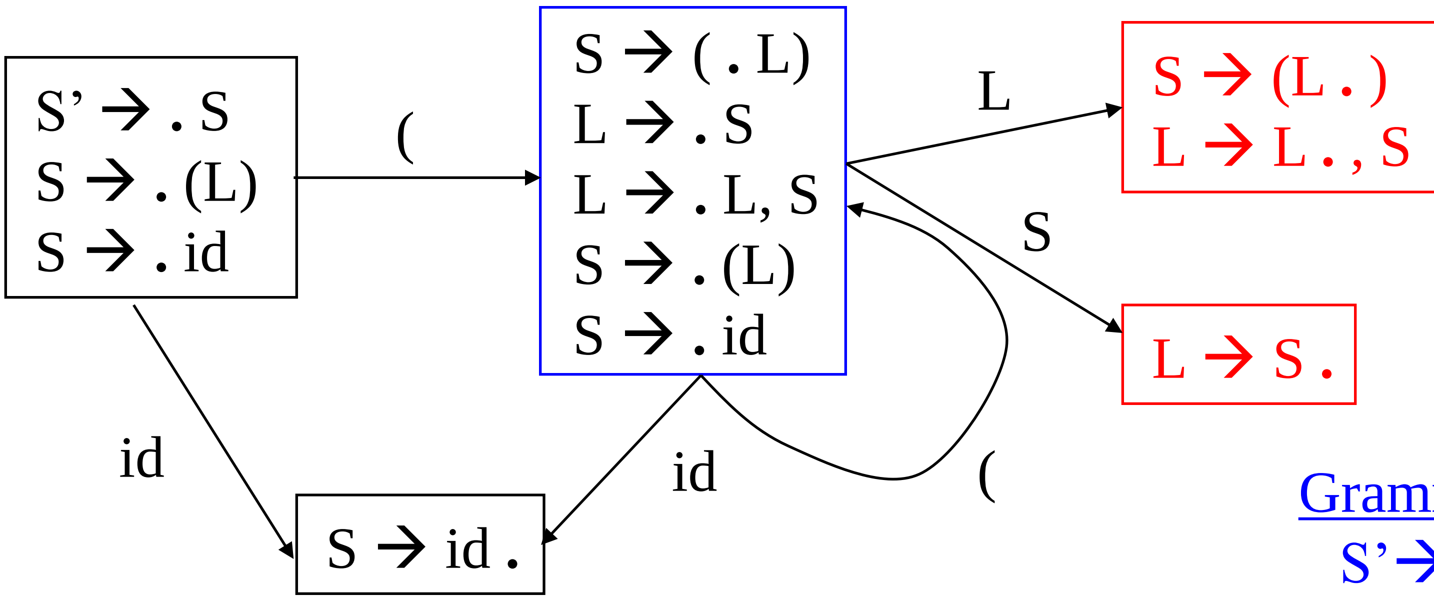
Grammar

S' → S

S → (L) | id

L → S | L,S

goto 의 예: Non-terminal Symbols



Grammar

S' → S

S → (L) | id

L → S | L,S

의미

- [X → α . β]
  - "α 만큼은 이미 매치됨"
  - 앞으로 β가 올 가능성 있음. (β 는 '마크심벌' 이라함)
  - 특별히 [A→X.]를 'reduce item' 이라 한다
- closure([A→ α.Xβ])
  - "α 까지 parse됐고, X를 parse 할 것으로 기대하는 상태"
- 파서 상태 만들기
  - 먼저, 생성 규칙 'S' → S' 을 추가 ('Augmented Grammar')
  - 시작 상태 : [S' → . S] 의 <u>closure</u>
  - 그 다음 상태들 : 여기서부터 <u>goto</u> 를 계산해서 만들어 냄(C_0)

C_0

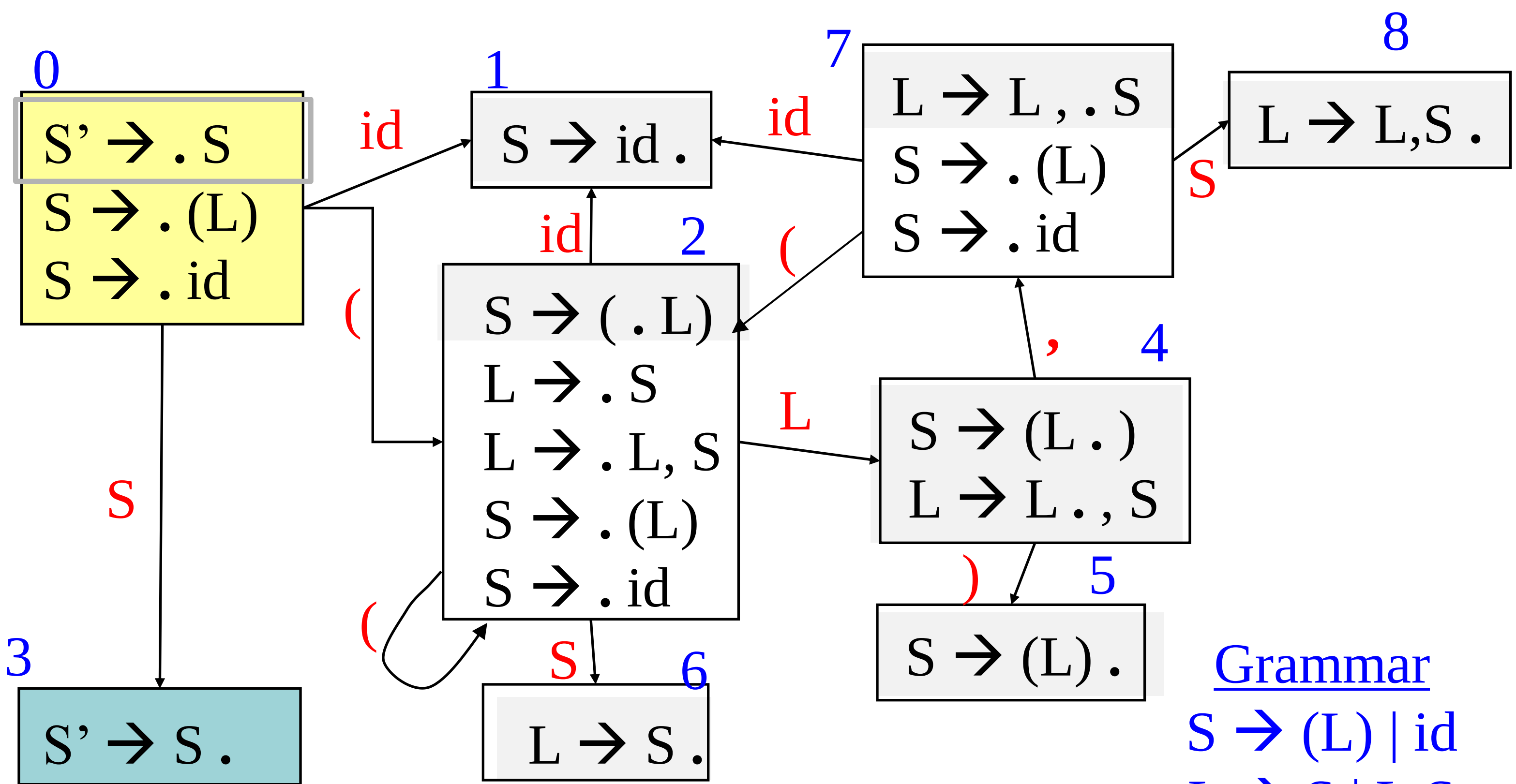
- $C_0 = \{closure(\{[S' \rightarrow .S]\})\} \cup \{goto(I,X) \mid I \in C_0, X \in V\}$
  - 추가된 생성규칙 $S' \rightarrow S$ 에서 부터 차례로 closure와 goto를 적용하여 얻은 모든 타당한 LR(0) 아이템 집합들
- $C_0 = \{I_0, I_1, \ldots, I_n\}$라면
  - I_0 는 $\{S' \rightarrow S\}$ 의 closure, 다른 것들은 여기서부터 각 마크심벌에 따라 goto를 적용하여 만든 상태
- Item 의 분류

  $[A \rightarrow X.Y]$: $X \neq \varepsilon$ 일때 kernel item

  $[A \rightarrow .X]$: closure item

  $[A \rightarrow X.]$: reduce item

예) 앞의 예에서 C_0와 Transition들



Grammar
S → (L) | id
L → S | L,S
and augmented by S'→S

Class Problem

다음 문법에서 C0와 LR(0) transition 을 구하시오.
(힌트: 상태개수는 시작상태 포함 6개)

$S' \rightarrow S$
$S \rightarrow E + S$
$S \rightarrow E$
$E \rightarrow num$

Parsing Table 구성하기

- LR(0) 파싱테이블
  - 파서 상태로 $C_0 = \{ I_0, I_1, \ldots, I_n \}$ 과 transition 이용
  - $goto(I_1,a)= I_2$ 면: $M[I_1,a]$ = 'shift I_2'
  - $goto(I_1,N)= I_2$ 면: $M[I_1,N]$ = 'goto I_2'
  - I_1 이 $[X \rightarrow \beta.]$ 를 포함한다면: $M[I_1,*]$ = 'reduce $X \rightarrow \beta$'
  - I_1 이 $[S' \rightarrow S.]$를 포함한다면: $M[I_1, \$]$:= 'accept'
- SLR (Simple LR) 파싱테이블
  - 약간의 노력으로 LR(0) 보다 정교하게, conflict도 줄임
  - I_1 이 $[X \rightarrow \beta.]$ 를 포함할 때:
    - $M[I_1,a]$ = 'reduce $X \rightarrow \beta$', 단 $a \in FOLLOW(X)$
  - 나머지는 LR(0)와 똑같음

LR(0) 파싱 테이블

Grammar

S → (L) | id

L → S | L,S

Input terminal — Non-terminals

State	(	)	id	,	$	S	L
0	s2		s1			g3	
1	S→id	S→id	S→id	S→id	S→id		
2	s2		s1			g6	g4
3					accept		
4		s5		s7			
5	S→(L)	S→(L)	S→(L)	S→(L)	S→(L)		
6	L→S	L→S	L→S	L→S	L→S		
7	s2		s1			g8	
8	L→L,S	L→L,S	L→L,S	L→L,S	L→L,S		

blue = shift　　red = reduce

SLR 파싱 테이블

Grammar
S → (L) | id
L → S | L,S

Input terminal / Non-terminals

State	(	)	id	,	$	S	L
0	s2		s1			g3	
1		S→id		S→id	S→id		
2	s2		s1			g6	g4
3					accept		
4		s5		s7			
5		S→(L)		S→(L)	S→(L)		
6		L→S		L→S			
7	s2		s1			g8	
8		L→L,S		L→L,S			

FOLLOW(S) = {$,), ,}
FOLLOW(L) = {), ,}

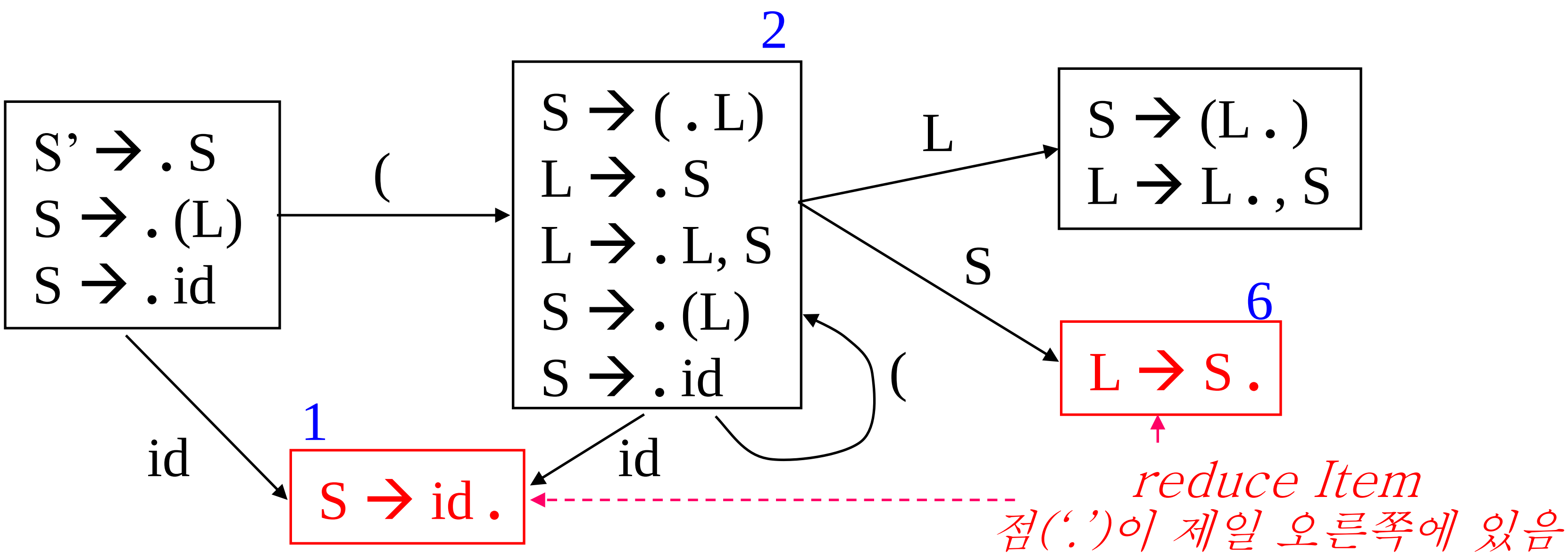
Class Problem

다음 문법에서 LR(0) 파싱테이블과 SLR 파싱 테이블을 각 각 구하시오

S’ → S
S → E + S
S →E
E → num

Reduce 하기

Grammar
S → (L) | id
L → S | L,S



- rhs 만큼 stack 에서 pop 하고, lhs 를 대신 push
  - 즉, X → β. 라면 β 를 pop 하고 X 를 push
  - table (!) 에서 다음 상태도 읽어 함께 push

derivation	stack	input	action
((a),b) ←	0 (2 (2	a),b)	shift, goto 1
((a),b) ←	0 (2 (2 a 1	),b)	reduce S → id
((S),b) ←	0 (2 (2 S 6	),b)	reduce L → S

Parsing Example : ((a),b)

$S \rightarrow (L) \mid id$
$L \rightarrow S \mid L,S$

derivation	stack	input	action
((a),b) ←	0	((a),b)$	shift, goto 2
((a),b) ←	0(2	(a),b)$	shift, goto 2
((a),b) ←	0(2(2	a),b)$	shift, goto 1
((a),b) ←	0(2(2a1	),b)$	reduce S→id
((S),b) ←	0(2(2S6	),b)$	reduce L→S
((L),b) ←	0(2(2L4	),b)$	shift, goto 5
((L),b) ←	0(2(2L4)5	,b)$	reduce S→(L)
(S,b) ←	0(2S6	,b)$	reduce L→S
(L,b) ←	0(2L4	,b)$	shift, goto 7
(L,b) ←	0(2L4,7	b)$	shift, goto 8
(L,b) ←	0(2L4,7b1	)$	reduce S→id
(L,S) ←	0(2L4,7S8	)$	reduce L→L,S
(L) ←	0(2L4	)$	shift, goto 5
(L) ←	0(2L4)5	$	reduce S→(L)
S ←	0S3	$	done

Why SLR?

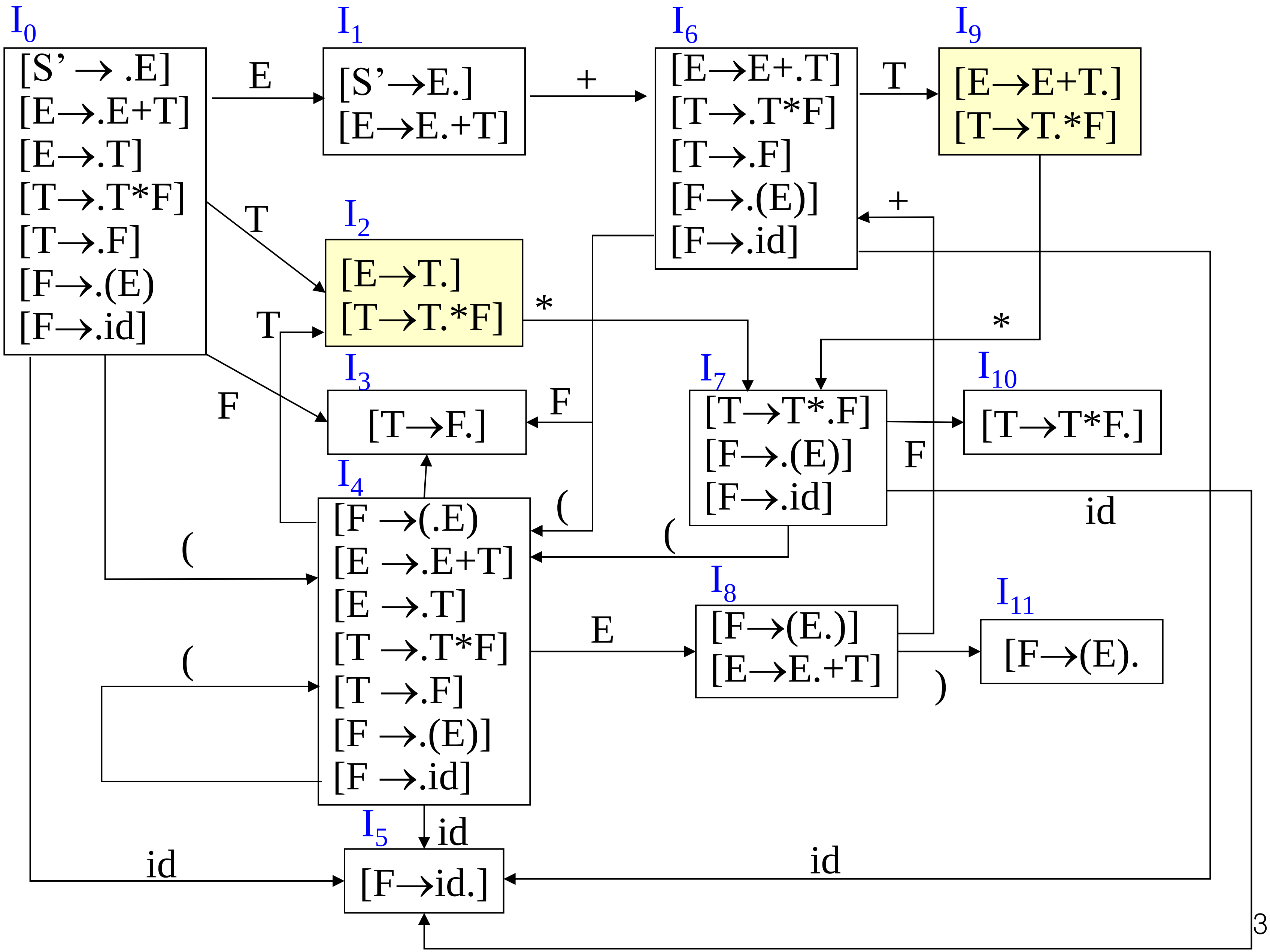
(1) 문법

0. S' → E (첨가됨)
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

(2) FOLLOW의 계산

FOLLOW(E) = {$, + ,)}
FOLLOW(T) = {*, + ,), $}
FOLLOW(F) = {*, + ,), $}

(3) C_0 계산



I0
[S’ → .E]
[E→.E+T]
[E→.T]
[T→.T*F]
[T→.F]
[F→.(E)]
[F→.id]
I1
[S’→E.]
[E→E.+T]
I6
[E→E+.T]
[T→.T*F]
[T→.F]
[F→.(E)]
[F→.id]
I9
[E→E+T.]
[T→T.*F]
I2
[E→T.]
[T→T.*F]
I3
[T→F.]
I7
[T→T*.F]
[F→.(E)]
[F→.id]
I10
[T→T*F.]
I4
[F →(.E)
[E →.E+T]
[E →.T]
[T →.T*F]
[T →.F]
[F →.(E)]
[F →.id]
I8
[F→(E.)]
[E→E.+T]
I11
[F→(E).
I5
[F→id.]
E
+
T
*
F
(
)
id

(4) SLR 파싱 테이블

상태	id	+	*	(	)	$	E	T	F
0	s5			s4			g1	g2	g3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			g8	g2	g3
5		r6	r6		r6	r6			
6	s5			s4				g9	g3
7	s5			s4					g10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

LR(0) 테이블은?

(5) 구문분석 과정

a*a+ a

스택	입력	구문 분석 내용	출력
0	a*a+a$	shift 5	
0a5	*a+a$	reduce 6 F→id	6
0F	*a+a$	goto 3	
0F3	*a+a$	reduce 4 T→F	4
0T	*a+a$	goto 2	
0T2	*a+a$	shift 7	
0T2*7	a+a$	shift 5	
0T2*7a5	+a$	reduce 6 F→id	6
0T2*7F	+a$	goto 10	
0T2*7F10	+a$	reduce 3 T→T*F	3
0T	+a$	goto 2	
0T2	+a$	reduce 2 E→T	2
0E	+a$	goto 1	
0E1	+a$	shift 6	
0E1+6	a$	shift 5	
0E1+6a5	$	reduce 6 F→id	6
0E1+6F	$	goto 3	
0E1+6F3	$	reduce 4 T→F	4
0E1+6T	$	goto 9	
0E1+6T9	$	reduce 1 E→E+T	1
0E	$	goto 1	
0E1	$	accept	

LR(1) Parsing

- LR(1)
  - 한개의 lookahead 를 통해 얻을 수 있는 것들을 모두 얻어 parsing tree를 구성
  - LR(1) 문법 : LR(1) parsing으로 parsing 되는 CFG
- LR(1) vs. LR(0) : "아이템의 차이"
  - 둘다 : parser 상태 = set of items
  - LR(1) item = LR(0) item + lookahead 심벌
    - LR(0) item: $S \rightarrow .\, S + E$
    - LR(1) item: $S \rightarrow .\, S + E$, +

LR(1) 아이템

$[S \rightarrow S \,.\, + E,\ +/\$]$
$[S \rightarrow S + .\, E,\ num]$

- LR(1) item = $[X \rightarrow \alpha \,.\, \beta\ ,\ y\,]$

  "스택에 α 까지 이미 match 되었고,
  그 다음으로 β y 가 들어올 수 있는 상태"
  - Lookahead 심벌 in LR(1) item
    - 해당 생성규칙으로 유도된 직후에 나타날 심벌
    - reduce 할 때만 쓰임
  - $[X \rightarrow \alpha \,.\, \beta\ ,\ \{x1, \ldots, xn\}]$ 또는
    $[X \rightarrow \alpha \,.\, \beta\ ,\ x1/ \ldots/xn]$ 는?
    - 의미 : $\{[X \rightarrow \alpha \,.\, \beta\ ,\ x1], \ldots [X \rightarrow \alpha \,.\, \beta\ ,\ xn]\}$
- closure, goto 등도 확장해야함

LR(1) closure, goto

$$\text{closure}(I) = I \cup \{[B \to .\gamma, \text{FIRST}(\beta z)] \mid [A \to \alpha.B\beta, z] \in \text{closure}(I),\ B \to \gamma \in P\}$$

$$\text{goto}(I,X) = \text{closure}(\{[A \to \alpha X.\beta, z]\} \mid [A \to \alpha.X\beta, z] \in I\}$$

- closure는 LR(0) closure와 비슷하나, lookahead를 기록하며 계산하는 것만 다름

  $$\text{LR(0) closure}(I) = I \cup \{[B \to .\gamma\ ] \mid [A \to \alpha.B\beta\ ] \in \text{closure}(I),\ B \to \gamma \in P\}$$

- goto는 똑같음.

예

$S' \rightarrow S$
$S \rightarrow E + S \mid E$
$E \rightarrow num$

- FIRST(S) = FIRST(E) = {num}
- 시작 상태는 [S' → . S , $]의 closure임
- closure({[S'→.S, $]})
  ={[S' → . S , $], [S → . E + S ,$], [S → . E , $],
  [E → . num , +],[E → . num , $]}
  ={[S' → . S , $], [S → . E + S ,$], [S → . E , $],
  [E → . num , +/$]}

- I1 = {[S → E . + S , $], [S → E . , $]} 일 때,
  goto (I1, '+') = closure({[S → E + . S , $]})
  = {[S→E+ .S, $],[S→.E+ S, $][S→.E, $]
  [E→.num, +/$]}

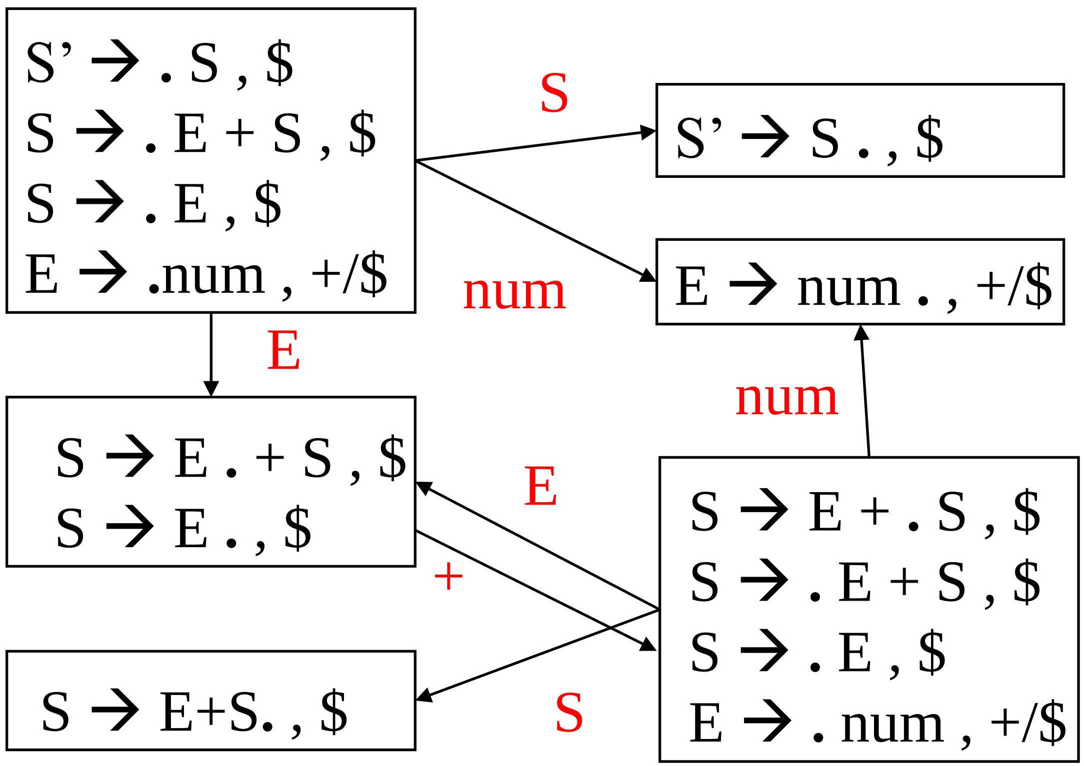
Class Problem

$I = \{[S \rightarrow E + . S , \$]\}$ 일 때,
다음을 구하여라.

1) I'=closure(I)
2) goto(I', num)
3) goto(I', E)

$S' \rightarrow S$
$S \rightarrow E + S \mid E$
$E \rightarrow num$

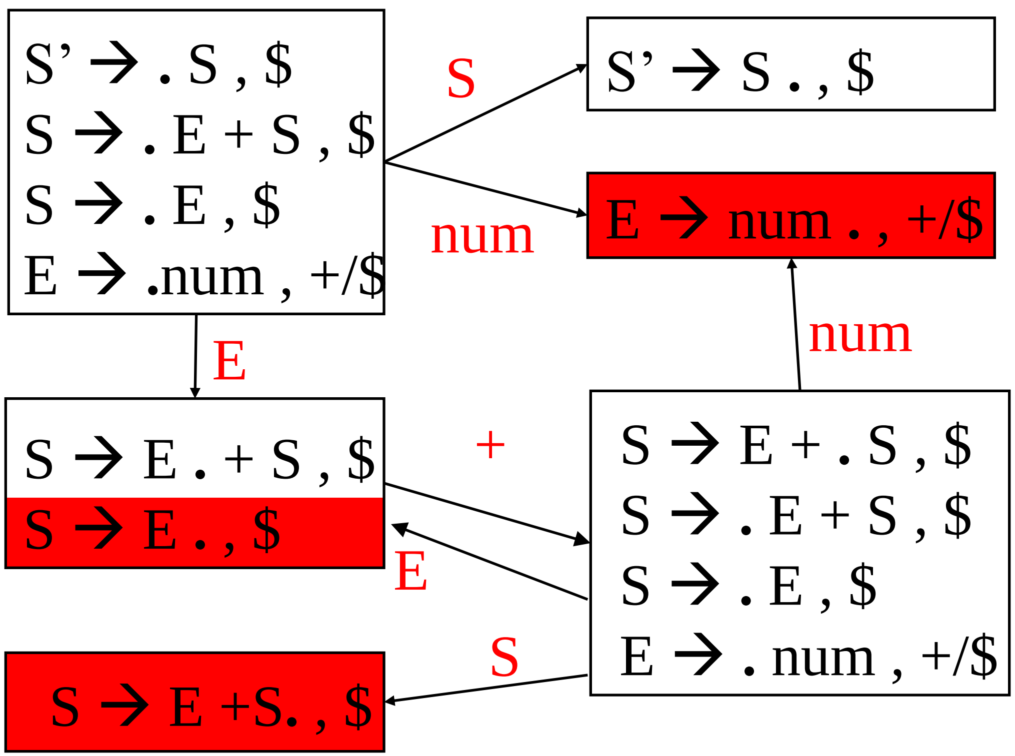
LR(1) 파서 상태 C_1 와 전이도



Grammar
S’ → S
S → E + S | E
E → num

LR(1) Reductions

- [X → γ . , y] 에 맞추어 reduce



<u>Grammar</u>

S’ → S

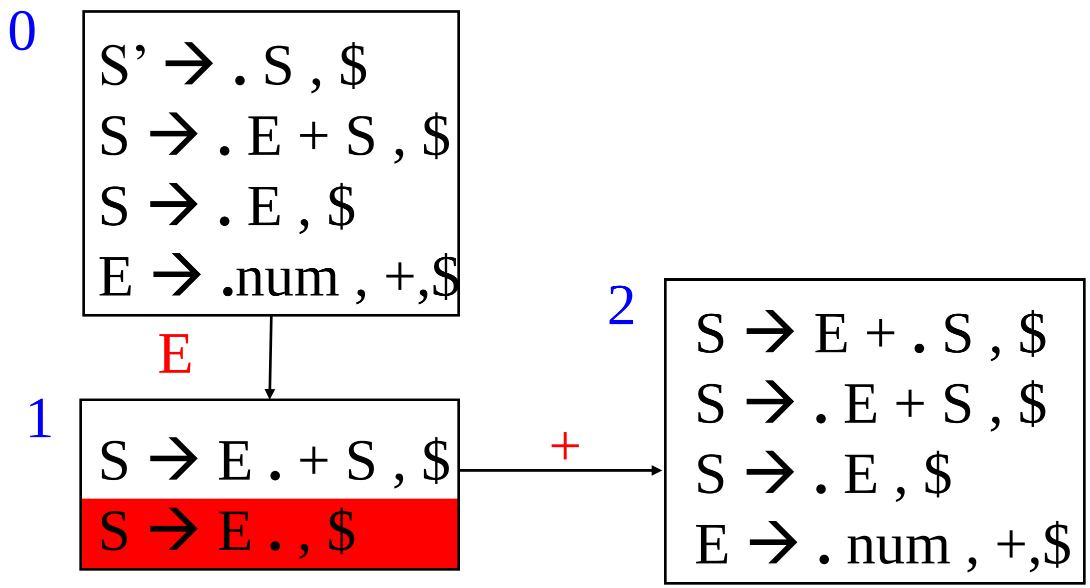
S → E + S | E

E → num

LR(1) 파싱 테이블

- Reduction 빼고 LR(0)와 동일
  - goto(I_1,a)= I_2 면: M[I_1,a] = ‘shift I_2’
  - goto(I_1,N)= I_2 면: M[I_1,N] = ‘goto I_2’
  - I_1 이 [X → β. , z] 를 포함하면:
    - M[I_1,z] = ‘reduce X → β’
  - I_1 이 [S’→S.]를 포함하면: M[I_1, $] := ‘accept’
- 참고) I_1 이 [X → β.] 를 포함하면:
  - LR(0) : M[I_1,*] = ‘reduce X → β’
  - SLR : M[I_1,a] = ‘reduce X → β’, 단 a ∊ FOLLOW(X)

LR(1) Parsing Table 예



Grammar
S’ → S
S → E + S | E
E → num

파싱테이블
(부분)

	+	$	E
0			g1
1	s2	S→E	

Class Problem

다음 문법의 LR(1) 파싱 테이블을 완성하고, 1+2 를 파싱하시오.

S’ → S
S → E + S
S →E
E → num

LALR Parsing

- LR 파싱의 문제점
  - 파서 상태 갯수가 너무 많다.
    - SLR : 수백개 for PASCAL언어
    - LR(1) : 수천개 for PASCAL언어
- LALR 파싱
  - LR에서 두 상태가 core 똑같으면 하나의 상태로 묶음.

$S \rightarrow id\ .\ ,\ +$ $S \rightarrow E\ .\ ,\ \$$	+	$S \rightarrow id\ .\ ,\ \$$ $S \rightarrow E\ .\ ,\ +$	=	??

  - SLR 보다 아주 많이 정교
    - LR(1) 보다는 이론적으로 less powerful하나 실제 처리할 수 있는 의미있는 문법의 종류가 거의 동일.
  - 파서 상태들의 갯수는 일반적으로 SLR 과 동일
    - LR(1) 보다 현저히 적은 갯수

LALR(1) 파서 만드는 방법

(1) C_1에서 작성하는 방법

- LR(1) 아이템들과 C_1 을 만든다.
- 같은 core를 가진 LR(1) 아이템 집합들을 한 개의 LR(0)아이템 집합으로 만든다.
  - 이때, 각 아이템의 lookahead는 이들 LR(1)아이템의 lookahead 의 합집합으로 구성
- 단점: 그 많은 LR(1) 파서 상태를 다 만들고 시작
  - 상태 수 문제를 극복 못함

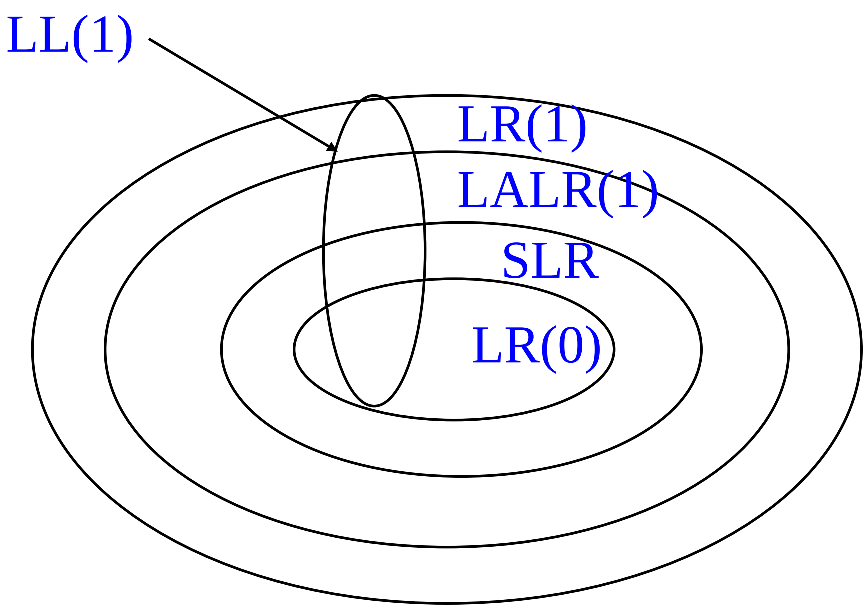
(2) C_0 에서 작성하는 방법

- LR(0) 아이템들과 C_0 을 만든다.
- 파싱 테이블의 shift와 accept 그리고 GOTO 행동은 SLR와 같고 reduce 만 수정
- 자세한 방법은 교재 참고

구문분석 방법 정리

- LL 파싱 테이블
  - Non-terminals × terminals = 생성규칙
  - FIRST/FOLLOW 로 계산
- LR 파싱 테이블
  - LR states × terminals = {shift/reduce}
  - LR states × non-terminals = goto
  - 아이템의 closure와 파서 상태들의 goto로 계산
- 어떤 문법이..
  - LL(1) 이란 것은 LL(1) 테이블에 conflict가 없단 뜻.
  - LR(0) 란 것은 LR(0) 테이블에 conflict 가 없단 뜻.
  - SLR이란 것은 SLR 테이블에 conflict 가 없단 뜻.
  - LALR(1)이란 것은 LALR(1) 테이블에 conflict 가 없단 뜻
  - LR(1)이란 것은 LR(1) 테이블에 conflict 가 없단 뜻

문법 종류들의 관계



$LR(k) \subseteq LR(k+1)$
$LL(k) \subseteq LL(k+1)$

$LL(k) \subseteq LR(k)$
$LR(0) \subseteq SLR$
$LALR(1) \subseteq LR(1)$

LR Conflict

$$E \rightarrow E + E \mid E \times E \mid num \mid (E)$$

$E \rightarrow E\,.+E$, ...
$E \rightarrow E \times E\,.$, +

1)'+ '가 들어오면 shift? reduce?

$E \rightarrow E + E\,.$, ×
$E \rightarrow E\,.\times E$, ...

2) '×' 가 들어오면 reduce? shift?

- **모호한 문법** : 파싱테이블에 shift-reduce 나 reduce-reduce conflict 야기
- **우선순위로 conflict 제어**
  - reduce/reduce : 생성 규칙 간의 우선순위로 제어
  - shift/reduce : reduce 할 생성규칙의 우선 순위가 입력 토큰 보다 높으면 reduce, 입력 토큰이 더 높으면 shift

우선순위 예 (× 먼저)
1: E × E, ×
2: E + E, +

LR Conflict (계속)

$E \rightarrow E + E$
$E \rightarrow num$

$\boxed{\begin{array}{l} E \rightarrow E + E\ .\ ,\ + \\ E \rightarrow E\ .\ + E\ ,\ +,\$ \end{array}} \xrightarrow{\color{red}{+}}$

1 + 2 + 3

shift: 1+ (2+3)
reduce: (1+2)+3

- 결합법칙도 우선순위로 제어
  - 좌측 결합 : reduce 가 일어나도록
    - 생성 규칙의 우선순위를 입력 토큰 보다 높이 둠
  - 우측결합 : shift 가 일어나도록
    - 입력 토큰의 우선순위가 생성 규칙보다 더 높음

우선순위 예
(좌측 결합)
1: E + E
2: +

Yacc에서 Conflict 해결 방법

- Yacc 해결방안 : **선언부에서 우선순위 지정**
  - Associativity – (left, right, none)
    - %left   TK_PLUS
    - %right TK_ASSIGNMENT
    - %nonassoc TK_LESSTHAN

      … 우선순위 높은 것이 아래에
  - Precedence
    - 규칙 우선순위를 기술
    - %prec

Yacc = LALR(1) + alpha

Yacc 예 1-사칙연산

```
%left '+' '-'
%left '*' '/'
%right '='
%nonassoc UMINUS
expression: expression '+' expression
          | expression '-' expression
          | expression '*' expression
          | expression '/' expression
          | '-' expression %prec UMINUS
          | NUMBER ;
```

- 나열된 순서대로 +,-가 가장 낮고, UMINUS 가 가장 높음.
- '-' expression 는 UMINUS로 지정된 우선순위를 사용하라는 의미.

Yacc 예 2 – If 의 모호성 제거

- 가장 가까운 if 로 else 를 엮는 방법
  - 문법 자체를 바꾸거나 (배운적 있음),
  - 다음과 같이 yacc 을 구성(과제 때문에 알려준 적 있음)

%nonassoc LOWER_THAN_ELSE

%nonassoc ELSE

statement : if expr statement **%prec** LOWER_THEN_ELSE

| if expr statement ELSE statement