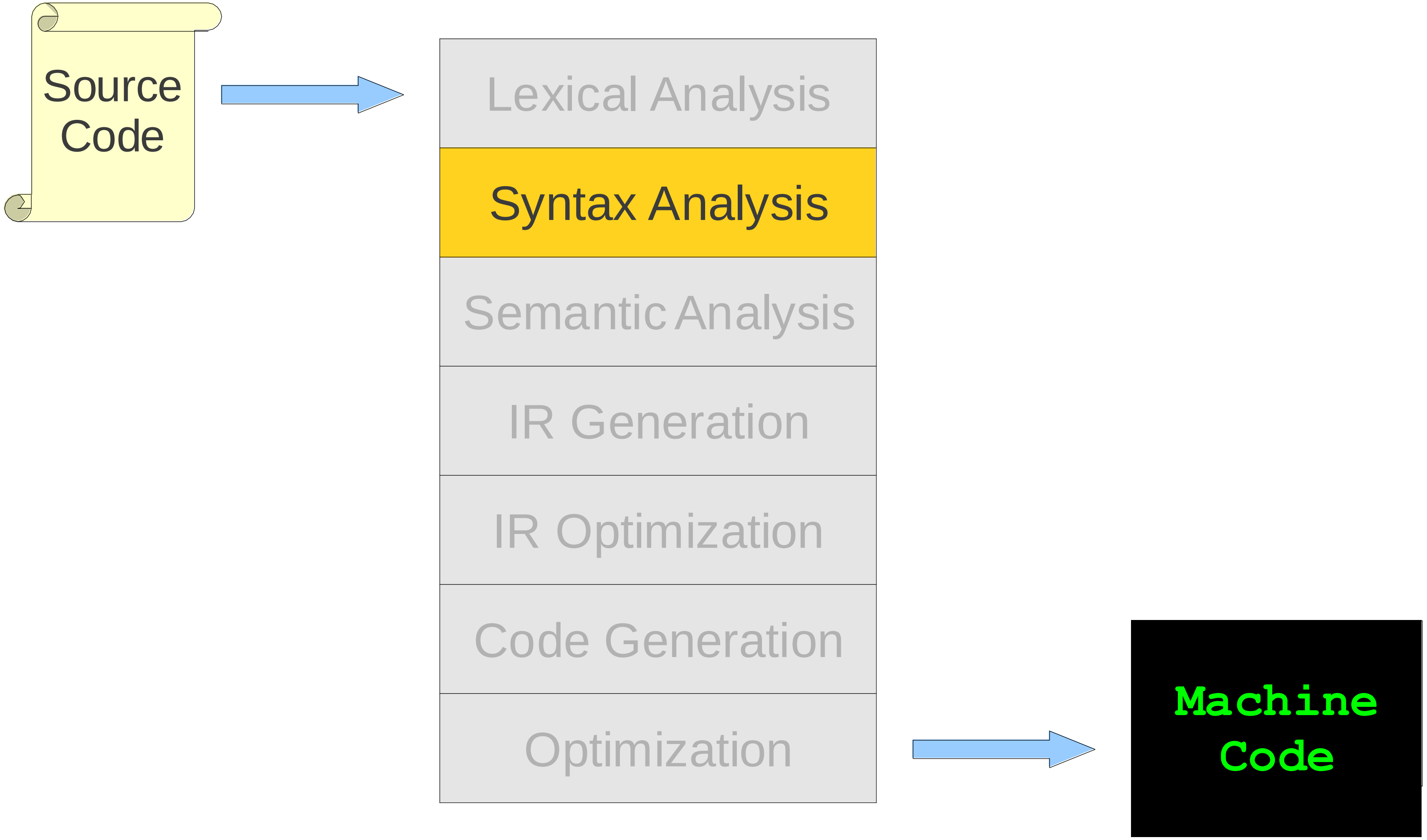
Compiler
– 3–6. Simple LR, More About LR –

JIEUNG KIM

jieungkim@inha.ac.kr

YONSEI UNIVERSITY

Where are we?



YONSEI UNIVERSITY

Outlines

- Role of the syntax analysis (parser)
- Context free grammar
- Push down automata
- Top-down parsing
- Bottom-up parsing
- Simple LR
- More powerful LR parsers and other issues in parsers
- Syntactic error handler
- Parser generator

Simple LR (SLR)

Simple LR (SLR)

- LR conflicts
  - A **shift/reduce conflict** is an error where a shift/reduce parser cannot tell whether to shift a token or perform a reduction.
    - Often happens when two productions overlap.
  - A **reduce/reduce conflict** is an error where a shift/reduce parser cannot tell which of many reductions to perform.
    - Often the result of ambiguous grammars.
  - A grammar whose handle-finding automaton contains a shift/reduce conflict or a reduce/reduce conflict is not LR(0).

Simple LR (SLR)

- What conflicts mean
  - Recall: our automaton was constructed by looking for viable prefixes.
  - Each accepting state represents a point where the handle might occur.
  - A **shift/reduce conflict** is a state where the handle might occur, but we might need to keep searching.
  - A **reduce/reduce conflict** is a state where we know we have found the handle but can't tell which reduction to apply.

Simple LR (SLR)

- **Why LR(0) is weak**
  - LR(0) only accepts languages where the handle can be found with no **right context**.
  - Our shift/reduce parser only looks to the left of the handle, not to the right.
  - How do we exploit the tokens after a possible handle to determine what to do?
- **Why simple LR?**
  - We have built the LR(0) state machine and parser tables
    - No lookahead yet
    - Different variations of LR parsers add lookahead information, but basic idea of states and edges remains the same
  - A grammar that is not LR(0)
    - Build the state machine and parse tables for a simple expression grammar
    - Add `$` in the grammar

Grammar:

```
S ::= E$
E ::= id + E
E ::= id
```

Simple LR (SLR)

- Simple LR (SLR)
  - Idea
    - Use information about what can follow a non-terminal to decide if we should perform a reduction
  - Easiest form is SLR – Simple LR
  - It requires the following "`FOLLOW`" (we learned it in the previous slide – LL parser)
  - `FOLLOW(A)` – the set of symbols that can follow `A` in any possible derivation

Simple LR (SLR)

- SLR grammers
  - SLR (Simple LR): SLR is a simple extension of LR(0) shift-reduce parsing
  - SLR eliminates some conflicts by populating the parsing table with reductions `A::=`α on symbols in `FOLLOW(A)`

```
S ::= E$
E ::= id + E
E ::= id
```

State I1:
`S ::= •E`
`E ::= •id + E`
`E ::= •id`

goto(I1,**id**)

State I3:
`E ::= id•+ E`
`E ::= id•`

goto(I3,**+**)

Shift on +

FOLLOW(E)={**$**}
thus reduce on **$**

Simple LR (SLR)

- SLR parsing table
  - Reductions do not fill entire rows
  - Otherwise, the same as LR(0)

```
1. S ::= E$
2. E ::= T + E
3. E ::= T
4. T ::= id
```

State	*Action*			*Goto*	
	id	+	$	E	T
1	s5			g2	g3
2			acc		
3		s4	r3		
4	s5			g6	g3
5		r4	r4		
6			r2		

Shift on **+**

FOLLOW(E)={**$**}
thus reduce on **$**

Simple LR (SLR)

- Simple LR (SLR)
  - This is identical to LR(0) – states, etc., except for the calculation of reduce actions
  - Algorithm

```
Initialize R to empty
for each state I  in T
  for each item [A ::= α .] in I
    for each terminal a in FOLLOW(A)
      add (I, a, A ::= α) to R
```

- i.e., reduce α to A in state I only on lookahead a

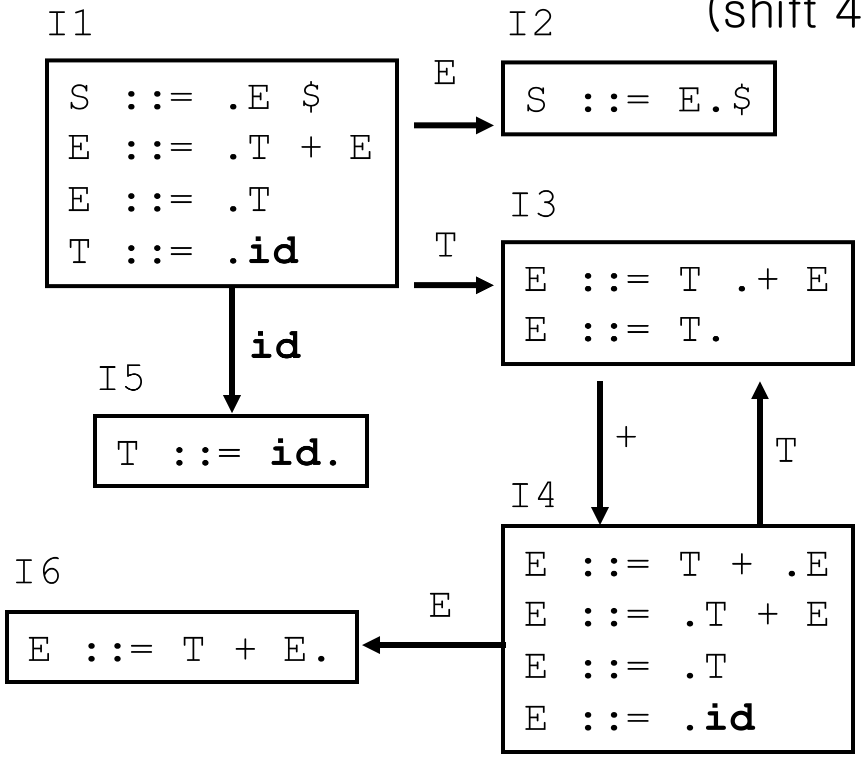
Simple LR (SLR)

- Examples – simple LR(SLR)

State I3 has two possible actions on +
(shift 4, or reduce 2), so the grammar is not LR(0)

Grammar:

1. S ::= E$
2. E ::= T + E
3. E ::= T
4. T ::= **id**



State	*Action*			*Goto*	
	id	+	$	E	T
1	s5			g2	g3
2			acc		
3	r3	s4,r3	r3		
4	s5			g6	g3
5	r4	r4	r4		
6	r2	r2	r2		

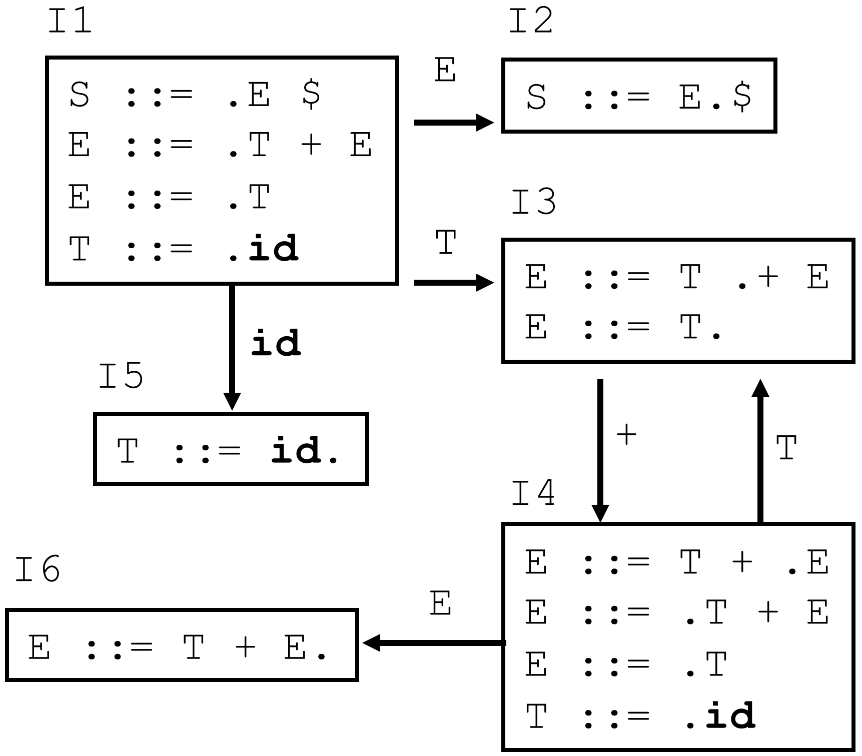
YONSEI UNIVERSITY

Simple LR (SLR)

- Examples – simple LR(SLR)

Grammar:

1. S ::= E$
2. E ::= T + E
3. E ::= T
4. T ::= **id**



State	*Action*			*Goto*	
	id	+	$	E	T
1	s5			g2	g3
2			acc		
3	r3	s4, r3	r3		
4	s5			g6	g3
5	r4	r4	r4		
6	r2	r2	r2		

Simple LR (SLR) – self study slide

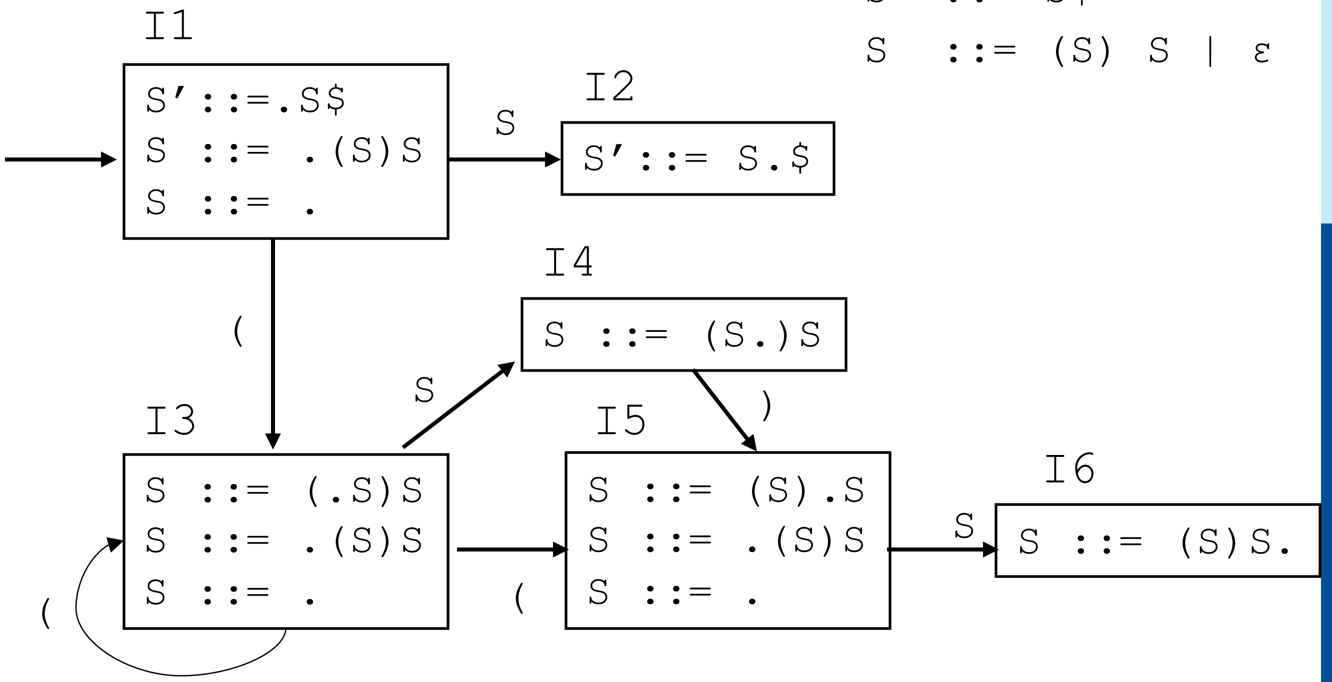
- Examples – simple LR(SLR)

Grammar:

S′ ::= S$

S ::= (S) S | ε

S	*action*	*rules*	*input* (	*input*)	*goto* S
1	???	S ::= ε	s3		g2
2	Reduce	S′ ::= S			
3	???	S ::= ε	s3		g4
4	Shift			s5	
5	???	S ::= ε	s3		g6
6	Reduce	S ::= (S)S			



Simple LR (SLR) – self study slide

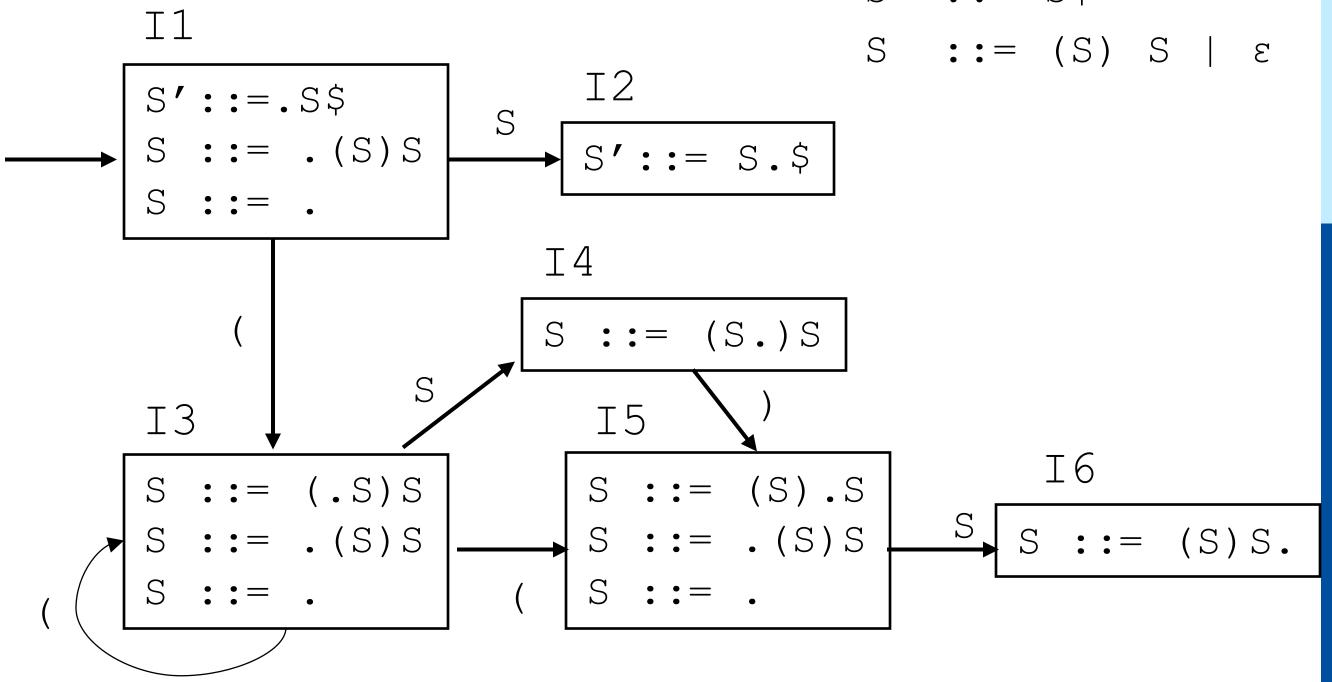
- Examples – simple LR(SLR)

Grammar:

S’ ::= S$

S ::= (S) S | ε

State	action			goto
	(	)	$	S
1	s3	r(S::=ε)	r(S::=ε)	g2
2			accept	
3	s3	r(S::=ε)	r(S::=ε)	g4
4		s5		
5	s3	r(S::=ε)	r(S::=ε)	g6
6		r(S::=(S)S)	r(S::=(S)S)	



Simple LR (SLR) – self study slide

- Examples – simple LR(SLR)

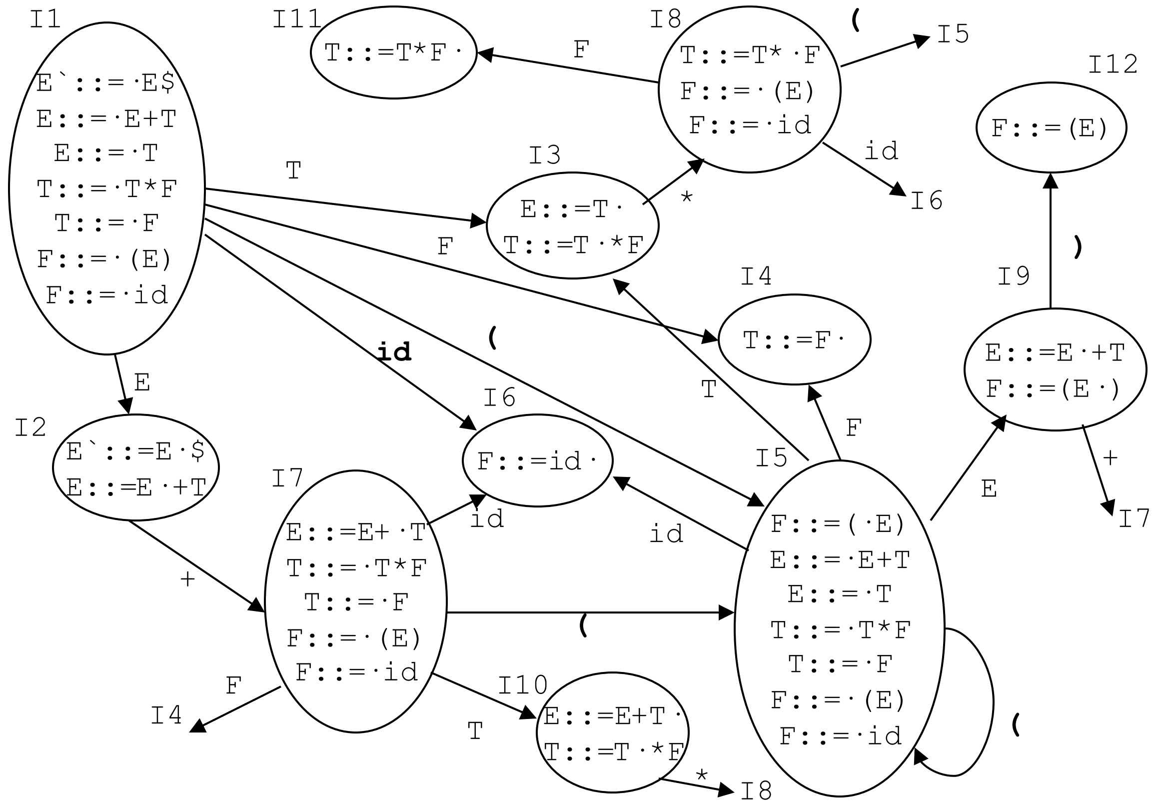
State	*action*			*goto*
	(	)	$	S
1	s3	r(S::=ε)	r(S::=ε)	2
2			accept	
3	s3	r(S::=ε)	r(S::=ε)	4
4		s5		
5	s3	r(S::=ε)	r(S::=ε)	6
6		r(S::=(S)S)	r(S::=(S)S)	

Stack	Input	Action
$1	()() $	shift 3
$1(3	)() $	reduce S ::= ε
$1(3S4	)() $	shift 5
$1(3S4)5	() $	shift 3
$1(3S4)5(3	) $	reduce S ::= ε
$1(3S4)5(3S4	$	shift 5
$1(3S4)5(3S4)5	$	reduce S ::= ε
$1(3S4)5(3S4)5S6	$	reduce S ::= (S) S
$1(3S4)5S6	$	reduce S ::= (S) S
$1S2	$	accept

YONSEI UNIVERSITY

Simple LR (SLR) – self study slide

- Examples – simple LR(SLR)



Grammar:

E′ ::= E$

E ::= E + T | T

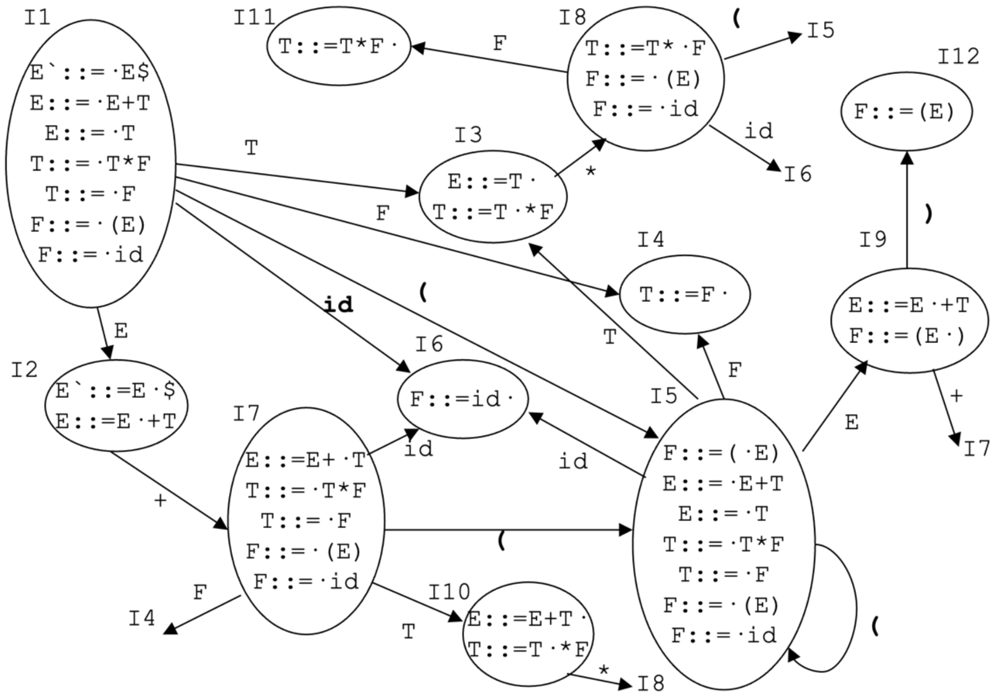
T ::= T * F | F

F ::= (E) | **id**

Grammar:

1. E′ ::= E
2. E ::= E **+** T
3. E ::= T
4. T ::= T ***** F
5. T ::= F
6. F ::= **(** E **)**
7. F ::= **id**

State	*action*						*goto*		
	id	+	*	(	)	$	E	T	F
1	s6			s5			g2	g3	g4
2		s7				acc			
3		r3	s8		r3	r3			
4		r5	r5		r5	r5			
5	s6			s5			g9	g3	g4
6		r7	r7		r7	r7			
7	s6			s5				g10	g4
8	s6			s5					g11
9		s7			s12				
10		r2	s8		r2	r2			
11		r4	r4		r4	r4			
12		r6	r6		r6	r6			



Grammar:

1. E′ ::= E
2. E ::= E **+** T
3. E ::= T
4. T ::= T ***** F
5. T ::= F
6. F ::= **(** E **)**
7. F ::= **id**

State	*action*						*goto*		
	id	+	*	(	)	$	E	T	F
1	s6			s5			g2	g3	g4
2		s7				acc			
3		r3	s8		r3	r3			
4		r5	r5		r5	r5			
5	s6			s5			g9	g3	g4
6		r7	r7		r7	r7			
7	s6			s5				g10	g4
8	s6			s5					g11
9		s7			s12				
10		r2	s8		r2	r2			
11		r4	r4		r4	r4			
12		r6	r6		r6	r6			

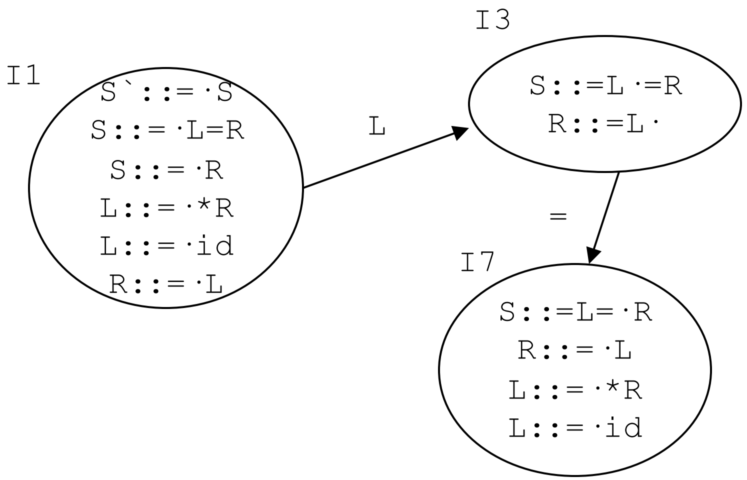
Stack	Input	Action
$ 1	id*id+id$	s6
$ 1 id 6	*id+id$	r7 g4
$ 1 F 4	*id+id$	r5 g3
$ 1 T 3	*id+id$	s8
$ 1 T 3 * 8	id+id$	s6
$ 1 T 3 * 8 id 6	+id$	s7 g11
$ 1 T 3 * 8 F 11	+id$	r4 g3
$ 1 T 3	+id$	r3 g2
$ 1 E 2	+id$	s7
$ 1 E 2 + 7	id$	s6
$ 1 E 2 + 7 id 6	$	s7 g3
$ 1 E 2 + 7 F 4	$	s5 g10
$ 1 E 2 + 7 T 10	$	s3 g2
$ 1 E 2	$	accept

More powerful LR parsers & other issues in parsers

LR(1) and LALR(1)

- Is SLR enough?
  - A grammar that is not ambiguous, not SLR

```
S  ::=  L=R
S  ::=  R
L  ::=  *R
L  ::=  id
R  ::=  L
```



  - FOLLOW(R) = FLLOW(S) = FOLLOW(L) = { = }
  - Action(3,=) ➔ Shift or Reduce
    - Because SLR is not powerful enough to remember sufficient left context to decide next action on "="

LR(1) and LALR(1)

- **LR(1) and LALR(1)**
  - CanonicalLR(1) (vs SLR)
    - Solve problems in SLR
    - Complexity is too high (LR(1) is not used in ordinary cases – the parsing table becomes too big)
  - LALR(1) (vs SLR and LR(1))
    - $L_{ook}A_{head}$ LR(1)
    - It almost preserves all advantages of LR(1), but also preserves efficiencies of SLR
    - It is an internal engine of multiple parser generators, such as Yacc and ocamlyacc

LR(1)

- LR(1)
  - Many practical grammars are SLR
  - LR(1) is more powerful yet
  - Similar construction, but notion of an item is more complex, incorporating lookahead information
  - An LR(1) item $[A ::= \alpha.\beta,\ a]$ (v.s. LR(0) item $[A ::= \alpha.\beta]$)
    - A grammar production ($A ::= \alpha\beta$)
    - A right-hand side position (the dot)
    - A lookahead symbol a when $a \in \mathrm{FOLLOW}(A)$
      - The lookahead symbol a has no effect when $\beta \neq \varepsilon$
  - Idea: This item indicates that α is the top of the stack and the next input is derivable from βa
  - Pro: extremely precise; largest set of grammars
  - Con: potentially very large parse tables with many states

YONSEI UNIVERSITY

LALR(1)

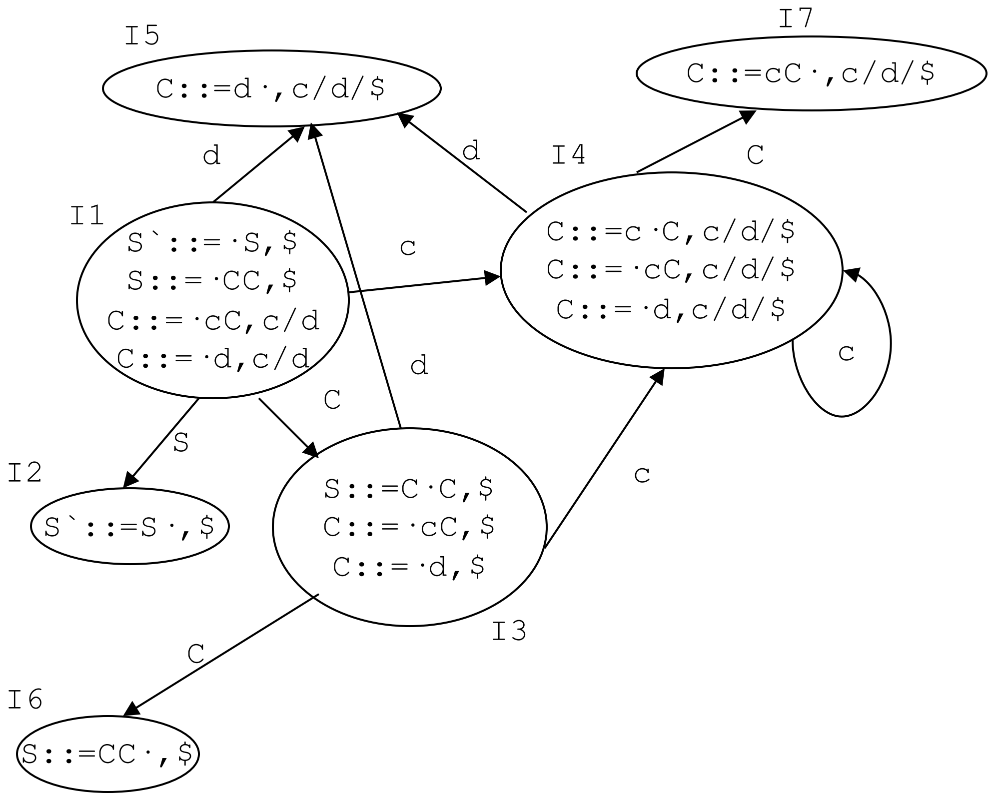
- LALR(1)
  - Variation of LR(1)
  - Often used in practice
  - SLR and LALR have the same number of states (in general several hundred for PASCAL)
  - Look for set of LR(1) items having the same core, and merge these sets with the common cores into one set of items
  - For example, these two would be merged

    [A ::= x., a]

    [A ::= x., b]

LALR(1) – self-study page

- Example – LALR(1)



Grammar:

1. S'::= S$
2. S ::= CC
3. C ::= cC
4. C ::= d

State	Action			Goto	
	c	**d**	**$**	**S**	**C**
1	S4	S5		2	3
2			Acc		
3	S4	S5			6
4	S4	S5			7
5	R4	R4	R4		
6			R2		
7	R3	R3	R3		

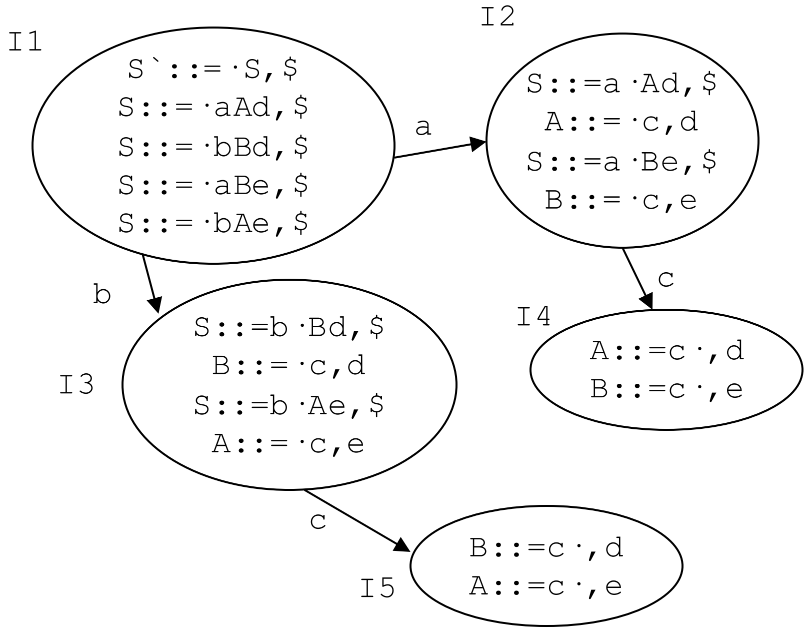
YONSEI UNIVERSITY

LR(1) vs LALR(1)

- LR(1) vs LALR(1)
  - LALR(1) tables can have many fewer states than LR(1)
  - LALR(1) may have reduce conflicts where LR(1) would not (but in practice this doesn't happen often)

LR(1) vs LALR(1) – self study slide

- Example – LR(1) vs LALR(1)



LR(1)

Grammar:

S’::= S$

S ::= aaAd | bBd | aBe | bAe

A ::= c

B ::= c

- In LALR, the two states (A::=c·,d B::= c·,e) and (A::=c·,e B::=c·,d) are merged
- It will raise a reduce/reduce conflict occurs when lookahead symbol is either d or e

Ambiguous grammars

- **Using ambiguous grammars**
  - Every ambiguous grammar fails to be a LR
  - An ambiguous grammar provides a shorter and more natural specification
  - Disambiguating rules
    - Use precedence and associativity
    - "Dangling else" ambiguity by favorable choice
    - Ambiguities from special case productions

Ambiguous grammars

- Use precedence and associativity
  - Example
    - `E ::= E + E | E * E | (E) | id`
    - The given grammar creates finite states
      - `I8:    E ::= E + E·          I9:    E ::= E * E·`
        `       E ::= E· + E                 E ::= E· + E`
        `       E ::= E· * E                 E ::= E· * E`
      - `FOLLOW(E) = {+, *, ), $}`
      - Thus, when `id + id * id` is an input string, the configuration could be `1E2 + 5E8` and `* id$`
  - Using the precedence
    - Assume `*` takes precedence over `+`, then shift `*` onto the stack, preparing reduce the `*` and its surrounding `id`'s to an expression
    - Assume `+` takes precedence over `*`, then parser should reduce `E + E` to `E`

Ambiguous grammars

- Use precedence and associativity
  - Using the associativity
    - Suppose `id+id+id` is processed. Then stack contains `1E2+5E8` after `id+id`
    - If + is left-associative, reduce by `E::=E+E` (`id`'s surrounding the first + should be grouped first)
    - Equivalent grammar is
      - `E::=E+T | T`
        `T::=T*F | F`
        `F::=(E) | id`

Ambiguous grammars – self-study page

- Use precedence and associativity

1. S' ::= E
2. E ::= E **+** E
3. E ::= **id**

state	*action*			*goto*
	id	+	$	E
1	s3			2
2		s4	acc	
3		r3	r3	
4	s3			5
5		s4/r2	r2	

Shift/reduce conflict:
action(5,**+**) = shift 4
action(5,**+**) = reduce E ::= E **+** E

stack	input
$ 1	id+id+id$
…	…
$ 1 E 2 **+** 4 E 5	+id$

When shifting on +: yields right associativity
id+(id+id)

When reducing on +: yields left associativity
(id+id)+id

Ambiguous grammars – self-study page

- Use precedence and associativity
  - Left-associative operators: reduce
  - Right-associative operators: shift
  - Operator of higher precedence on stack: reduce
  - Operator of lower precedence on stack: shift

```
1. S'::= E
2. E ::= E + E
3. E ::= E * E
4. E ::= id
```

stack	input
$ 1	id*id+id$
…	…
$ 1 *E* 2 ***** 4 *E* 6	+id$

reduce E ::= E ***** E

Ambiguous grammars

- "Dangling else" ambiguity by favorable choice
  - `S' ::=S`
    `S ::= iSeS | iS | a`
  - makes a state `I5: S ::= iS·eS  S ::= iS·` (shift reduce conflict)
    - "in favor of shift on input `e`"

Ambiguous grammars

- **Ambiguities from special case productions**
  - Problem
    - When we add an extra productions, it may be possible to have a parsing action conflict
  - Resolution
    - Reduce by the special case production

Ambiguous grammars

Example grammar:

```
E ::= E sub E sup E
E ::= E sub E
E ::= E sup E
E ::= { E }
E ::= c
```

- Ambiguities from special case productions
  - Example
    - With the presented grammar, we assume that sub and sup do not have any precedence relationship nor associativity
    - Then, one of the states(I8) contains the following cores and have shift/reduce conflict
      - In this case, a resolution is to process it in favor of shift action

```
E ::= E · sub E sup E          E ::= E sub E · sup E
E ::= E · sub E                E ::= E sub E ·
E ::= E · sup E
```

    - Also, I11 contains the following cores and have a reduce/reduce conflict on inputs } and $

```
E ::= E sub E sup E ·          E ::= E sup E ·
```

    - If we prefer the first production, then E ::= E sub E sup E is the special case

LR, SLR, LALR summary

- LR, SLR, LALR summary

	Lookahead	Item	Advantage	Problem
LR(0)	0	LR(0)	–	Lack of expressiveness
SLR	1	LR(0)	Solve expressiveness problems in LR(0)	Lack of expressiveness High complexity
Canonical LR(1)	1	LR(1)	Solve expressiveness problems in SLR	High complexity Too big table size
LALR(1)	1	LR(0) ≤ core < LR(1)	Solve complexity in LR(1) Keep expressiveness in LR(1)	

LL, LR Summary

- LL, SLR, LR, LALR Summary

	Advantage	Problem
To-down recursive descent – LL(1)	Fast Good locality Simplicity Good error detection	Hand-coded High maintenance Right associativity
LR(1)	Fast Deterministic langs Automatable Left associativity	Large working sets Poor error messages Large tables sizes

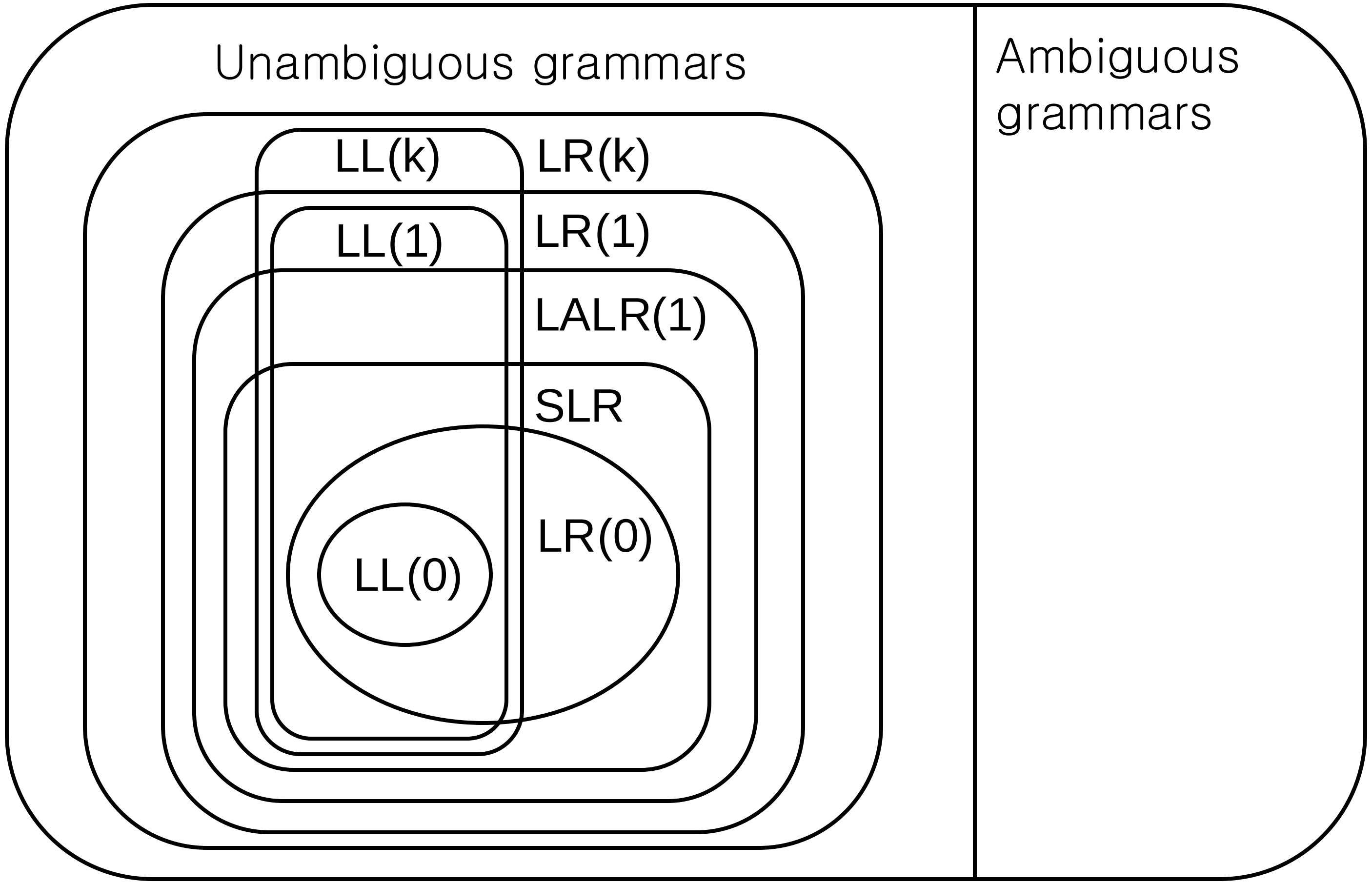
YONSEI UNIVERSITY

LL, SLR, LR, LALR Summary

- LL, SLR, LR, LALR Summary
  - LL parse tables computed using `FIRST/FOLLOW`
    - `Nonterminals × terminals → productions`
    - Computed using `FIRST/FOLLOW`
  - LR parsing tables computed using `action/goto`
    - `LR states × terminals → shift/reduce actions`
    - `LR states × nonterminals → goto state transitions`
  - A grammar is
    - LL(1) if its LL(1) parse table has no conflicts
    - SLR if its SLR parse table has no conflicts
    - LALR(1) if its LALR(1) parse table has no conflicts
    - LR(1) if its LR(1) parse table has no conflicts

LL, SLR, LR, LALR Summary

- LL, SLR, LR, LALR Summary



Questions?