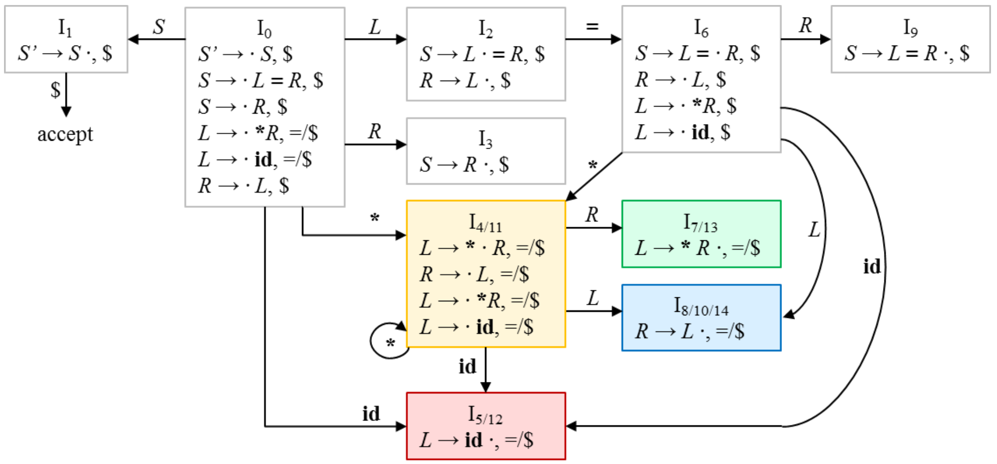
SLR, LR and LALR Parsing



SLR Parsing

CSE 컴퓨터공학부
Department of Computer Science & Engineering

SLR Parsing

- "Simple LR" Parsing
  - built on valid items and viable prefixes
  - refines when to shift and when to reduce
    - fewer states have conflicts

  - SLR(k)
    - k = number of lookahead symbols
    - in practice k = 1

SLR Parsing

- SLR Parser
  - next input $\mathbf{t}$
  - stack contains α
  - LR(0) DFA recognizing viable prefixes
    - terminates in state s on input α

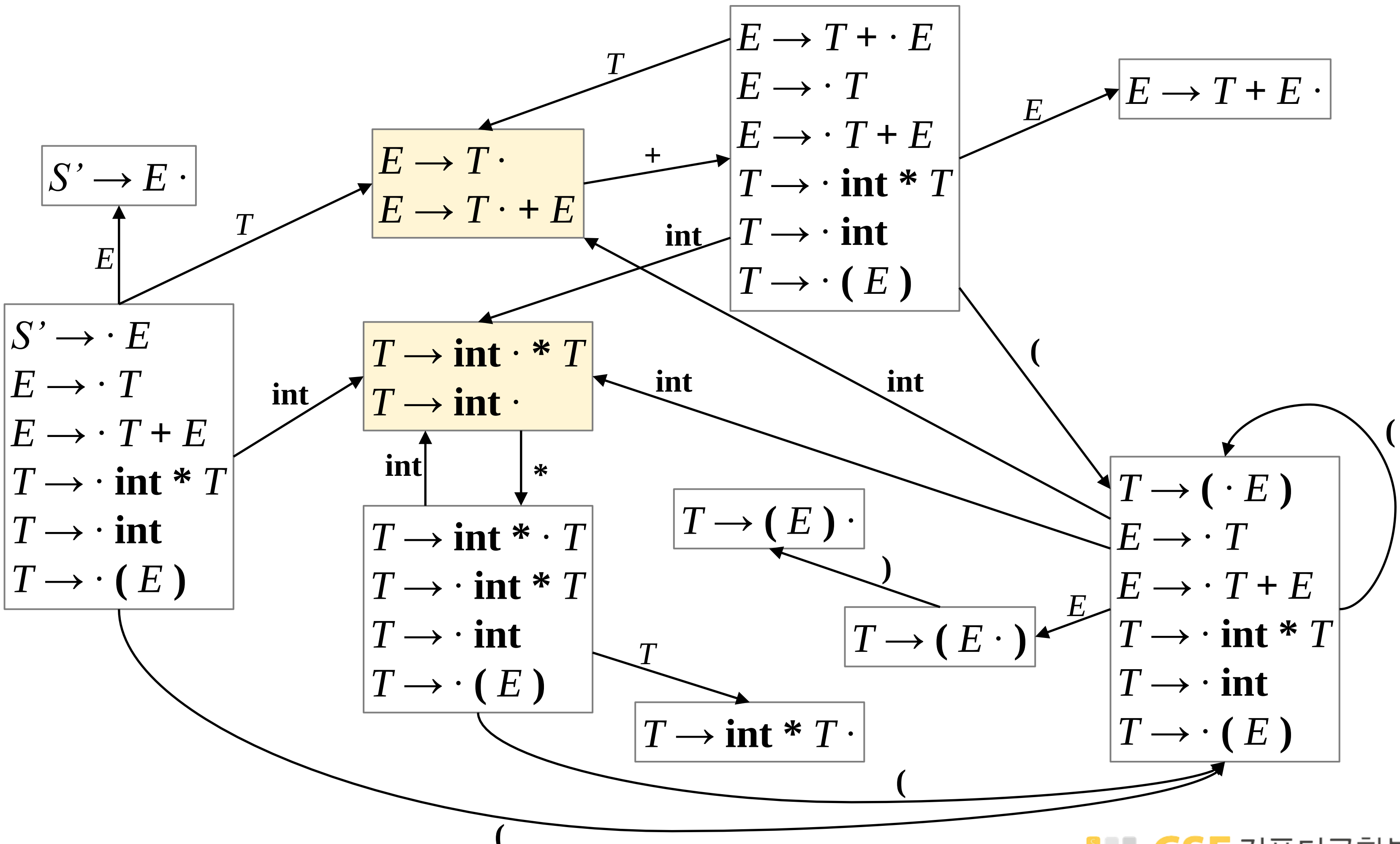
  - actions:
    - if s contains the item $X \rightarrow \beta \cdot$ and $\mathbf{t} \in \mathrm{FOLLOW}(X)$ then **reduce** $X \rightarrow \beta$
    - if s contains the item $X \rightarrow \beta \cdot \mathbf{t}\, \omega$ then **shift**

  - SLR grammars $\leftrightarrow$ no conflicts
    - heuristics unambiguously detect the handles
    - ambiguous grammars are not SLR

    → use precedence rules

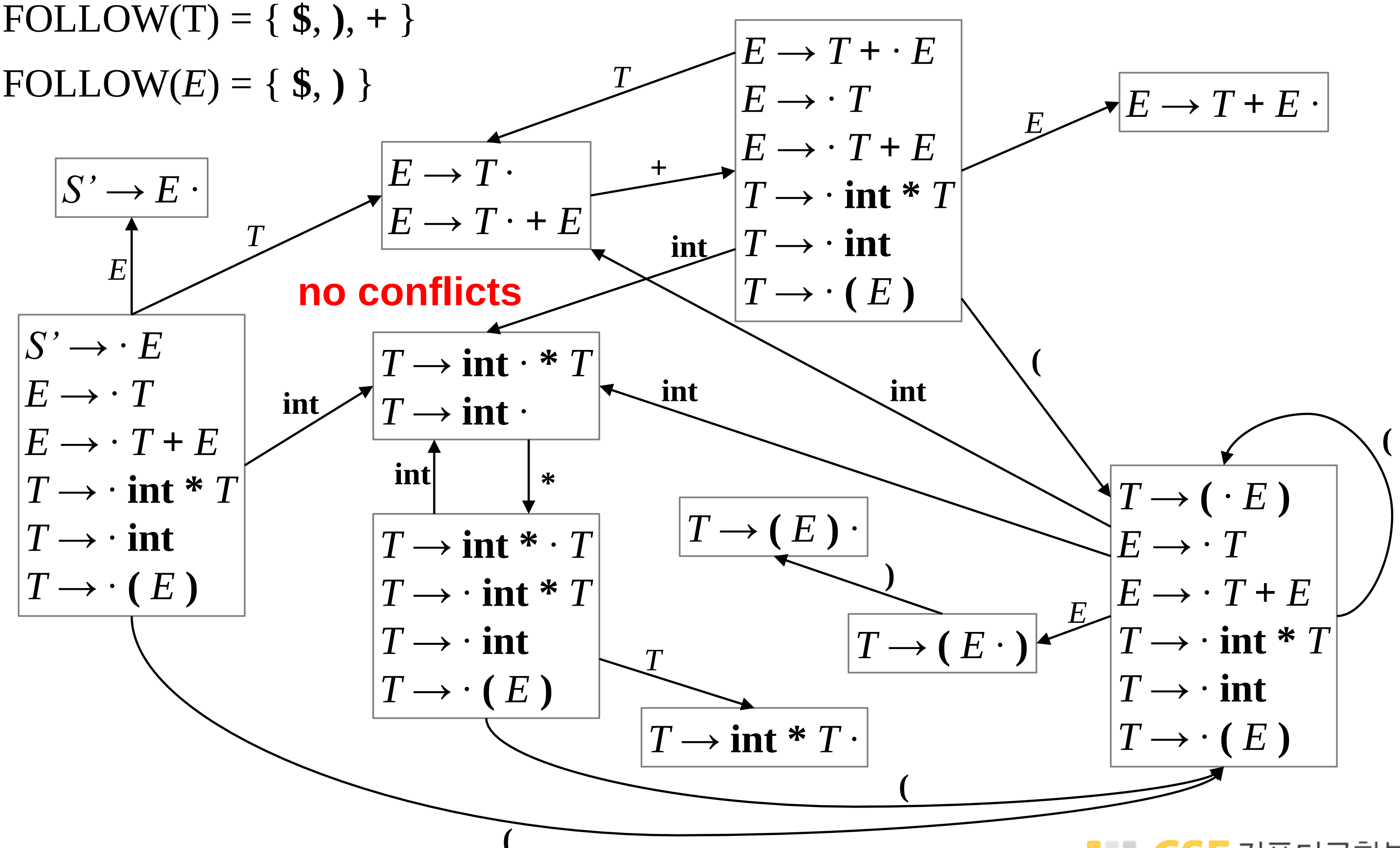
SLR Parsing

- Example: LR(0)/SLR DFA



SLR Parsing

- Example: LR(0)/SLR DFA
  - FOLLOW(T) = { **$,), +** }
  - FOLLOW(*E*) = { **$,)** }



SLR Parsing

- Many grammars are not SLR
  - ambiguous grammars

  - parse a bigger class of grammars using an SLR parser
    - define rules for resolving conflicts
      - precedence

SLR with Ambiguous Grammars

- Ambiguous grammar

$$E \quad \rightarrow \quad E + E \mid E * E \mid (E) \mid \textbf{int}$$

- LR(0) DFA contains a state with the following items
  - shift/reduce conflict

$$\boxed{\begin{array}{l} E \rightarrow E * E \cdot \\ E \rightarrow E \cdot + E \\ \ldots \end{array}}$$

  - precedence declaration
    - define conflict resolution rules, *not* precedence

SLR(1) Parsing Algorithm

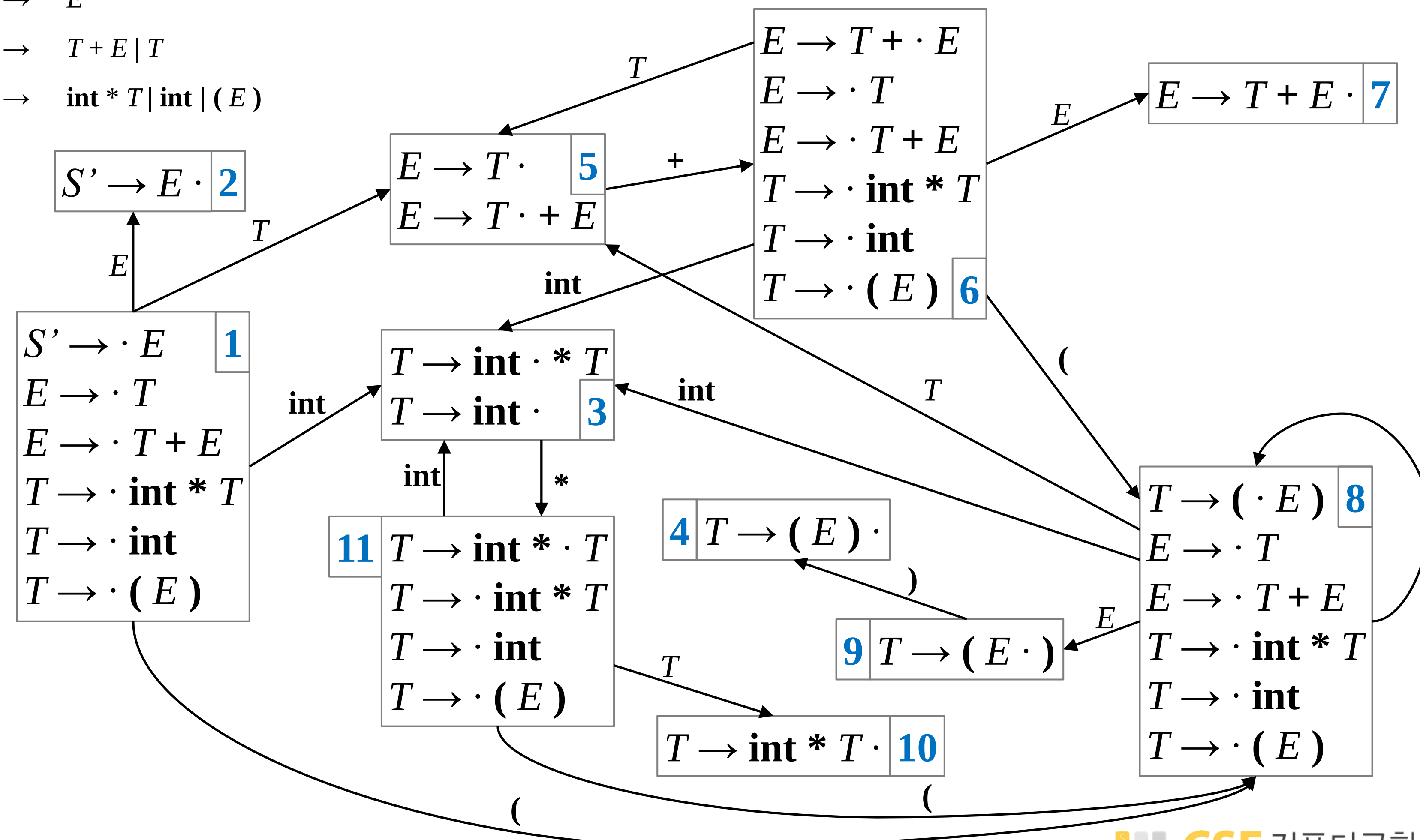
- let M be the DFA for viable prefixes of G
- let | $\mathbf{t_1 t_2 \ldots t_n \$}$ be the initial configuration
- repeat until configuration is $\mathrm{S} \mid \$$
  - input configuration: $\alpha \mid \mathbf{t}\ \omega$
  - run M on stack α
    - *if M rejects α, report parsing error*
    - if M accepts in state s
      with I = elements of s, next input symbol = $\mathbf{t}$
      - shift
        if $X \rightarrow \beta \cdot \mathbf{t}\ \gamma \in I$
      - reduce
        $X \rightarrow \beta \cdot \in I$ and $\mathbf{t} \in \mathrm{FOLLOW}(X)$
      - *otherwise report a parse error*

- if conflicts exist, the grammar is not SLR(1)

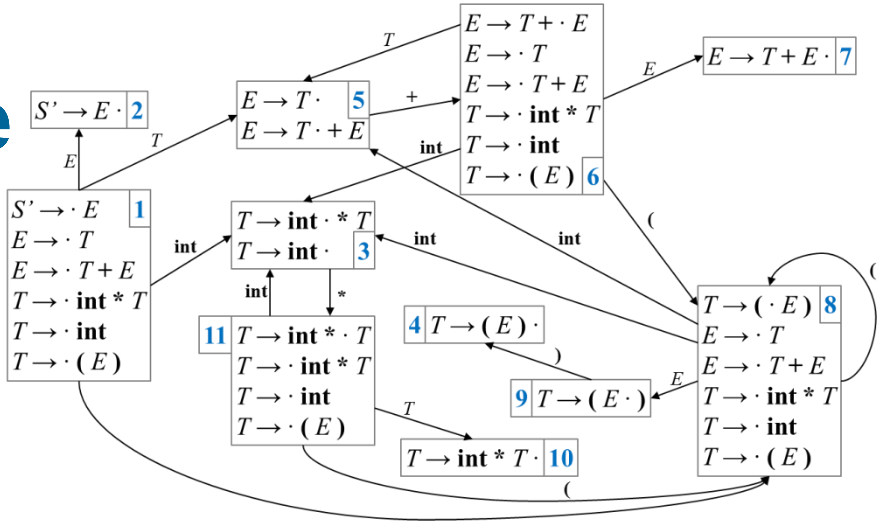
SLR(1) Parsing Example

- LR(0) Parsing Automaton

S'	$\rightarrow$	E
E	$\rightarrow$	$T + E \mid T$
T	$\rightarrow$	**int** $* \; T \mid$ **int** $\mid (E)$

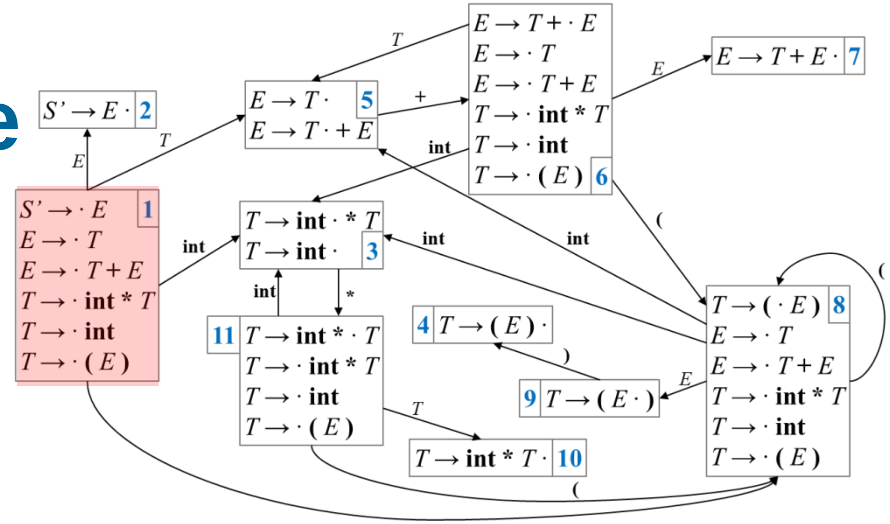


SLR(1) Parsing Example



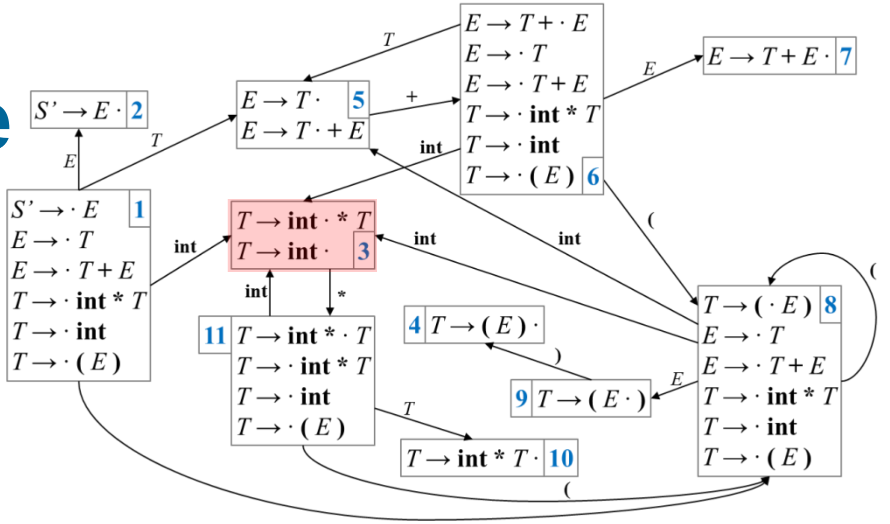
Configuration	DFA	Action	
| int * int $			

SLR(1) Parsing Example



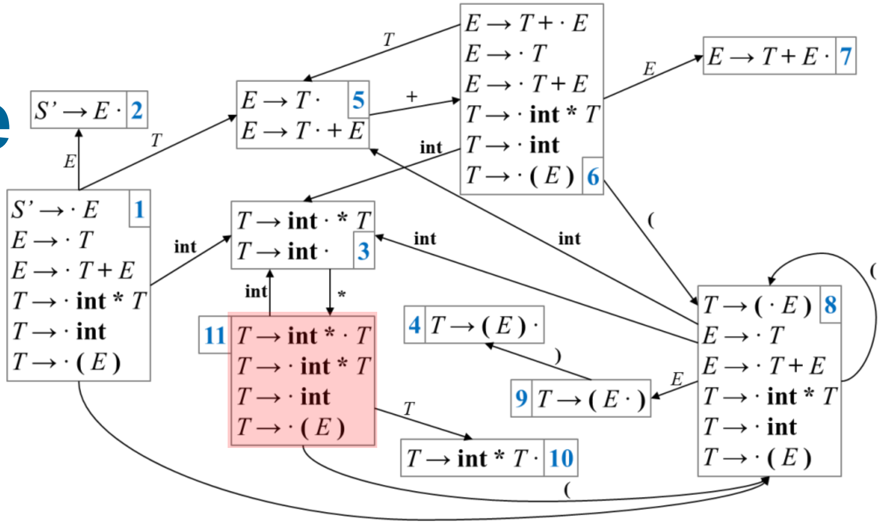
Configuration	DFA	Action	
| **int** * **int** $	1		

SLR(1) Parsing Example



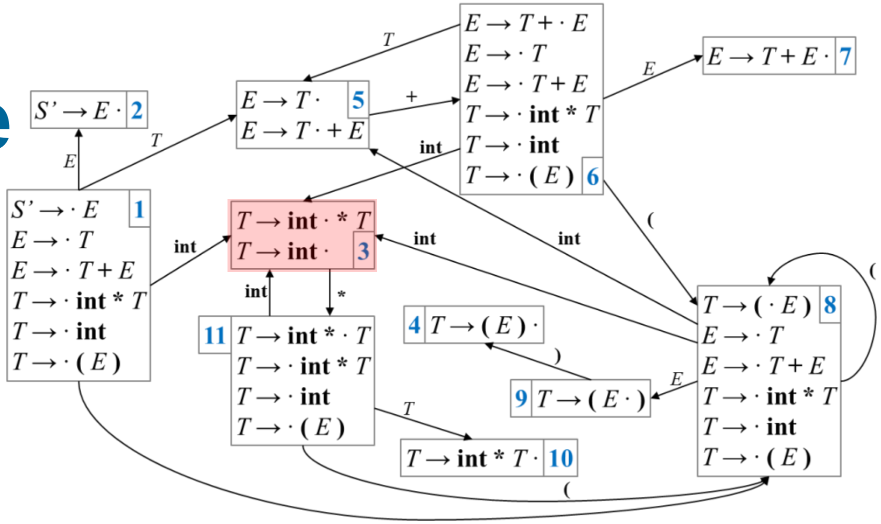
Configuration	DFA	Action	
| **int** * **int** $	1	shift	
int | * **int** $	3		

SLR(1) Parsing Example



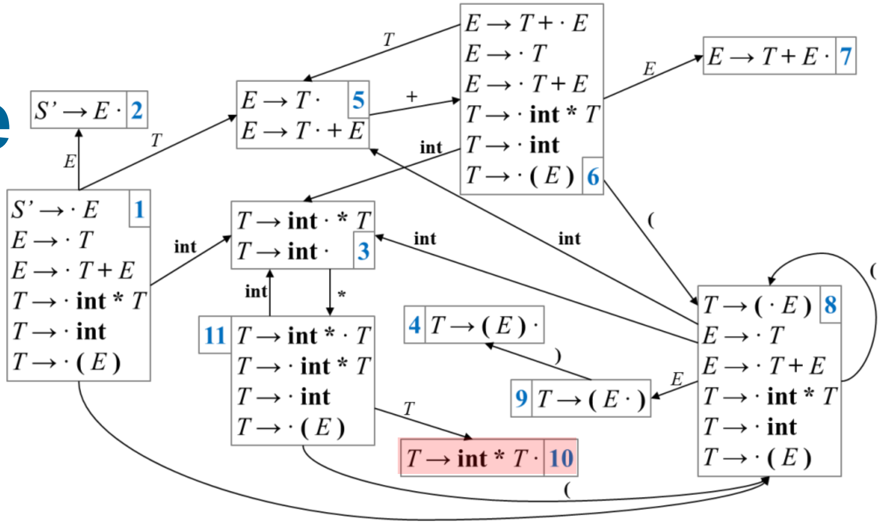
Configuration	DFA	Action	
| **int** * **int** $	1	shift	
int | * **int** $	3	shift	* not in FOLLOW(T)
int * | **int** $	11		

SLR(1) Parsing Example



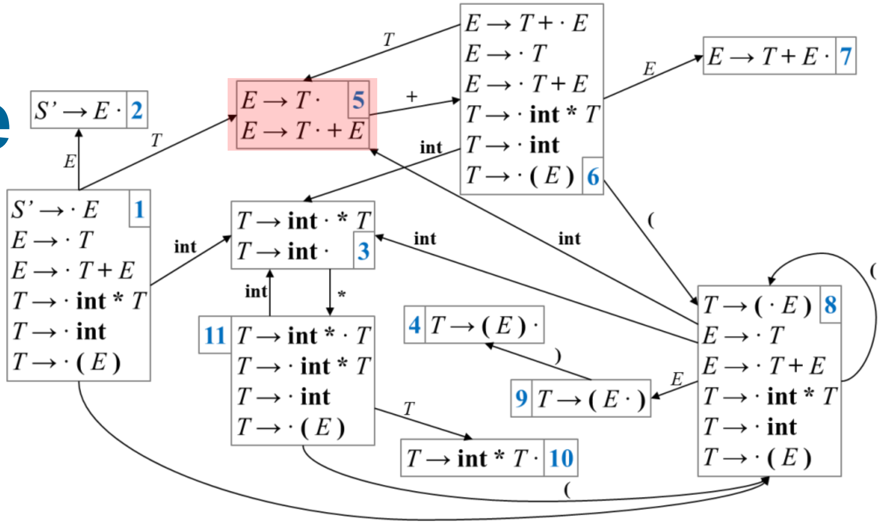
Configuration	DFA	Action	
| **int** * **int** $	1	shift	
int | * **int** $	3	shift	* not in FOLLOW(*T*)
int * | **int** $	11	shift	
int * **int** | $	3		

SLR(1) Parsing Example



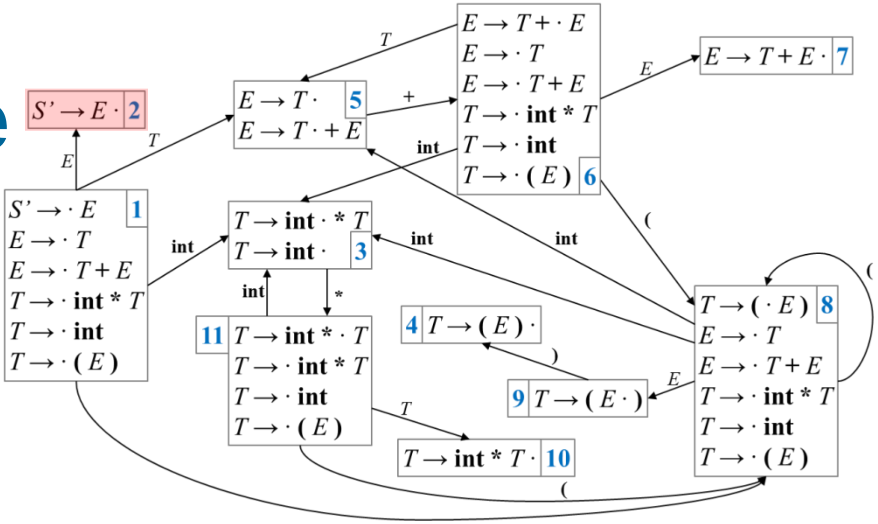
Configuration	DFA	Action	
| **int** * **int** $	1	shift	
int | * **int** $	3	shift	* not in FOLLOW(T)
int * | **int** $	11	shift	
int * **int** | $	3	reduce $T \to$ **int**	$ in FOLLOW(T)
int * T | $	10		

SLR(1) Parsing Example



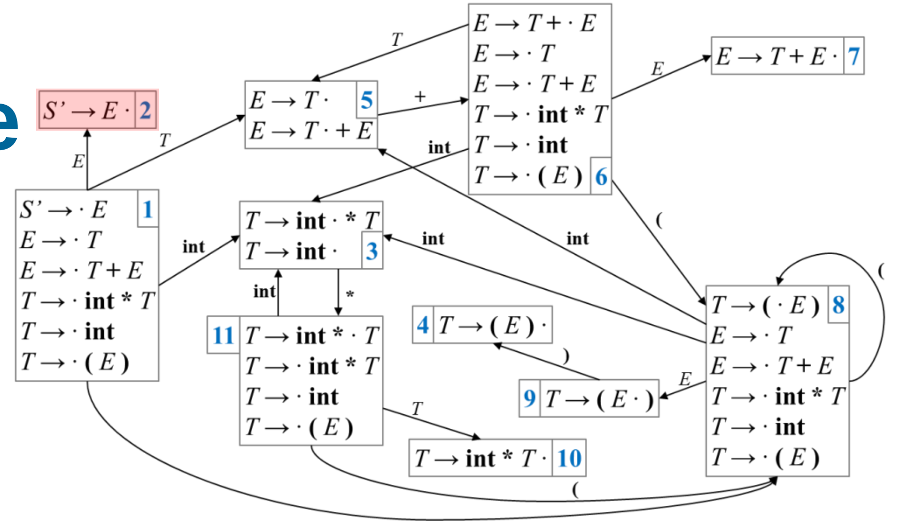
Configuration	DFA	Action	
| **int** * **int** $	1	shift	
int | * **int** $	3	shift	* not in FOLLOW(*T*)
int * | **int** $	11	shift	
int * **int** | $	3	reduce *T* → **int**	$ in FOLLOW(*T*)
int * *T* | $	10	reduce *T* → **int** * *T*	$ in FOLLOW(*T*)
T | $	5		

SLR(1) Parsing Example



Configuration	DFA	Action	
| **int** * **int** $	1	shift	
int | * **int** $	3	shift	* not in FOLLOW(*T*)
int * | **int** $	11	shift	
int * **int** | $	3	reduce *T* → **int**	$ in FOLLOW(*T*)
int * *T* | $	10	reduce *T* → **int** * *T*	$ in FOLLOW(*T*)
T | $	5	reduce *E* → *T*	$ in FOLLOW(E)
E | $	2		

SLR(1) Parsing Example



Configuration	DFA	Action	
| **int** * **int** $	1	shift	
int | * **int** $	3	shift	* not in FOLLOW(*T*)
int * | **int** $	11	shift	
int * **int** | $	3	reduce *T* → **int**	$ in FOLLOW(*T*)
int * *T* | $	10	reduce *T* → **int** * *T*	$ in FOLLOW(*T*)
T | $	5	reduce *E* → *T*	$ in FOLLOW(*E*)
E | $	2	reduce *S'* → *E*	$ in FOLLOW(*S'*)
S' | $		accept	

Efficient SLR Parsing

- Up to now, the SLR(1) Parser parsed the stack at every step
  - not necessary; only the top of the stack changes
  - idea: remember the state of the DFA on each stack prefix
  - augmented stack:

    <symbol, DFA state>

  - stack contents
    - start state: <any, start>
    - final state of DFA on input $X_1 X_2 \ldots X_n$ is $\mathrm{s_n}$
      - $<any, start>, <X_1, s_1>, <X_2, s_2>, \ldots, <X_n, s_n>$

- We need two tables, **goto** and **action** to implement efficient SLR parsers

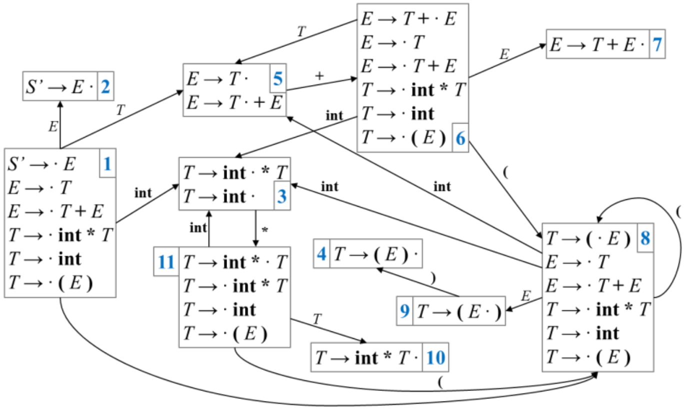
Efficient SLR Parsing: GOTO table

- Define

  `goto[i, A] = j` if $\text{state}_i \xrightarrow{A} \text{state}_j$

- `goto` is identical to the transition function of the LR(0) DFG for non-terminals

state	E	T
1	2	5
2		
3		
4		
5		
6	7	5
7		
8	9	
9		
10		
11		10



Efficient SLR Parsing: Four Actions

- Shift s
  - push <$\mathbf{t}$, s> onto the stack

- Reduce
  - same as before
  - $X \rightarrow \beta \cdot \in s$ and $\mathbf{t} \in \mathrm{FOLLOW}(X)$

- Accept

- Error

Efficient SLR Parsing: ACTION table

- For each state s_i and next input symbol (terminal $\mathbf{t}$)
  - if s_i has item $X \rightarrow \alpha \cdot \mathbf{t}\, \beta$ and `goto[i, t] = j` then

    `action[i, t] = shift j`

  - if s_i has item $X \rightarrow \alpha \cdot$ and $\mathbf{t} \in \mathrm{FOLLOW}(X)$ and $X \neq S'$ then

    `action[i, t] = reduce` $X \rightarrow \alpha$

  - if s_i has item $S' \rightarrow S \cdot$ then

    `action[i, $] = accept`

  - otherwise

    `action[i, t] = error`

Efficient SLR Parsing Algorithm

```
I = w$                                    I[j] = j-th element in input stream
j = 0
DFA state 1 has item S'→.S
stack = < dummy, 1 >
repeat
  switch (action[top(stack), I[j]]) {
    shift k:
      push < I[j++], k >

    reduce X → A:
      pop |A| elements
      push < X, goto[top(stack), X] >

    accept:
      stop, accept

    error:
      stop, report error
```

Running Example

- **Grammar**

$S \rightarrow L = R \mid R$
$L \rightarrow *R \mid \mathbf{id}$
$R \rightarrow L$

Augmented Grammar

$S' \rightarrow S$
$S \rightarrow L = R \mid R$
$L \rightarrow *R \mid \mathbf{id}$
$R \rightarrow L$

Production #

(1) $S \rightarrow L = R$
(2) $S \rightarrow R$
(3) $L \rightarrow *R$
(4) $L \rightarrow \mathbf{id}$
(5) $R \rightarrow L$

- **FIRST and FOLLOW sets**

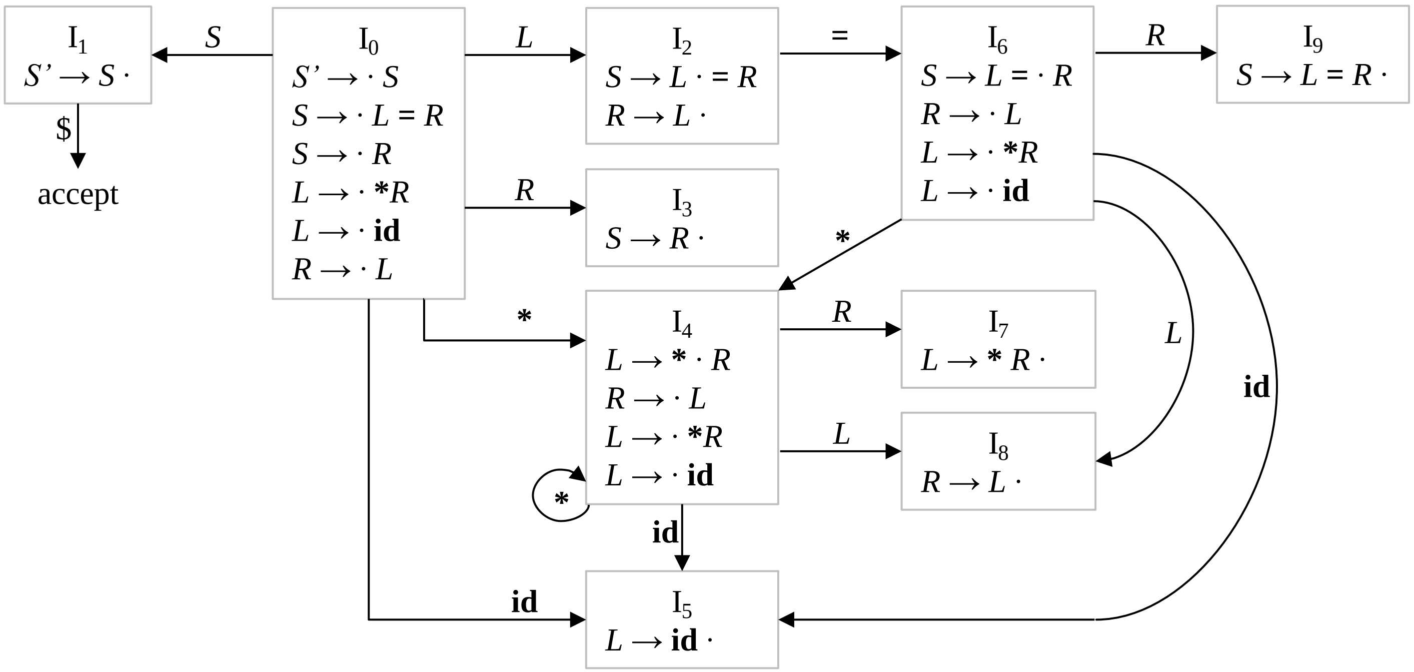
FIRST(**=**) = { **=** }
FIRST(*****) = { ***** }
FIRST(**id**) = { **id** }

FIRST($S'/S/L/R$) = { *****, **id** }

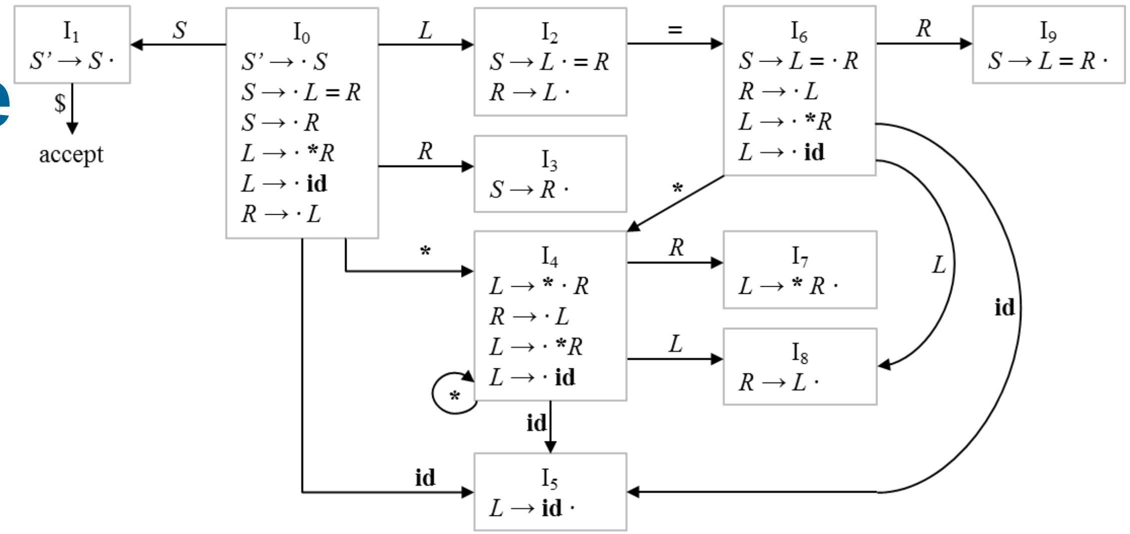
FOLLOW(S') = { \$ }
FOLLOW(S) = { \$ }
FOLLOW(L) = { **=**, \$ }
FOLLOW(R) = { **=**, \$ }
FOLLOW(**=**) = { *****, **id** }
FOLLOW(*****) = { *****, **id** }
FOLLOW(**id**) = { **=**, \$ }

Running Example

- **LR(0) Automaton**



Running Example



- **SLR Parsing Table**

	ACTION				GOTO		
	=	*	**id**	$	S	L	R
0		s4	s5		1	2	3
1				accept			
2	s6/r5 *			r5			
3				r2			
4		s4	s5			8	7
5	r4			r4			
6		s4	s5			8	9
7	r3			r3			
8	r5			r5			
9				r1			

(1) $S \to L = R$
(2) $S \to R$
(3) $L \to *R$
(4) $L \to \mathbf{id}$
(5) $R \to L$

FOLLOW(S') = { $ }
FOLLOW(S) = { $ }
FOLLOW(L) = { =, $ }
FOLLOW(R) = { =, $ }
FOLLOW(=) = { *, **id** }
FOLLOW(*) = { *, **id** }
FOLLOW(**id**) = { =, $ }

* shift/reduce conflict

Canonical LR and LALR Parsing

Limitations of SLR

- For SLR Parsers
  - condition for reduction by $X \rightarrow \beta$:
    - s contains the item $X \rightarrow \beta \cdot$ and $\mathbf{t} \in \mathrm{FOLLOW}(X)$
  - for some situations, this reduction may be invalid, i.e., even though the grammar is not ambiguous we get a shift/reduce or reduce/reduce conflict
    - the grammar in the SLR example on p. 25-27 is not ambiguous yet has a shit/reduce conflict in state 2 on input “=”

$$S \rightarrow L = R \mid R$$

$$L \rightarrow *R \mid \mathbf{id}$$

$$R \rightarrow L$$

	ACTION				GOTO		
	=	*	id	$	S	L	R
0		s4	s5		1	2	3
1				accept			
2	s6/r5			r5			
3				r2			
4		s4	s5			8	7
5	r4			r4			
6		s4	s5			8	9
7	r3			r3			
8	r5			r5			
9				r1			

- SRL does not remember enough left context to take the correct action

Canonical LR(1) Items

- Incorporate extra information
  - idea: split states when necessary to have each state indicate exactly which input symbols can follow a handle β for which there is a possible reduction $X \rightarrow \beta$
  - LR(1) items: augmented handles by terminal symbols (or $)
    - $X \rightarrow \alpha \cdot \beta \quad \Longrightarrow \quad [\, X \rightarrow \alpha \cdot \beta, \mathbf{t} \,]$
      - "1": length of lookahead item (for LR(2) include two terminals, …)
      - $\mathbf{t}$: lookahead of item $X \rightarrow \alpha \cdot \beta$
    - no meaning if $\beta \neq \varepsilon$
    - for handles of the form $[\, X \rightarrow \alpha \cdot, \mathbf{t} \,]$ ($[\, X \rightarrow \alpha \cdot \beta, \mathbf{t} \,]$ with $\beta = \varepsilon$)
      - reduce only if the next input symbol is $\mathbf{t}$
      - the set of $\mathbf{t}$'s will always be a subset of $\mathrm{FOLLOW}(X)$, (possibly a proper subset)

Canonical LR(1) Items

- An LR(1) item $[\ X \rightarrow \alpha \cdot \beta, \mathbf{t}\ ]$ is valid for a viable prefix γ if there is a derivation

$$S \overset{*}{\underset{rm}{\Rightarrow}} \delta X \omega \underset{rm}{\Rightarrow} \delta \alpha \beta \omega$$

  where

  - $\gamma = \delta\alpha$ and
  - $\mathbf{t}$ is the first symbol of ω, or ω is ε and $\mathbf{t}$ is \$.

Canonical LR(1) Items

- Example

$$S \rightarrow BB$$
$$B \rightarrow \mathbf{a}B \mid \mathbf{b}$$

- $S \overset{*}{\underset{rm}{\Rightarrow}} \mathbf{aa}B\mathbf{ab} \underset{rm}{\Rightarrow} \mathbf{aaa}B\mathbf{ab}$

  $[\, B \rightarrow \mathbf{a} \cdot B,\ \mathbf{a} \,]$ is a valid item

- $S \overset{*}{\underset{rm}{\Rightarrow}} B\mathbf{a}B \underset{rm}{\Rightarrow} B\mathbf{aa}B$

  $[\, B \rightarrow \mathbf{a} \cdot B,\ \$ \,]$ is a valid item

Construction of the sets of LR(1) Items

- Similar to construction of LR(0) items with slightly modified GOTO/CLOSURE

```
set<Item> CLOSURE(set<Item> I) {
  repeat {
    for each [A → α·Bβ, a] in I do
      for each production B → γ in G′ do
        for each terminal b in FIRST(βa) do
          I = I + {[B → · γ, b] }
  } until I doesn't change

  return I
}
```

  - observation (or "why **b** must be in FIRST(β**a**)")
    with $[A \rightarrow \alpha \cdot B\beta, \mathbf{a}]$ for some viable prefix γ, there exists $S \underset{rm}{\overset{*}{\Rightarrow}} \delta A\mathbf{a}x \underset{rm}{\Rightarrow} \delta\alpha B\beta\mathbf{a}x$ with $\gamma = \delta\alpha$. Assume $\beta\mathbf{a}x$ derives $\mathbf{b}y$, then for each production $B \rightarrow \eta$, there is a derivation $S \underset{rm}{\overset{*}{\Rightarrow}} \gamma B\mathbf{b}y \underset{rm}{\Rightarrow} \gamma\eta\mathbf{b}y$. Therefore, $[B \rightarrow \cdot\eta, \mathbf{b}]$ is valid for γ. If β derives ε then $\mathbf{b} = \mathbf{a}$.

Construction of the sets of LR(1) Items

- Similar to construction of LR(0) items with slightly modified GOTO/CLOSURE

```
set<Item> GOTO(set<Item> I, symbol B) {
  J = {}
  for each [X → α·Bβ, t] in I do
    J = J + {[X → αB·β, t]}

  return CLOSURE(J)
}

void LR1Items(augmented grammar G') {
  C = CLOSURE({[S' → ·S, $]})
  repeat {
    for each set of items I in C do
      for each grammar symbol X do
        if (GOTO(I, X) is not empty and not in C) then
          C = C + GOTO(I, X)
  } until no new sets are added to C
}
```

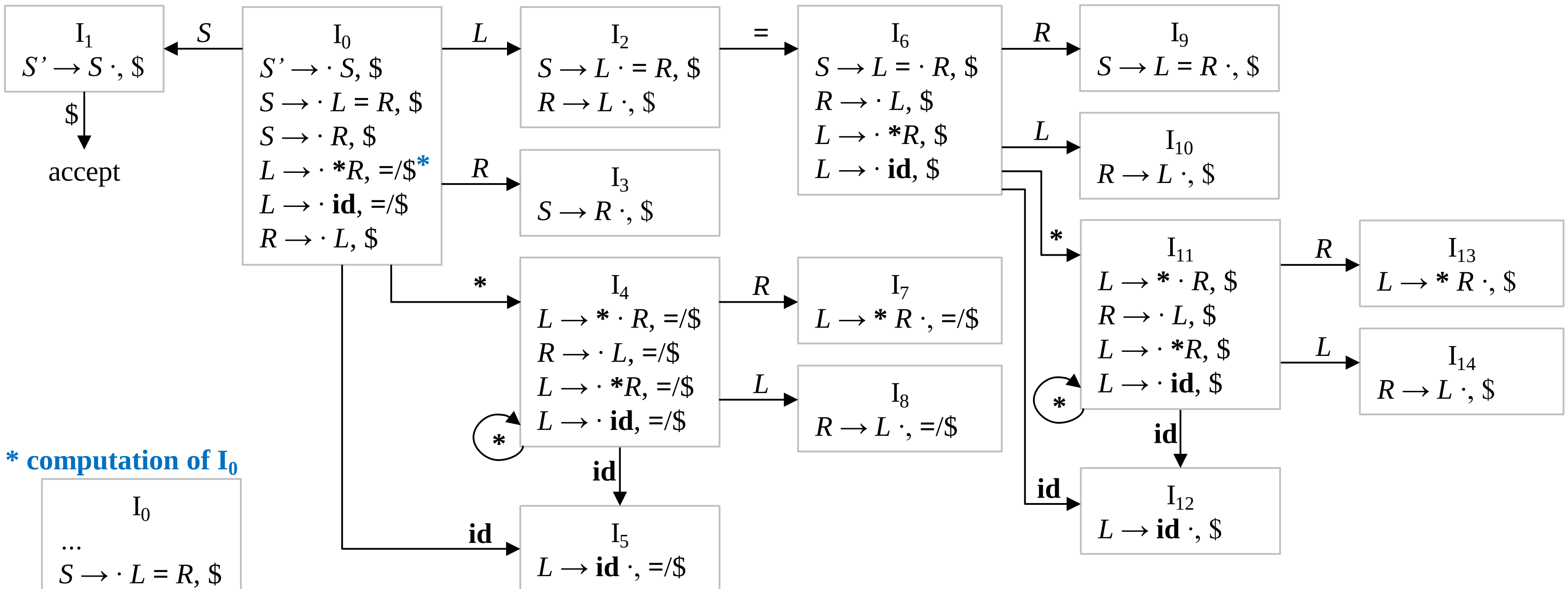
Canonical LR(1) Parsing Tables

- Input: augmented grammar G'
- Output: canonical-LR parsing tables ACTION/GOTO for G'
- Method
  1. construct $C' = \{ I_0, I_1, \ldots, I_n \}$, the collection of sets of LR(1) items for G'
  2. set I_i produces state i of the parser
     - if $[A \rightarrow \alpha \cdot \mathbf{a} \beta, \mathbf{b}]$ is in I_i and $\mathrm{GOTO}(I_i, \mathbf{a}) = I_j$, then $\mathrm{ACTION}[i, \mathbf{a}] = \mathrm{shift}\ j$.
     - if $[A \rightarrow \alpha \cdot, \mathbf{a}]$ is in I_i and $A \neq S'$, then $\mathrm{ACTION}[i, \mathbf{a}] = \mathrm{reduce}\ A \rightarrow \alpha$.
     - if $[S' \rightarrow S \cdot, \mathbf{\$}]$ is in I_i, then $\mathrm{ACTION}[i, \mathbf{\$}] = \mathrm{accept}$.
  3. construct the goto transitions for all nonterminals A as follows
     if $\mathrm{GOTO}(I_i, A) = I_j$, then $\mathrm{GOTO}[i, A] = j$.
  4. all empty entries are set to error.
  5. initial state of the parser: I_i containing $[S' \rightarrow \cdot S, \mathbf{\$}]$.

Running Example

$$S \rightarrow L = R \mid R$$
$$L \rightarrow {}^*R \mid \mathbf{id}$$
$$R \rightarrow L$$

- **LR(1) Automaton**



*** computation of I_0**

I_0

…

$S \rightarrow \cdot L = R$, \$

causes

$L \rightarrow \cdot {}^*R$, =

$L \rightarrow \cdot$ **id**, =

…

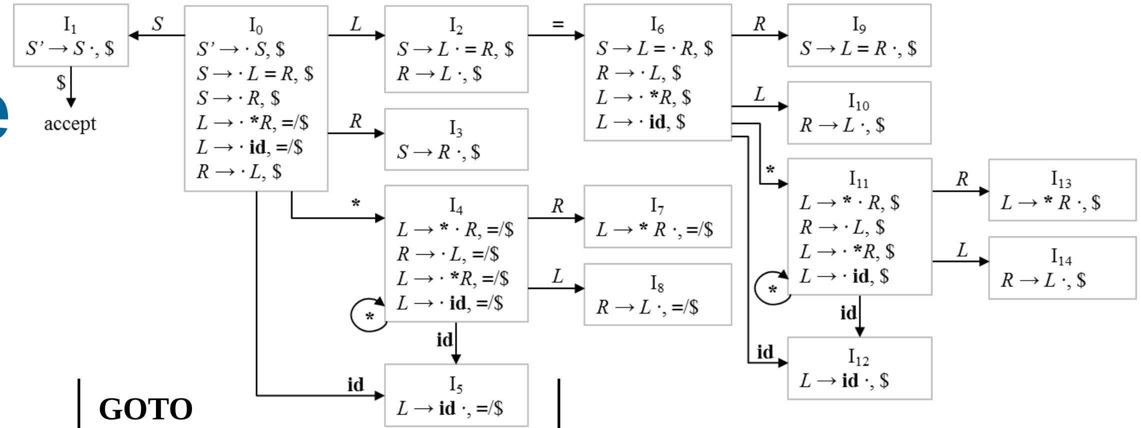
$R \rightarrow \cdot L$, \$

causes

$L \rightarrow \cdot {}^*R$, \$

$L \rightarrow \cdot$ **id**, \$

Running Example



- **LR(1) Parsing Table**

	ACTION				GOTO		
	=	*	id	$	S	L	R
0		s4	s5		1	2	3
1				accept			
2	s6			r5			
3				r2			
4		s4	s5			8	7
5	r4			r4			
6		s11	s12			10	9
7	r3			r3			
8	r5			r5			
9				r1			
10				r5			
11		s11	s12			14	13
12				r4			
13				r3			
14				r5			

(1) $S \rightarrow L = R$
(2) $S \rightarrow R$
(3) $L \rightarrow *R$
(4) $L \rightarrow \mathbf{id}$
(5) $R \rightarrow L$

no conflict anymore!

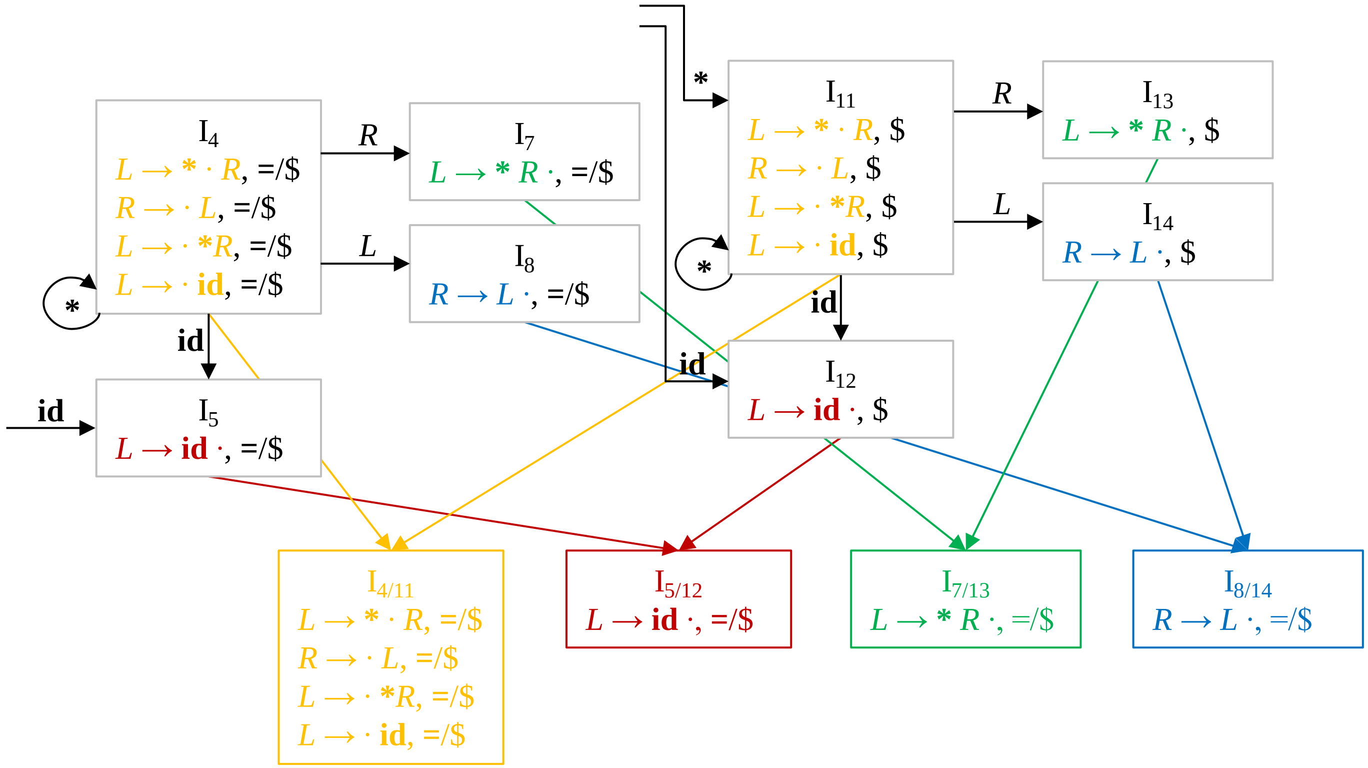
Lookahead-LR Parsing

- Canonical LR(1) parsers can require (considerably) more states than SLR parsers
  - our running example
    - SLR: 10 states
    - LR(1): 15 states
  - for a typical programming language such as C
    - SLR: few hundred states
    - LR(1): several thousand states

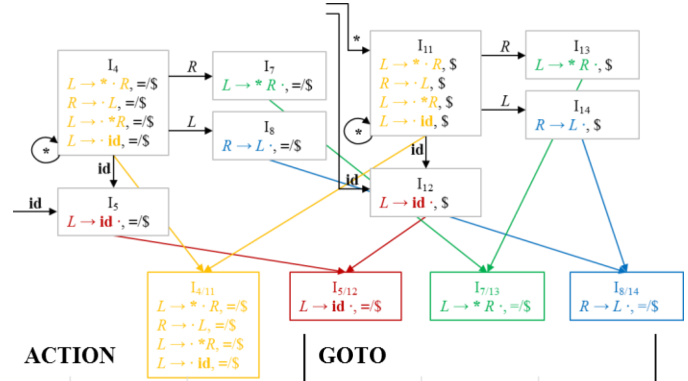
- Lookahead-LR (LALR) parsing
  - combine functionally equivalent states of an LR(1) automaton
  - same number of states as an LR(0) automaton (SRL) but can avoid the artificial shift/reduce conflicts in SRL

LALR Parsing

- LALR(1) automaton
  - combine states having the same core from the LR(1) items
  - the *core* of an LR(1) item is its LR(0) item, i.e., the core of the LR(1) item $[A \rightarrow \alpha{\cdot}\mathbf{a}\beta, \mathbf{b}]$ is simply $[A \rightarrow \alpha{\cdot}\mathbf{a}\beta]$
  - combined lookahead: union of the individual states' lookaheads



LALR Parsing



- LALR(1) parsing table
  - the GOTO of an LR(1) item set only depends on the core → the GOTOs of merged sets can themselves be merged
  - ACTIONs are modified to reflect the new sets

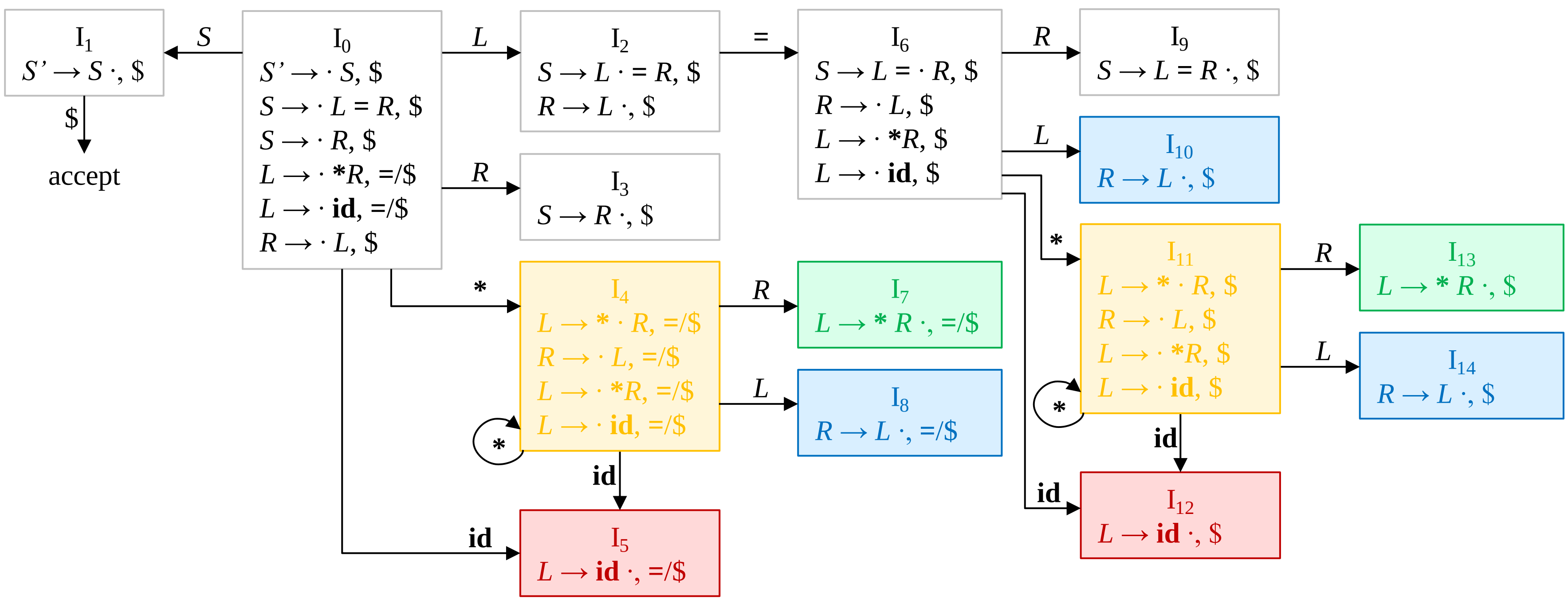
	ACTION				GOTO		
	=	*	id	$	S	L	R
4		s4	s5			8	7
5	r4			r4			
6		s11	s12			10	9
7	r3			r3			
8	r5			r5			
9				r1			
10				r5			
11		s11	s12			14	13
12				r4			
13				r3			
14				r5			

	ACTION				GOTO		
	=	*	id	$	S	L	R
0		s411	s512		1	2	3
411		s411	s512			814	713
512	r4			r4			
6		s411	s512			10	9
713	r3			r3			
814	r5			r5			

Running Example

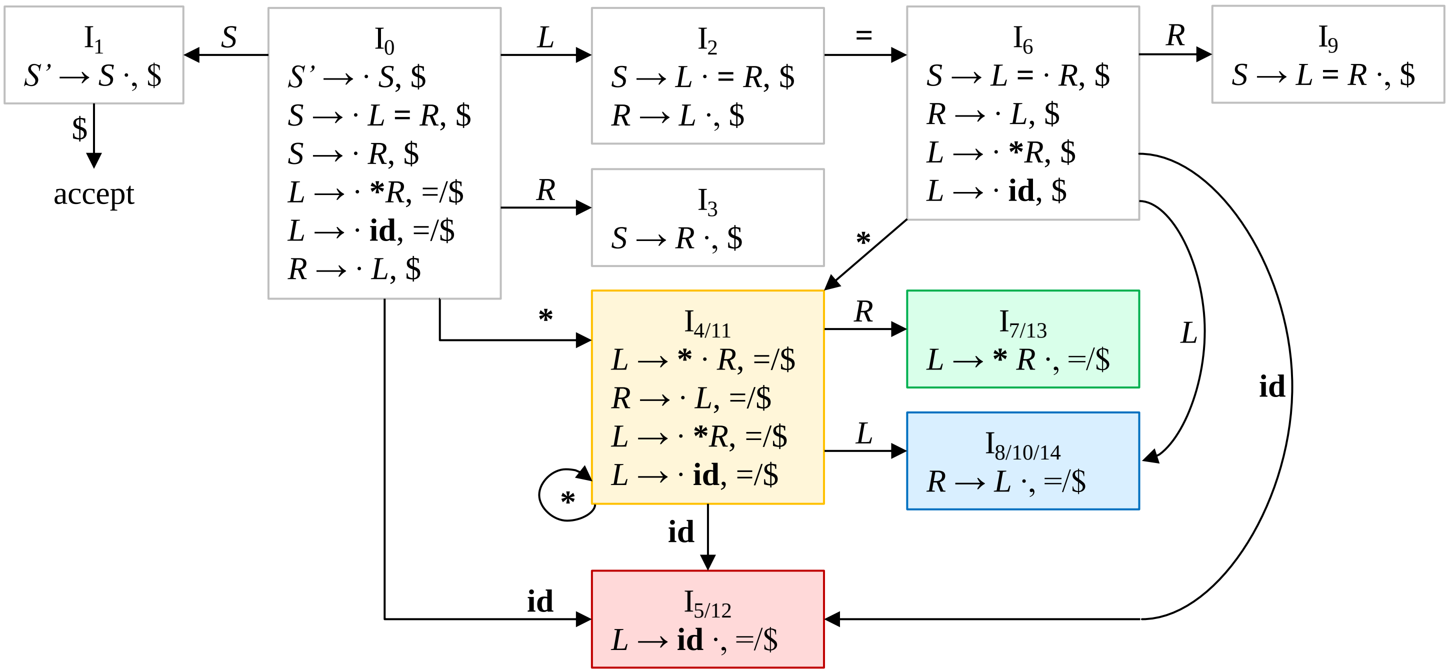
$$S \rightarrow L = R \mid R$$
$$L \rightarrow *R \mid \mathbf{id}$$
$$R \rightarrow L$$

- **LR(1) Automaton: sets with same cores**

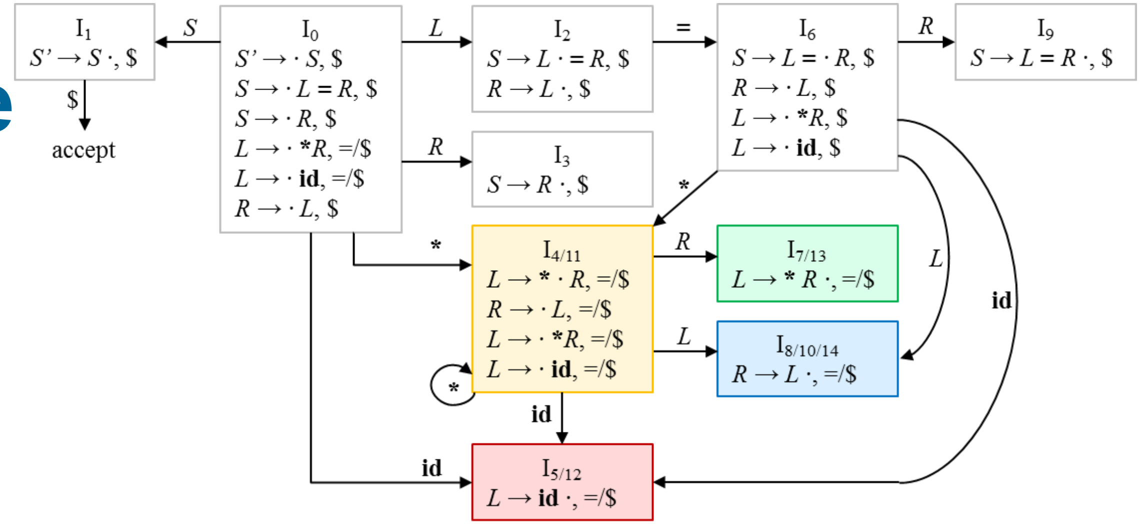


Running Example

- **LALR(1) Automaton**



Running Example



- **LALR(1) Parsing Table**

	ACTION				GOTO		
	=	*	**id**	$	*S*	*L*	*R*
0		s411	s512		1	2	3
1				accept			
2	s6			r5			
3				r2			
411		s411	s512			81014	713
512	r4			r4			
6		s411	s512			81014	9
713	r3			r3			
81014	r5			r5			
9				r1			

(1) $S \rightarrow L = R$
(2) $S \rightarrow R$
(3) $L \rightarrow *R$
(4) $L \rightarrow \textbf{id}$
(5) $R \rightarrow L$

no shift/reduce conflict

LALR Parsing

- Can merging LR(1) states with the same core lead to shift/reduce conflicts?
  - not if the conflict was not already present in the original LR(1) parsing table
  - assume the merger produces a shift/reduce conflict on lookahead $\mathbf{a}$ and items $[A \rightarrow \alpha\cdot, \mathbf{a}]$ (action: reduce) and $[B \rightarrow \beta\cdot\mathbf{a}\gamma, \mathbf{b}]$ (action: shift).
  - One of the original states must contain item $[A \rightarrow \alpha\cdot, \mathbf{a}]$. Since all core items of the merged states are identical that state also includes $[B \rightarrow \beta\cdot\mathbf{a}\gamma, \mathbf{c}]$; in other words, the conflict must have existed in the LR(1) parsing table already. This contradicts our initial assumption that no shift/reduce conflict existed.

- LALR can lead to reduce/reduce conflicts that were not present in the corresponding LR parser:

$$S \rightarrow \mathbf{a}\ A\ \mathbf{d} \mid \mathbf{a}\ B\ \mathbf{e} \mid \mathbf{b}\ B\ \mathbf{d} \mid \mathbf{b}\ A\ \mathbf{e}$$
$$A \rightarrow \mathbf{c}$$
$$B \rightarrow \mathbf{c}$$

LALR Parsing

- Generation of LALR parsing tables
  - Construction via LR(1) parsing table possible, but inefficient
  - Efficient methods to construct of LALR parsing tables exist (see textbook)

- LALR vs LR parsing actions
  - LALR may reduce more before declaring error. However, the error is caught before the next shift, i.e., at the same position in the source code.

LALR parsing table

	ACTION			
	=	*	id	$
411		s411	s512	
512	r4			r4
713	r3			r3
81014	r5			r5
9				r1

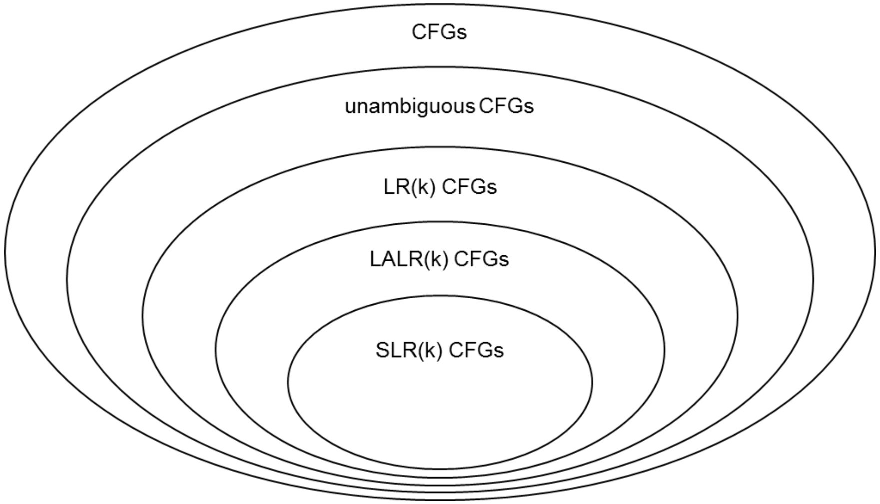
LR parsing table

	ACTION			
	=	*	id	$
4		s4	s5	
5	r4			r4
7	r3			r3
8	r5			r5
9				r1
10				r5
11		s11	s12	
12				r4
13				r3
14				r5

LR Parsing Summary

- LR(k) powerful enough to handle (almost) all programming languages, but inefficient due to large parsing tables
- SLR(k) has much fewer states, but may suffer from shift/reduce conflicts because reductions are applied too unconditionally
- LALR(k), finally, combines the best of both worlds: powerful to handle (almost) all programming languages, but only has as many states as the SLR(k) parsing table

- The corresponding grammars LR(k), LALR(k), and SLR(k) are true subsets of unambiguous context-free grammars



Parser Generators

Parser Generators

- Yacc/Bison

```
declarations
%%
translation rules
%%
supporting C code
```

Parser Generators

- A simple desktop calculator

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & \textbf{(}\ E\ \textbf{)} \mid \textbf{digit}
\end{array}
$$

Parser Generators

- Desktop Calculator: declarations

```
%{
#include <ctype.h>
%}

%token DIGIT

%%
```

Parser Generators

- Desktop Calculator: translation rules

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (\, E \,) \mid \textbf{digit}
\end{aligned}
$$

```
%%

line      : expr '\n'           { printf("%d\n", $1); }
          ;

expr      : expr '+' term       { $$ = $1 + $3; }
          | term
          ;

term      : term '*' factor     { $$ = $1 * $3; }
          | factor
          ;

factor    : '(' expr ')'        { $$ = $2; }
          | DIGIT
          ;

%%
```

Parser Generators

- Desktop Calculator: supporting C code

```
%%

#include <stdio.h>

yyerror(char *s) {
  printf("%s\n", s);
}

yylex() {
  int c;
  c = getchar();
  if (isdigit(c)) {
    yylval = c - '0';
    return DIGIT;
  }
  return c;
}

main() {
  yyparse();
}
```

Parser Generators

- Desktop Calculator

  - generating the parse tables

    ```
    $ bison deskcalc.y
    ```

    produces deskcalc.tab.c

  - compiling the executable program

    ```
    $ gcc -o deskcalc deskcalc.tab.y
    ```