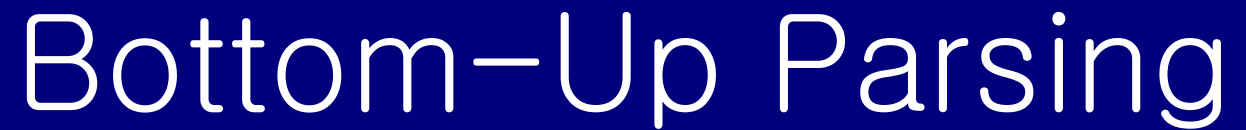


Dragon ch. 4.5 – 4.8

Bottom-Up Parsing

- Construct the parse tree from leaves
- At each step, decide on some substring that matches the RHS of some production
  - Replace this string by the LHS (called **reduction**)
- If the substring is chosen correctly at each step, it is the trace of a rightmost derivation in reverse

Handle

- A *Handle* of a string
  - A substring that matches the RHS of some production and whose reduction represents one step of a rightmost derivation in reverse
  - So we scan tokens from left to right, find the handle, and replace it by corresponding LHS
  - Problem: a leftmost substring that matches some RHS is NOT a handle

An Example of Bottom-Up Paring

- S → aABe
- A → Abc | b
- B → d

SHIFT/REDUCE Parsing

a **b** b c d e

a **A b c** d e

a A **d** e

a A B e

S

(shaded box) : handle

Rightmost Derivation

S => a A B e

=> a A d e

=> a A b c d e

=> a b b c d e

Shift-Reduce Parsing

- Bottom-up parsing is a.k.a. **shift-reduce** parsing
  - Use a stack of grammar symbols where tokens are shifted (i.e., pushed)
  - Perform table-driven shift/reduce actions
  - Shift tokens onto the stack until the handle shows up at the top of stack which is then reduced into the LHS
    - Handle always occurs at the stack top, never in the middle

Actions of Shift-Reduce Parsing

- Basic Operations
  - SHIFT: push the next token onto the stack
  - REDUCE: replace RHS on stack top of some production by its LHS (nonterminal)
  - ACCEPT: reduction to the start nonterminal
    - Assume a unique S production
    - If not, augment the grammar S'→ S
- In each step we must choose
  - SHIFT or REDUCE
  - If REDUCE, which production?

Let's first try our lucky guess

Example

- Example Grammar G1: S → (S) | a

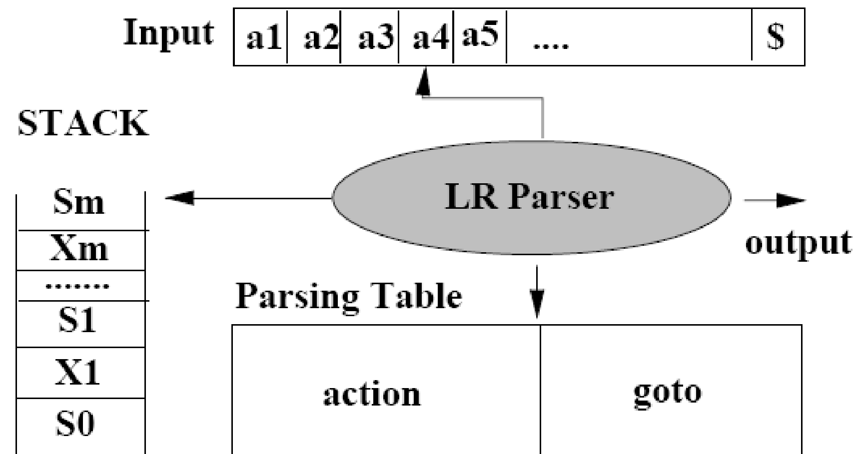
STACK	INPUT	ACTION
$	((a)) $	SHIFT
$ (	(a)) $	SHIFT
$ ((	a)) $	SHIFT
$ ((a	)) $	REDUCE S –> a
$ ((S	)) $	SHIFT
$ ((S)	) $	REDUCE S –>(S)
$ (S	) $	SHIFT
$ (S)	$	REDUCE S –>(S)
$ S	$	ACCEPT

- The REDUCE step define a rightmost derivation in reverse order

LR(k) Parsing

- LR(k): Left-to-right scan, rightmost derivation in reverse, with k-symbol lookaheads
  - LR parsing is the most general Shift/Reduce parsing
  - LR parsers are very general; parse not all CFG, but enough CFG including most programming languages
  - Can parse a superset of grammars that LL(k) parses
  - Good syntactic error detection capability
  - Good tools to construct LR parsers (YACC)

LR Parsing Data Structures



- STACK
  - Stores $s_0X_1s_1X_2s_2 \ldots X_ms_m$, where s_i is a state and X_i is a grammar symbol (i.e., terminal or nontermianl)
  - Each state summarizes the information contained in the stack below it
  - Therefore, grammar symbols need not be explicitly stored in real implementation

- **Parsing tables**:
  - Indexed by the state at the stack top and the current input symbol; composed of *action[]* & *goto[]* tables
  - *action*$[s_m, a_i]$ (s_m:state, a_i:terminal)
    - shift *s*, where *s* is a state
    - reduce by A → β
    - accept
    - error
  - *goto*$[s. X]$ (s: state, X: nonterminal)

    produces a state

Parsing Actions & Gotos

- A configuration of an LR parser is a pair ($s_0X_1s_1 \ldots X_ms_m$, $a_ia_{i+1}a_{i+2} \ldots a_n\$$) which represents a right−sentential form (RSF), $X_1X_2 \ldots X_ma_ia_{i+1}a_{i+2} \ldots a_n$
  - Initial configuration: (s_0, $a_0a_1a_2 \ldots a_n\$$)
- The next move of the LR parser depends on s_m and a_i
  - If *$action[s_m,a_i]$* = **shift** *s*, then shift: ($s_0X_1s_1 \ldots X_ms_ma_is$, $a_{i+1}a_{i+2} \ldots a_n\$$)
  - If *$action[s_m,a_i]$* = **reduce** $A \rightarrow \beta$, then reduce: ($s_0Xs_1 \ldots X_{m-r}s_{m-r}As$, $a_ia_{i+1}a_{i+2} \ldots a_n\$$) where s = *$goto[s_{m-r},A]$* and r is the length of β
  - If *$action[s_m,a_i]$* = **accept**, accept
  - If *$action[s_m,a_i]$* = **error**, call error recovery

Example of an LR Parsing

(1) E -> E + T

(2) E -> T

(3) T -> T * F

(4) T -> F

(5) F -> (E)

(6) F -> id

s5 means shift and stack state 5

r5 means reduce by prod. number (5)

	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

- How is *id* * *id* + *id* parsed?

How to Make the Parse Table?

- Use DFA again for building parse tables
  - Each state now summarizes how much we have seen so far and what we expect to see
    - Helps us to decide what action we need to take
- How to build the DFA, then?
  - Analyze the grammar and productions
    - Need a notation to show how much we have seen so far for a given production: LR(0) item

LR(0) Item

An **LR(0) item** is a production and a position in its RHS marked by a dot (e.g., $A \rightarrow \alpha \cdot \beta$)

- The dot tells how much of the RHS we have seen so far. For example, for a production $S \rightarrow XYZ$,
  - $S \rightarrow \cdot XYZ$: we hope to see a string derivable from XYZ
  - $S \rightarrow X \cdot YZ$: we have just seen a string derivable from X and we hope to see a string derivable from YZ

(X, Y, Z are grammar symbols)

State of LR(0) Items

- Equivalence of LR(0) items
  - If there are two productions S → XYZ and Y → WQ, then S → X·YZ and Y → ·WQ are equivalent items
    - If W → P is a production, W → · P is also equivalent
- State of equivalent LR(0) items
  - For a given LR(0) item, we can find the set of all its equivalent LR(0) items, which comprises a single state
  - If a state has S → X·YZ, make it transit to a different state that has S → XY·Z on Y and find its equivalence set
  - In this way, beginning from the start production S' → ·S, we can build a DFA of states of LR(0) items

DFA Construction Algorithm

- Build DFA from grammar by iterating two steps
  - CLOSURE: Given a “kernel” for a state (set of LR(0) items), complete the state by adding all equivalent items
  - GOTO: From a complete state, find the kernel of a successor state on a particular symbol
- Start with an LR(0) item set with the start production {S’ → ·S}

CLOSURE() Algorithm

- CLOSURE (`item_set`)

  Repeat

  If there is a "·A" in an item in **`item_set`**

  For every production A → **α**, add A → ·**α** to **`item_set`**

  Until no more changes

GOTO() Algorithm

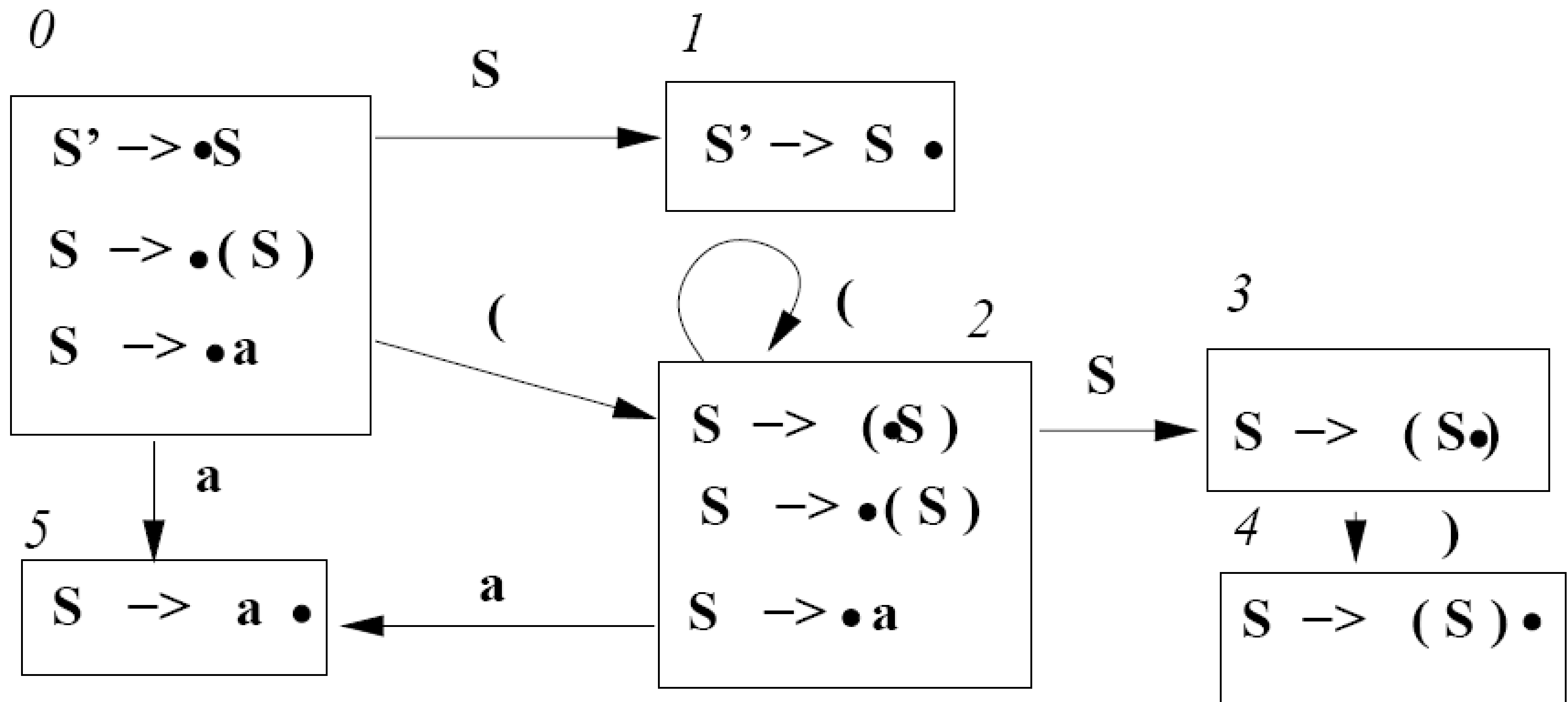
- Find the successor states of a state I
  - For every symbol X such that $A \rightarrow \alpha \cdot X\beta$ where $X \in V \cup T$, compute GOTO(I,X)
- GOTO(I,X)

```
kernel  = {};
For every item A → α·Xβ ∈ I
    add A → αX·β to kernel;
return CLOSURE(kernel);
```

- Add a transition on symbol X from state I to GOTO(I,X)
  - Note that GOTO(I,X) may have already been computed

An Example DFA for Grammar G1

G1: S → (S) | a



Classification of LR(0) Items

- Shift item
  - one that has · before a terminal (Ex: S → ·(S))
- Reduce item
  - one that has · at the end of RHS (Ex: S → a·)
- Conflict
  - When you have to choose between Shift/Reduce, or Reduce/Reduce in a state

LR(0) Grammar

- No conflict in the DFA
  - If a state has a reduce item, it has no other reduce or shift items
  - We know what to do in each state
    - Shift items only: shift
    - One reduce item only: reduce using the production
  - Unfortunately, LR(0) is a very limited grammar
    - Means many grammars produces conflict in their DFA

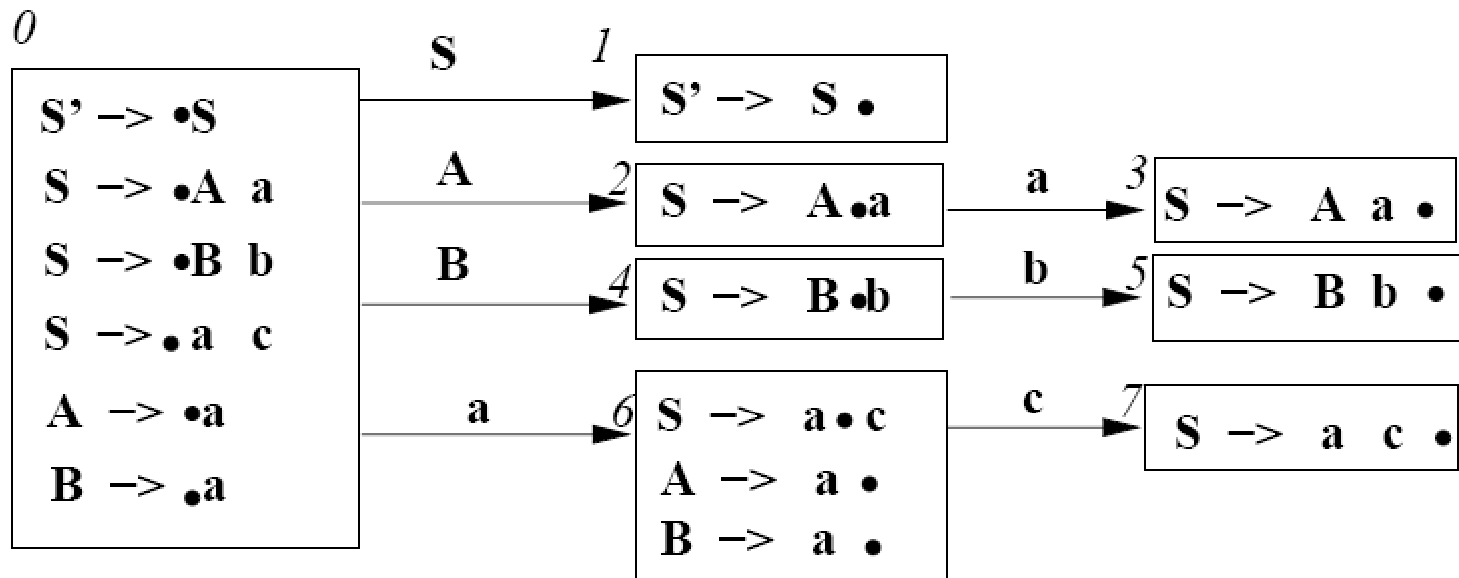
LR(0) Parsing Algorithm

- Stack
  - Keep state on stack which summarizes stack info below it
- Actions and GOTOs
  - Shift: If the next input is 'a' and there is a transition from the state on the top of stack to the state N on 'a', push N and advance input pointer
  - Reduce: If a state has a reduce item, (1) pop stack for every symbol on the RHS (2) push GOTO(top of stack, LHS)
  - Accept: if we reduce S' → S and there is no more input
  - Otherwise, ERROR and halt

SLR(1) parsing

- LR(0) is very limited, useless by itself
  - Even one symbol lookahead helps a lot!
- An Example Grammar G2 that is NOT LR(0)
  - S’→ S
  - S → Aa|Bb|ac
  - A → a
  - B → a

Corresponding DFA for G2



- Not LR(0) shift/reduce & reduce/reduce conflict in state 6

SLR(1) Parsing

- Simple LR(1) using a lookahead to resolve conflict
  - If a state has more than one reduce item or both reduce and shift items, compare the input symbol with the FOLLOW() set of the LHS of the reduce item
    - Why? If reduced correctly, stack+input will be a valid RSF
  - Ex: FOLLOW(S)={$}, FOLLOW(A)={a} FOLLOW(B)={b}
- In state 6 of previous example if the lookahead is
  - ‘a’: reduce A → a
  - ‘b’: reduce B → a
  - ‘c’: shift to state 7

Constructing SLR Parse Table

- Construct the DFA (state graph) as in LR(0)
- Action Table
  - If there is a transition from i to j on a terminal 'a',
    ACTION[i, a] = shift j
  - If there is a reduce item A → α· (for a production #j) in state i, for each a ∈ FOLLOW(A),
    ACTION[i, a] = Reduce j
  - If an item S' → S is in state i,
    ACTION[i, $] = Accept
  - Otherwise, error
- GOTO
  - Write GOTO for nonterminals: for terminals it is already embedded in the action table

Example SLR Parse Table for G2

- S’→ S, S → Aa|Bb|ac, A → a, B → a

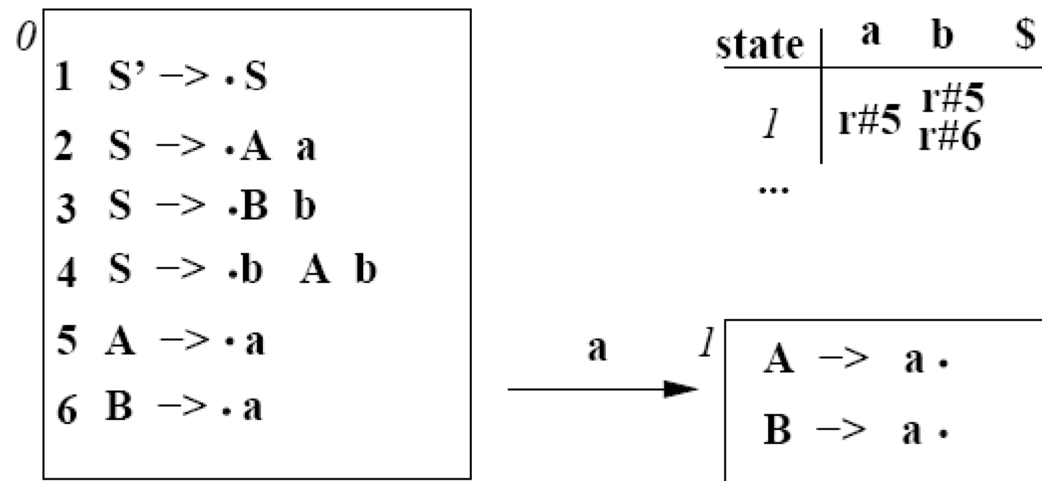
states	action				goto		
	a	b	c	$	S	A	B
0	S6				1	2	4
1				ACC			
2	S3						
3				R#1			
4		S5					
5				R#2			
6	R#4	R#5	S7				
7				R#3			

Limitations of SLR Parsing

- FOLLOW() does not always tell the truth
  - Remember similar situations in strong-LL(2)
- An Example Grammar G3 that is not SLR(1)
  - S' → S
  - S → Aa|Bb|bAb
  - A → a
  - B → a

Corresponding DFA for G3

- L(G) = {aa, ab, bab}, FOLLOW(A) = {a, b}, FOLLOW(B) = {b}



- A conflict in ACTION[1,b]. Actually, which production is right?
- In SLR(1) parsing, we reduce A → α for ANY lookahead a ∈ FOLLOW(A), which is too general such that sometimes a reduction cannot occur for some a ∈ FOLLOW(A)

(Canonical) LR(1) Parsing

- Most Powerful Parsing Technique
  - Still have one symbol lookahead, yet the use of the lookahead is more refined and detailed
  - LR items will now carry lookahead information
    - DFA of LR(1) items instead of LR(0) items
  - Has an effect of splitting some LR(0) DFA states that have reduce/reduce conflicts

LR(1) Items

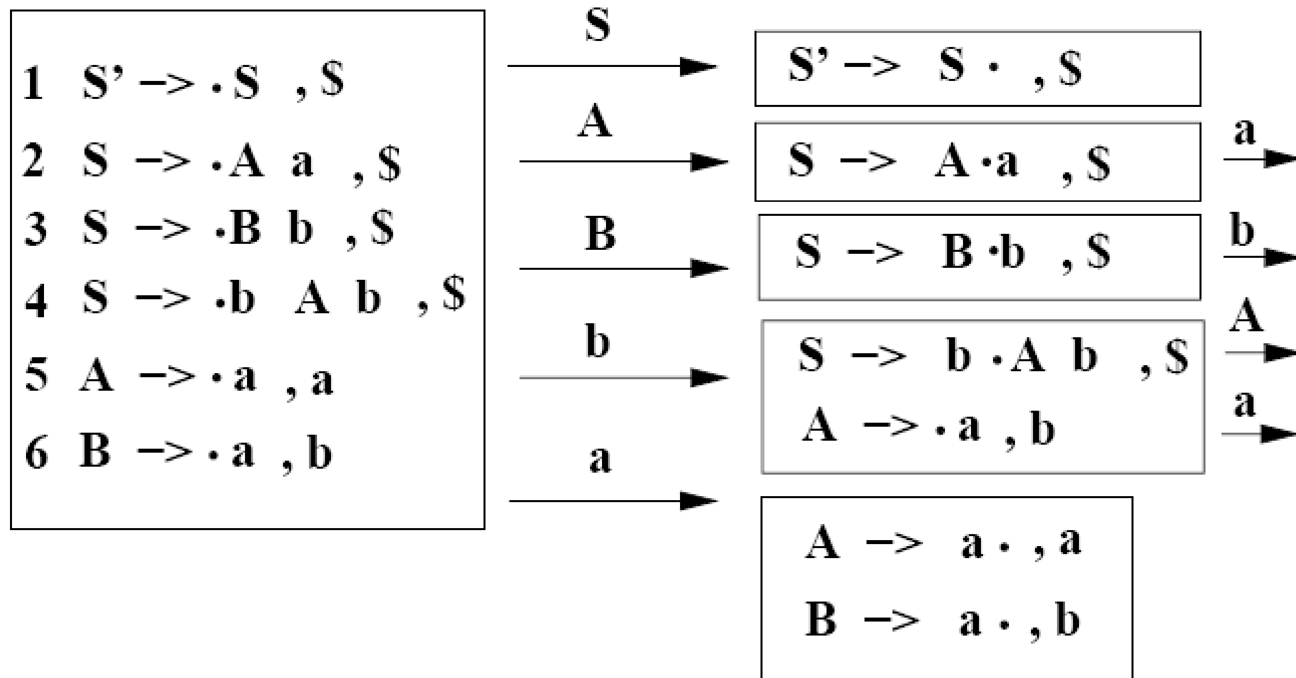
- LR(1) item has the following form [A → α·β, a], where a is a lookahead (a can be $.)
  - The lookahead is ignored unless β ∈ ε
    - i.e., it is used only for reduce items
  - A reduce item [A → α·,a] means "reduce A → α if the lookahead is a"
  - The lookahead a ∈ FOLLOW(A), but perhaps not all of FOLLOW(A) appear in the lookahead of some item whose LHS is A
  - The first LR(1) item is [S' → ·S , $]
    - "Accept" state is [S' → S· $]

DFA Construction Modification

- As before use CLOSURE() & GOTO() to unwind a DFA
- CLOSURE()
  - Whenever $[A \rightarrow \alpha \cdot B\beta, a] \in I$, add $[B \rightarrow \cdot \gamma, b]$ for all productions $B \rightarrow \gamma$ and for terminals $b \in FNE(\beta a)$
- GOTO()
  - Essentially be the same as before;
    $[A \rightarrow \alpha \cdot B\beta, a]$, then $[A \rightarrow \alpha B \cdot \beta, a]$ on B
    Lookahead carries through
- A grammar is LR(1) if there are no shift/reduce or reduce/reduce conflicts under this construction

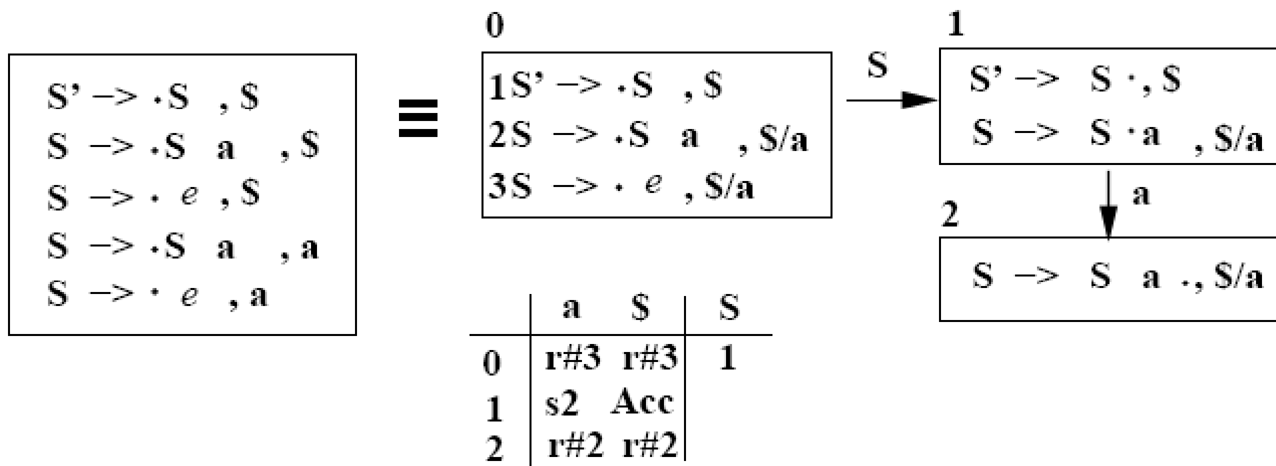
LR(1) DFA Construction for G3

- S’→ S, S → Aa|Bb|bAb, A → a, B → a



LR(1) Parsing Table

- (1) S' → S
- (2) S → Sa
- (3) S → ϵ



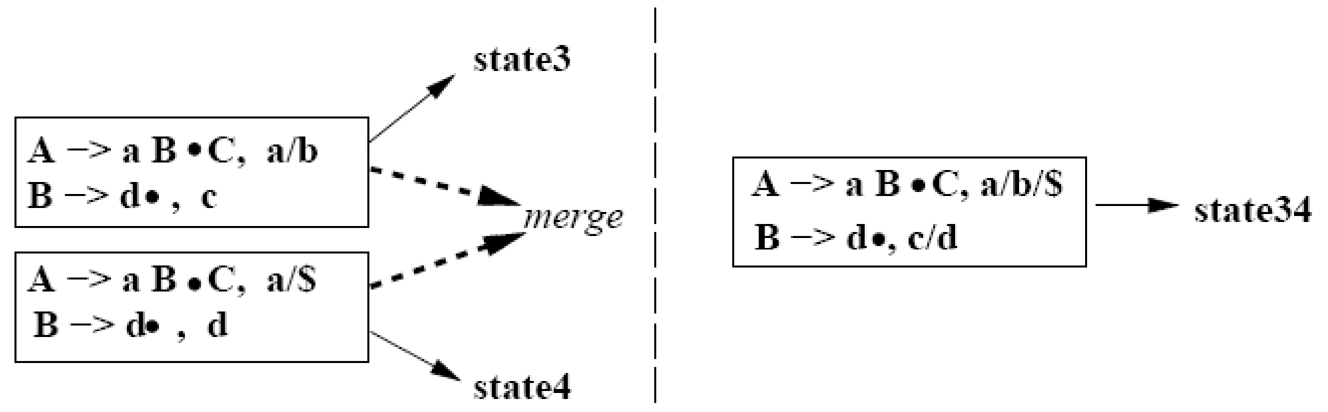
	a	$	S
0	r#3	r#3	1
1	s2	Acc	
2	r#2	r#2	

LALR(1) Parsing

- Canonical LR(1) Parsing is quite Powerful
  - However the number of states can be big
    - Big and slow parser
- **Lookahead LR(1)** (LALR(1)) Parsing
  - Number of states is greatly reduced
    - In an order of magnitude
  - Tools that generate LALR Parser: YACC

LALR(1) Parsing

- Merge states having exactly the same set of LR(0) “cores”
- Take the union of lookaheads
- Merge the GOTOs in the parsing table



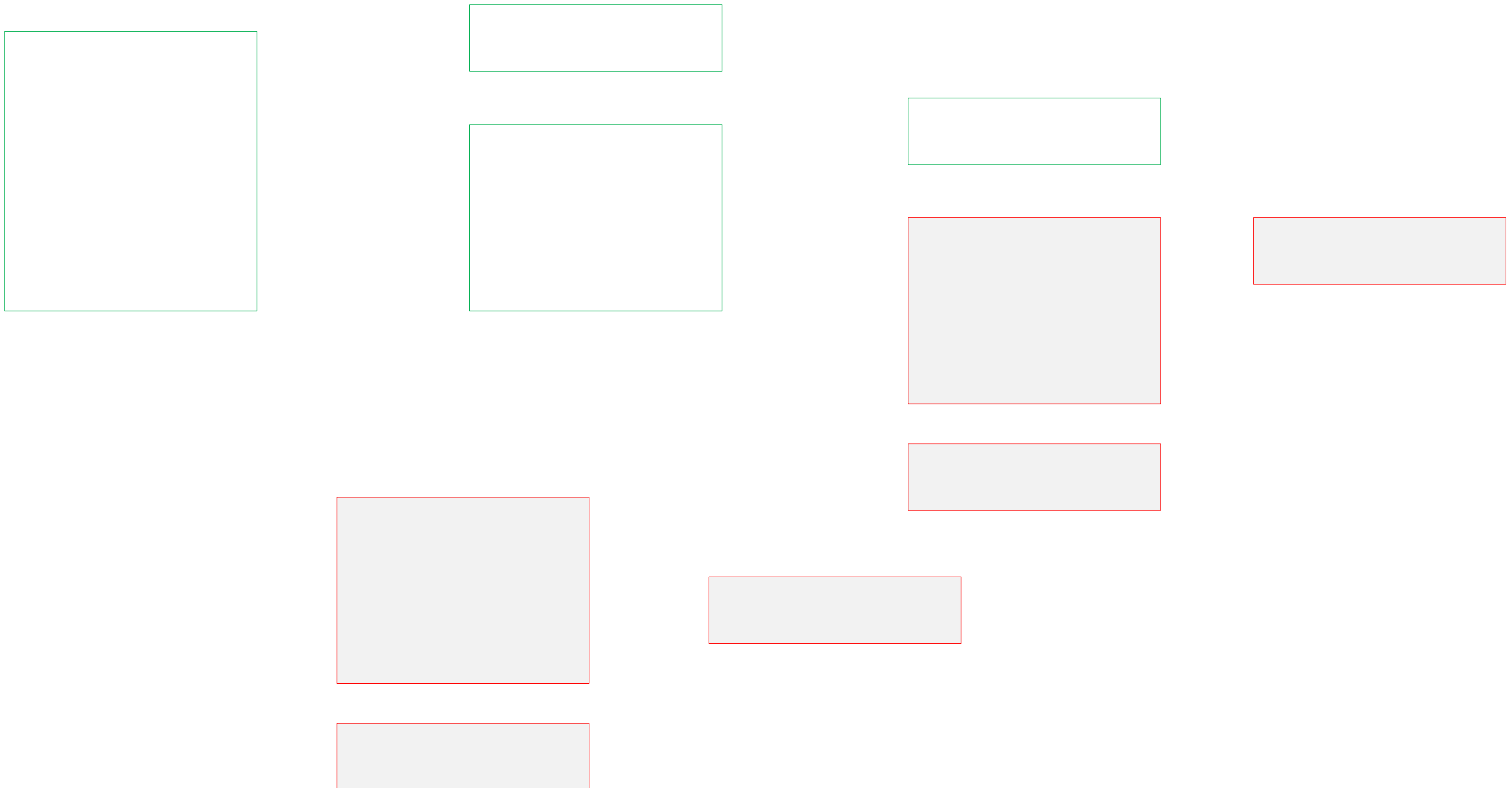
- Two issues
  - Can merged DFA parse correctly?
  - Does merging introduce any conflicts?

Correctness of a Merged DFA

- Example in the textbook P262 {c*dc*d}
  - S' → S
  - S → CC
  - C → cC
  - C → d
- How does "ccd" or "cdcdc" fail to be parsed correctly in the merged DFA?

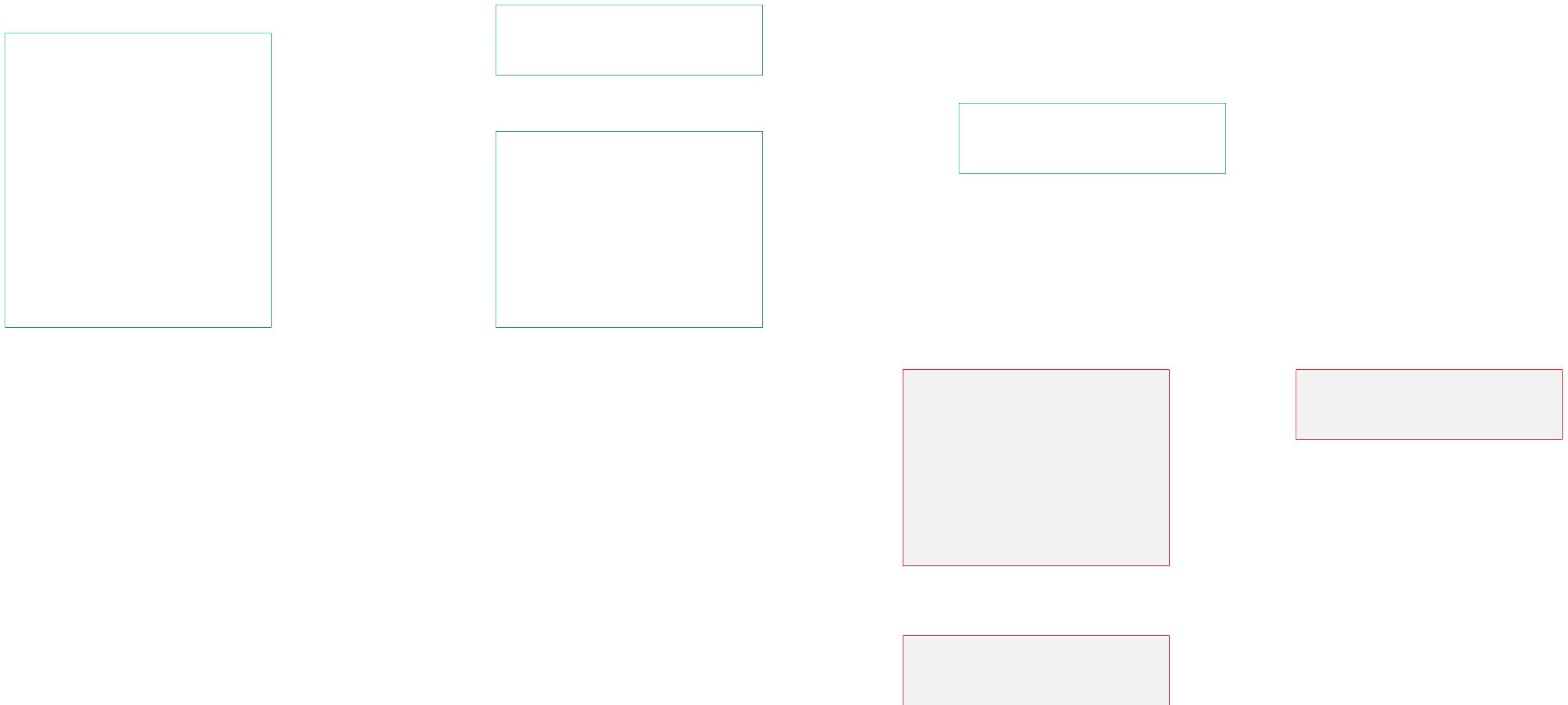
Correctness of a Merged DFA

- S' → S, S → CC, C → cC, C → d



Correctness of a Merged DFA

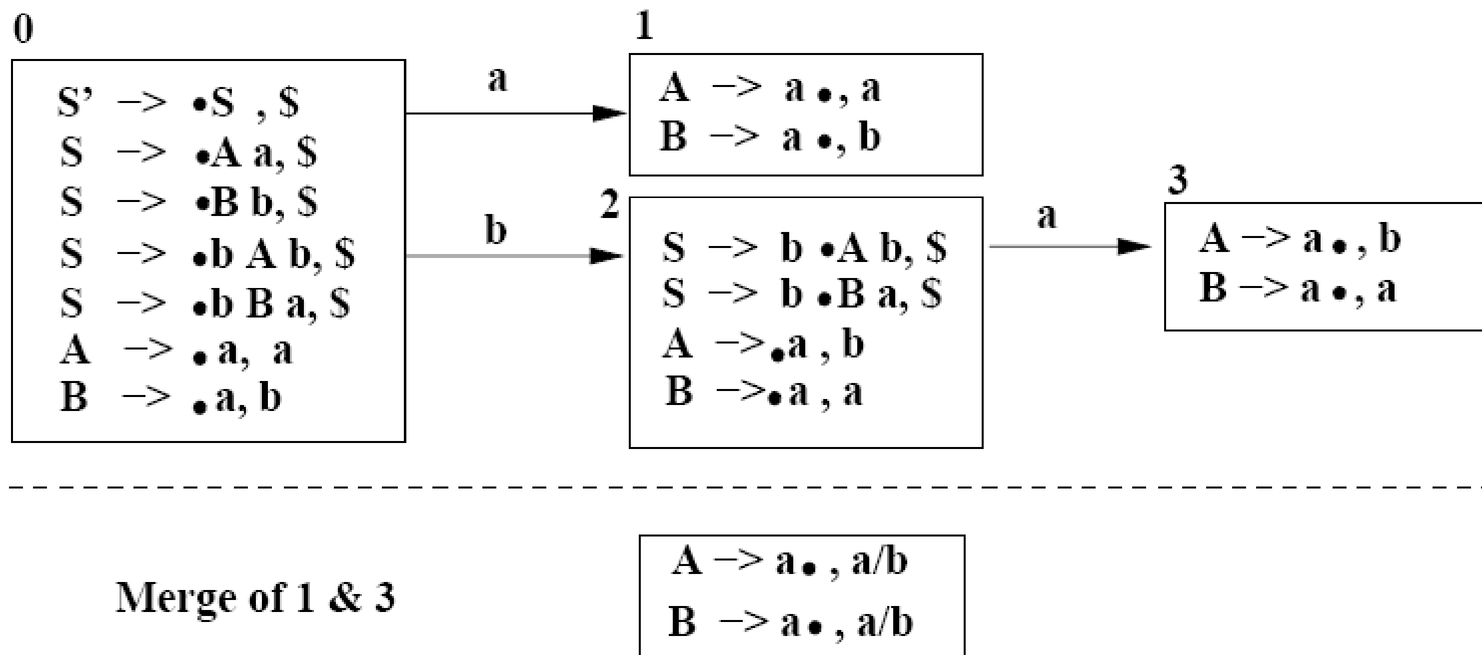
- S' → S, S → CC, C → cC, C → d



Conflicts caused by Merging?

- Merging LR(1) states might cause reduce-reduce conflicts but cannot cause shift-reduce conflicts; Why?
  - e.g., Can we have [A → **α**·, a, B → **β**·a γ ,b] after the merge?
- A grammar **G is LALR(1)** if merging implies no new conflicts
- An example of reduce-reduce conflicts after merging
  - S' → S
  - S → Aa|Bb|bAb|bBa
  - A → a
  - B → a

LALR(1) DFA



- Reduce/reduce conflicts; Not LALR(1) Grammar

Comparison of SLR(1), LR(1), LALR(1)

- SLR(1) Grammar
  - S → Aa|Bb|ac, A → a, B → a
- LALR(1) Grammar, but not SLR(1)
  - S → Aa|Bb|bAb, A → a. B → a
- LR(1), but not LALR(1)
  - S → Aa|Bb|bAb|bBa, A → a, B → a

Ambiguous Grammars

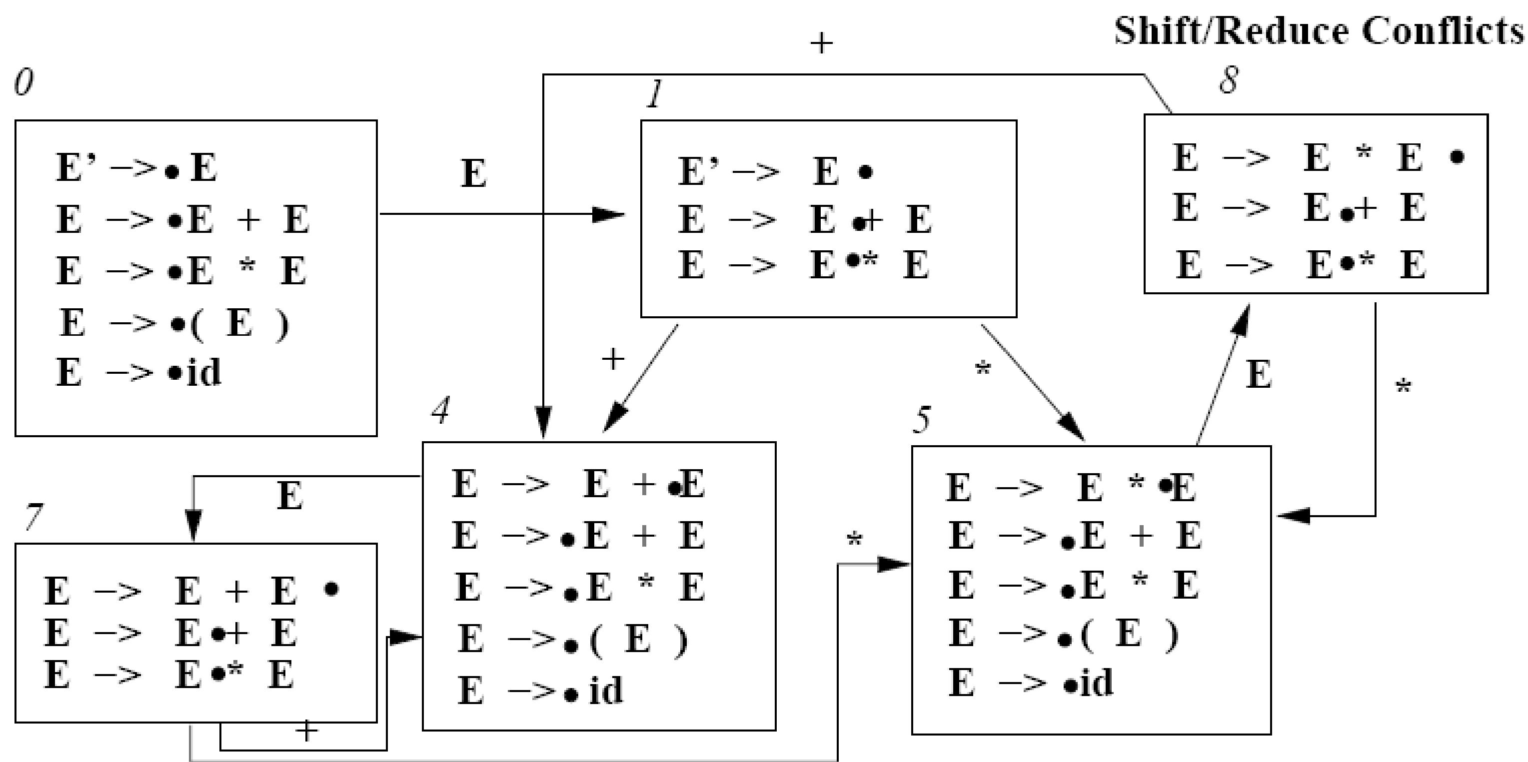
- LR parsing does not work for ambiguous grammars
  - Conflicts and two parse trees
- Why use ambiguous grammars? Advantages
  - Maybe natural (e.g., expressions) compared to unambiguous one
    $E \rightarrow E + E | E * E | (E) \mid id$ (No precedence/associativity), versus

    $E \rightarrow E + T | T$ (* has a higher precedence than +)
    $T \rightarrow T * F | F$ (*, + are left-associative)
    $F \rightarrow (E) \mid id$
- May change the precedence/associativity easily
- Smaller parse table, maybe w/o single productions ($E \rightarrow T$)

Resolving Conflicts

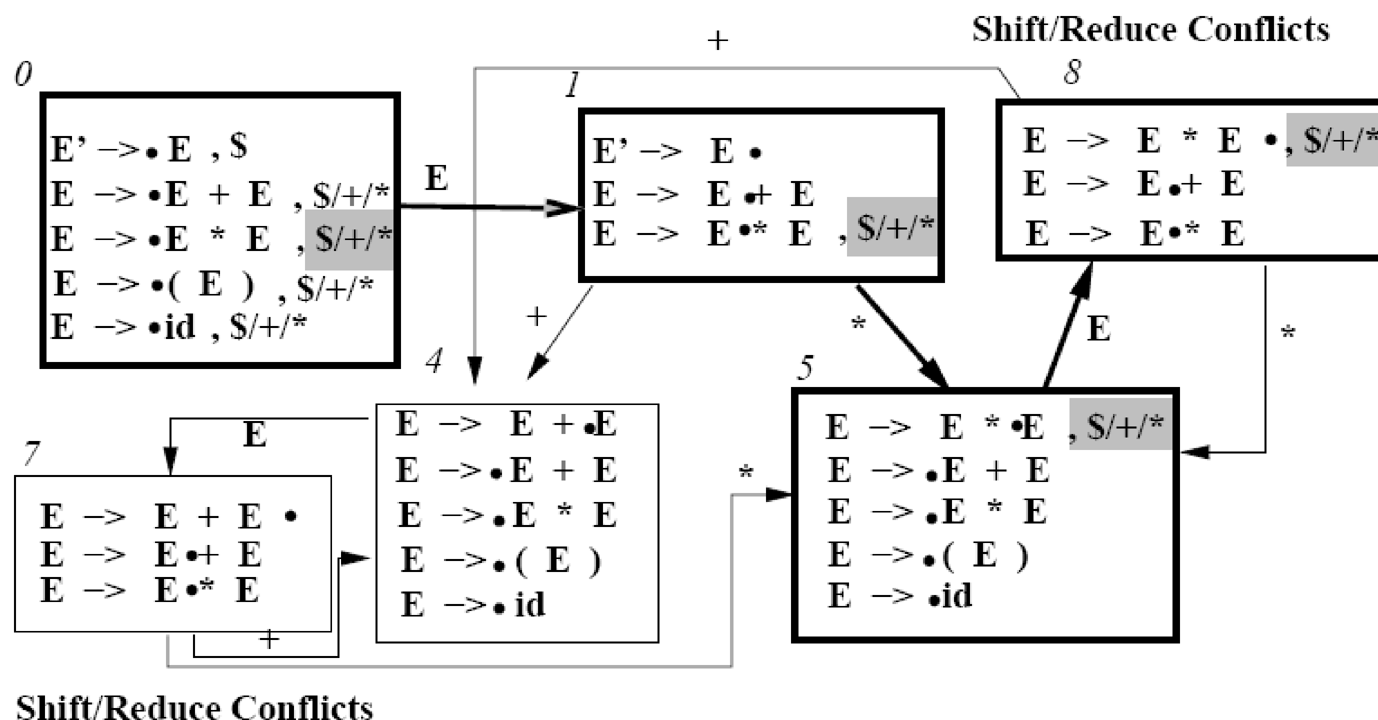
- Idea: when encounter a conflict in a parse table, apply some disambiguating rules to throw away some options
- Pitfall: May not parse the correct language
- The case in YACC
  - Shift/Reduce Conflicts: favor shifts over reduce
  - Reduce/Reduce Conflicts: reduce production that comes first in the YACC specification
- Reconsider our example ambiguous, expression grammar
  - E' → E
  - E → E + E | E * E | (E) | *id*

SLR(1) DFA



LR(1) DFA

- It should be noted that LR(1) parsing does not help at all for ambiguity resolution



Precedence/Associativity

- For disambiguating conflicts, we use precedence/associativity rules
- Precedence: since the precedence of * is higher than +,
  - Shift when * is the lookahead and + is in the left (in state 7)
  - Reduce when + is the lookahead and * is in the left (in state 8)
- Associativity: since +, * are left associative (e.g., (*id* + *id*) + *id*
  - Reduce when the operator is both in lookahead and in the left
  - If an operator is right associative, then shift

Example of Resolving Conflicts

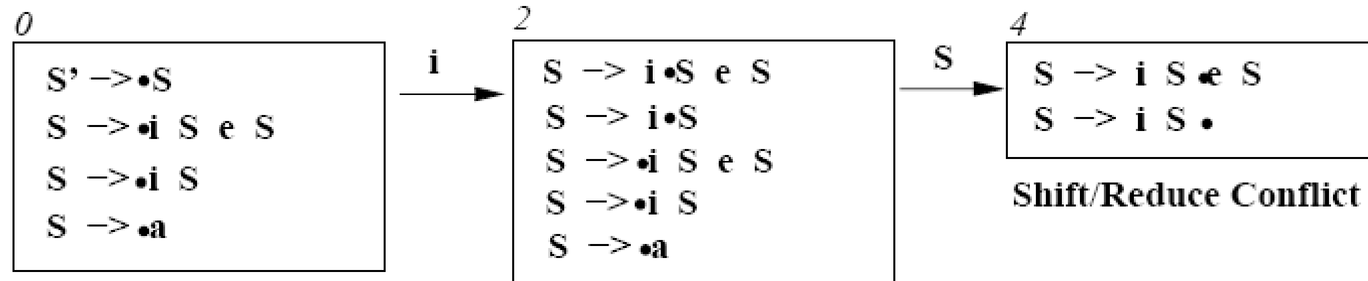
- Example: *id* + *id* * id or *id* + *id* + *id*
  - STACK: 0 E 1 + 4 E 7,
  - Input: * *id* $: Shift, + *id* $: Reduce
- Example *id* * *id* + *id* or *id* * *id* * *id*
  - STACK 0 E 1 * 5 E 8
  - Input: * *id* $: Reduce, + *id* $: Reduce
- Note that LR parsing table using ambiguous grammar in pp. 250 is smaller than that of unambiguous in pp 219
- A Rule of thumb
  - Disambiguating using precedence/associativity is harder to do for reducer/reduce conflicts

Dangling-Else Ambiguity

- Conditional statements
  - *stmt* → if *expr* then *stmt* else *stmt*
  - *stmt* → if *expr* then *stmt*
  - *stmt* → *other*
- Simplified Grammar
  - S' → S
  - S → iSeS | iS | a

  (i: if *expr* then, *e*: else, *a*: all others)

Build SLR(1) DFA



- Parsing Conflict in state 4
  - Should shift **else** since it is associated with previous **then**
  - Example: *iiaea*

Summary of LR(k) Parsing

- Much powerful than LL(k) parsing
  - Why? A nice exam question
- SLR(1), LR(1), LALR(1)
- Using ambiguous grammar with LR(1)
  - Resolving conflicts with disambiguation rule
- Project #2