Compilers Course
Lecture 6: Bottom-Up (Shift-Reduce) Parsing

A shift-reduce parser works as follows:

- Read tokens one by one and push them on the stack (shift) .
- If the top N symbols on the stack equal β in a production X → β, and the prefix of the stack "expects" an X, replace β with X on the stack (reduce) .
- Accept if the stack is just the start symbol S and the input is empty .

Note: the difference with LL(1), here choices are based on having seen complete right-hand sides. LL(1) has to choose having only seen a a single token that may be at the start of a right-hand side.

Example: Consider this grammar G1:

S → if E then S
    | id (proc. call)
E → id | num

Stack	**Input**	**Action**
$	if id then id$	shift
$if	id then id$	shift
$if id	then id$	
Problem: reduce S → id or E → id? Answer: reduce E → id		
$if E	then id$	shift
$if E then	id$	shift
$if E then id	$	
Problem: reduce S → id or E → id? Answer: reduce S → id		
$if E then S	$	reduce S → if E then S
$S	$	accept

This corresponds to the right-most derivation:

$S \Rightarrow_{rm}$ if E then S $\Rightarrow_{rm}$ if E then id $\Rightarrow_{rm}$ if id then id

which the parser traced in reverse order.

Choosing between reductions

The central question for a shift-reduce parser is:

"If two productions match the top of the stack, how does the parser choose which one to reduce? "

(If no reduction is applicable, the machine shifts.)

Tracking Candidates

In steps 3 and 6 above, the top of the stack matched both S → id and E → id. Why did the parser choose the second in step 3 and the first in step 6? The answer is that by scanning the stack the parser can identify the context to the left of the production it might reduce. That tells it if the production is in fact valid for that context.

Consider a transition system for a shift-reduce parser where states are pairs <stack of symbols, a candidate production with a "progress" point>.

In the initial state the stack is empty, and the candidate is the start symbol. We augment the grammar (add S' → S), and describe this as:

$<\varepsilon, S' \rightarrow .S>$

If the candidate in a state has a progress point before a symbol X, then the parser can shift that symbol onto the stack and move the progress point past that symbol:

$$<\gamma\ \alpha, A \rightarrow \alpha\ .\ X\ \beta> \xrightarrow{X} <\gamma\ \alpha\ X, A \rightarrow \alpha\ X\ .\ \beta>$$

If the candidate in a state has a progress point before a non-terminal B, then the productions of B are new candidates:

$$<\gamma\ \alpha, A \rightarrow \alpha\ .\ B\ \beta> \xrightarrow{\varepsilon} <\gamma\ \alpha, B \rightarrow .\ \delta>$$
for every production $B \rightarrow \delta$

Finally, if at some point the machine arrives at a state $<\gamma\ \alpha, A \rightarrow\ \alpha\ .\ >$, then $A \rightarrow \alpha$ is both "complete" and known to be valid. So the machine could reduce the stack to $\gamma\ A$ at that point.

The stacks don't actually affect the transitions but instead describe the path taken from the start state to the current state. Erasing the stacks turns this machine into an NFA_ε, with "candidate productions" as states.

Formal definitions

Consider a step in a right-most derivation:

$S \Rightarrow^*_{rm} \delta A w \Rightarrow_{rm} \delta \beta w$

w: remaining input
β: the handle (top of the stack)
δ β: a viable prefix (the entire stack)

Every prefix of δ β is a viable prefix.

An LR(0) item is a production $A \rightarrow X_1 \ldots X_n$ with a dot before any X_i or after X_n.

Example: S → abC has items S → . a b C, S → a . b C, S → a b . C, and S → a b C .

An item with the dot before X_1 is an initial item.
An item with the dot after X_n is a complete item.

An item A → α . β is valid for a viable prefix δ α if

$S \Rightarrow^*_{rm} \delta A w \Rightarrow_{rm} \delta \alpha \beta w$

If A → δ . is a complete item valid for δ α (the stack), then

$\delta A w \Rightarrow_{rm} \delta \alpha w$

could have been the last step in a right-most derivation of δ α w. It could be that (A → α .)is valid for some other remaining input w', or there could be other complete items valid for δ α

Intuitively, a grammar is defined to be LR(0) if (A → α .) is the only complete item valid for δ α.

Parsing a string w then becomes a process of finding handles and doing backwards derivations until w has been reduced to S.

Recognising valid items

For each viable prefix δ α (stack of symbols), we must now be able to compute its set of valid items.

Theorem: for every CFG G, its set of viable prefixes is a regular language recognized by an NFA_ε M whose states are LR(0) items for G. M is defined as follows:

1. $\delta[q_0, \varepsilon]$ = { S' → . S } S' → S is added to the grammar and S' is made the new start symbol; this is called augmenting the grammar .
2. δ[A → α . B β, ε] = { B → . γ | B → γ is a production} dot before variable implies ε transitions to initial items for each production of that variable .
3. δ[A → α . X β, X] = { A → α X . β} this is called "moving the dot" .

Example: (grammar G1)

Augmented grammar:
(p_0) S' → S
(p_1) S → if E then S
(p_2) S → id
(p_3) E → id
(p_4) E → num

Items/States:
I_0: S' → . S
I_1: S' → S .
I_2: S → . if E then S
I_3: S → if . E then S
I_4: S → if E . then S
I_5: S → if E then . S
I_6: S → if E then S .
I_7: S → . id
I_8: S → id .
I_9: E → . id
I_{10}: E → id .
I_{11}: E → . num
I_{12}: E → num .

Transitions:
$\delta[I_0, \varepsilon] \rightarrow \{I_2, I_7\}$
$\delta[I_0, S] \rightarrow I_1$

$\delta[I_2, \text{if}] \rightarrow I_3$

$\delta[I_3, \varepsilon] \rightarrow \{I_9, I_{11}\}$
$\delta[I_3, E] \rightarrow I_4$

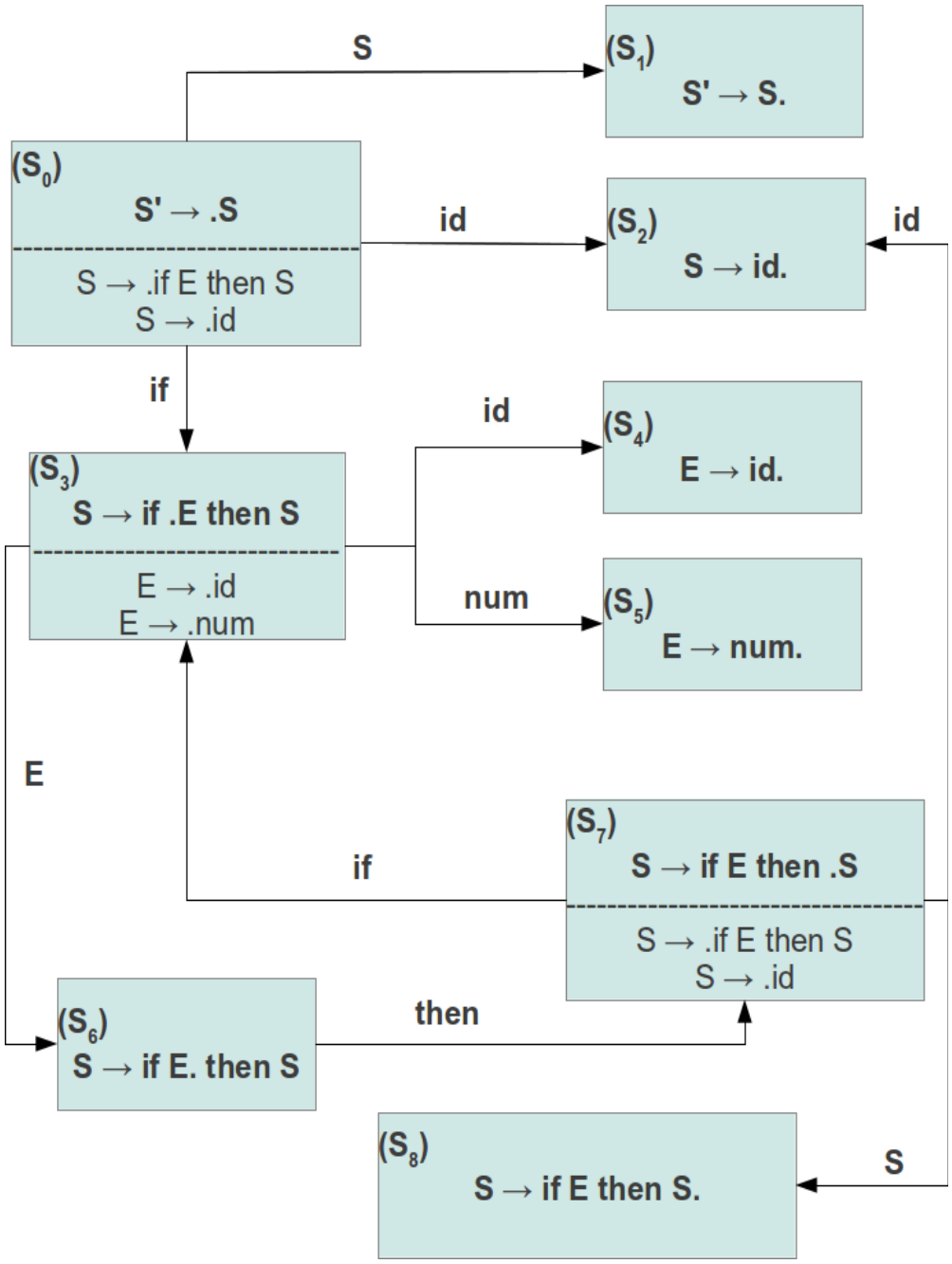
$\delta[I_4, \text{then}] \rightarrow I_5$

$\delta[I_5, \varepsilon] \rightarrow \{I_2, I_7\}$
$\delta[I_5, S] \rightarrow I_6$

$\delta[I_7, \text{id}] \rightarrow I_8$

$\delta[I_9, \text{id}] \rightarrow I_{10}$
$\delta[I_{11}, \text{num}] \rightarrow I_{12}$

Conversion to DFA:



The items above "-------" are called the "kernel", the items below come from ε-transitions and are called the "closure".

At each step, the shift-reduce parser applies this automaton to its stack. Let's say it ends at some state (set of items) *I*:

- If *I* contains a complete item *(A → α .)*, then reduce
- If *I* contains a non-complete item, the current input terminal is *T*, and *I* has a transition on *T*, then shift
- If *I* contains the complete item *(S' → S .)* and the input is empty, accept

Example 1: (grammar G1)

$	if id then id$	State S_0: shift on if
$if	id then id$	State S_3: shift on id
$if id	then id$	State S_4: reduce E → id
$if E	then id$	State S_6: shift on then
$if E then	id$	State S_7: shift on id
$if E then id	$	State S_2: reduce S → id
$if E then S	$	State S_8: reduce → if E then S
$S	$	State S_1: accept

LR(0) action/goto tables

Given a DFA recognizing viable prefixes, the LR(0) decision tables are defined as follows:

- Action: matrix indexed by *states* and *terminals*, containing *actions*:
Action[q, X] = shift q'
if X is a terminal and q → q' on X in the DFA
Action[q, X] = reduce A → α
if A → α . is a complete item in q
for all terminals X
Action[q, $] = accept
if S' → S . is a complete item in q
- Goto: matrix indexed by *states* and *variables*, containing *states*
Goto[q, X] = q'
if X is a variable and q → q' on X

	Action					GOTO	
	if	then	id	num	$	S	E
0	s_3		s_2			g_1	
1					A		
2	r_2	r_2	r_2	r_2	r_2		
3			s_4	s_5			g_6
4	r_3	r_3	r_3	r_3	r_3		
5	r_4	r_4	r_4	r_4	r_4		
6		s_7					
7	s_3		s_2			g_8	
8	r_1	r_1	r_1	r_1	r_1		

(S_N means "shift to state *N*", r_N means "reduce by production *N*", A means "accept", g_N means "goto state *N*")

Implementation note 1: instead of scanning the entire stack in each step, push the new state after pushing each symbol. Base decisions only on top-most state and current input symbol.

Stack	Input	Action
$0	if id then id$	Shift 3
$0 if 3	id then id$	Shift 4
$0 if 3 id 4	then id$	Reduce E → id pop 2 * |id| symbols, exposing state 3 push E, goto[3, E] = 6
$0 if 3 E 6	then id$	Shift 7
$0 if 3 E 6 then 7	id$	Shift 2
$0 if 3 E 6 then 7 id 2	$	Reduce S → id pop 2 * |id| symbols, exposing state 7 push S, goto[7, S] = 8
$0 if 3 E 6 then 7 S 8	$	Reduce S → if E then S pop 2 * |if E then S| symobls, exposing state 0 push S, goto[0, S]-1
$0 S 1	$	accept

Implementation note 2: the symbols on the stack are now redundant, so only push states.

LR(0) Parsing Algorithm

Let S be a stack of states, and SP the current position in S.

```
Init:   SP := 0; S[SP] := q0
Loop:
        let q = S[SP] and t = current input terminal
        if action[q,t] = shift q' then:
                push q' (SP := SP+1; S[SP] := q')
                remove t from input
                continue at Loop
        if action[q,t] = reduce A → β then:
                pop β from S (SP := SP - |β|)
                let q' = S[SP] and q" = goto[q',A]
                push q" (SP := SP+1; S[SP] := q")
                continue at Loop
        if action[q,t] = accept then
                done
```

Semantic actions, construction of AST

Each position in the state stack corresponds to a terminal, or a variable that is the result of reducing some production.

Both terminals and variables can have attributes.

We will add a "semantic stack" SEM that parallels the state stack S, such that SEM[SP] contains the attribute(s) for the symbol that branched to the state S[SP].

At actions shift q':
    push t's attribute on SEM (SEM[SP] := t's attribute)

At actions reduce A → β:
    evaluate synthesized attribute before popping S
    after pushing the new state, push attribute on SEM

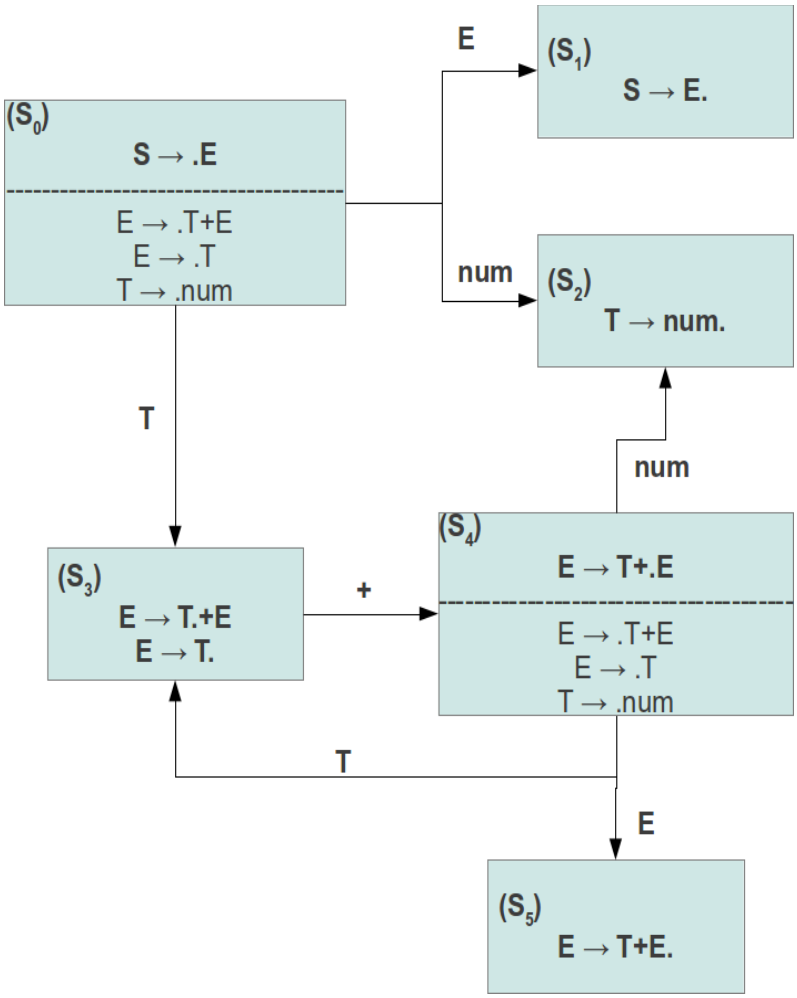
Example:
T → T x F (MKBINARY(MUL,T,F))

```
T := SEM[SP-2]
F := SEM[SP-0]
SP := (SP - 3) + 1
SEM[SP] := MKBINARY(MUL,T,F)
```

Simple LR, SLR

LR(0) is still fairly weak; consider:

S → E
E → T+E
E → T
T → num



When defining the Action table,

Action[S_3, +] is shift 4, BUT Action[S_3, +] is reduce E → T because E → T. is a complete item, and LR(0) reduces for all terminals when seeing a complete item.

This is a classic shift/reduce conflict, caused in this case by LR(0)'s failure to use lookahead (the current input terminal) to decide whether to reduce or not.

Simple LR (SLR):
Action[q, X] = reduce A → α
if *(A → α .)* is a complete item in q
∀ X ϵ FOLLOW(A)

In the example, FOLLOW(S) = {$}, FOLLOW(E) = FOLLOW(S) = {$}, and FOLLOW(T) = {+, $}. So in S_3 we choose shift 4 for + and reduce E → T for $.

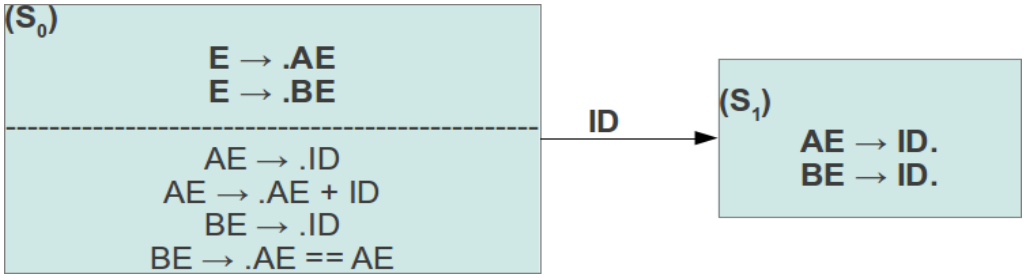
SLR is strong enough to handle many actual programming languages.

Reduce/Reduce Conflicts

This grammar attempts to describe arithmetic and boolean expressions as separate entities:

E → AE | BE
AE → ID | AE + ID
BE → ID | AE == AE

Now consider the LR(0) DFA for this grammar:



Now, FOLLOW(E) = {$}, FOLLOW(AE) = {+, ==, $}, and FOLLOW(BE) = FOLLOW(E) = {$}. So when the current input terminal is $ (in S_1), we can reduce either AE → ID or BE → ID.

This is called a reduce/reduce conflict, and is usually a sign of an ambiguous grammar.

In this case, the ambiguity comes from the fact that some arithmetic and boolean expressions look the same, so the grammar is unable to decide which interpretation to make.

The typical solution is to delay attempts at "type checking" to a separate type checking pass that runs after the parser has produced a parse tree. A better grammar for this language is:

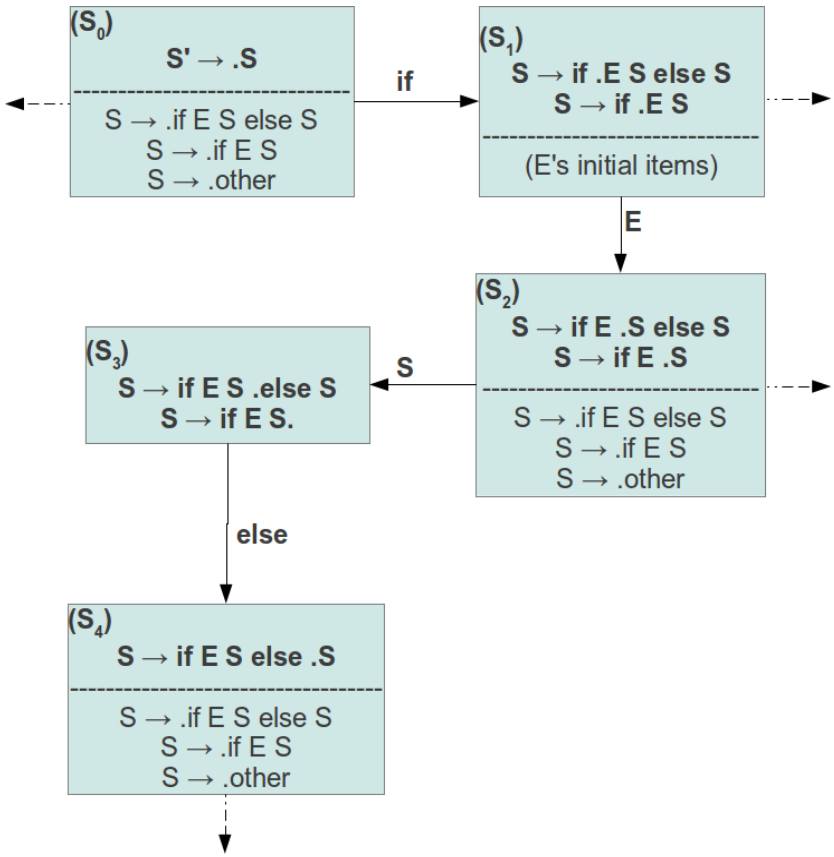
E → BE
BE → AE == AE | AE
AE → AE + ID | ID

Dangling-Else

S' → S
S → if E S else S
S → if E S
S → other

This is an ambiguous grammar for the dangling-else construct.

Constructing the LR(0) DFA for this grammar results in:



Now, FOLLOW(S) = {else} ∪ FOLLOW(S') = {else, $}, so in S_3 we should shift to S_4 on else, but we should also reduce S → if E S on else.

So the well-known dangling-else ambiguity results in a shift/reduce conflict in the LR parser.

One solution is to make the grammar unambiguous, as shown earlier.

However, this case is very common so LR parser generators have a heuristic which is to *prefer shifting over reducing* whenever there is a shift/reduce conflict. In general that's not correct, but for the

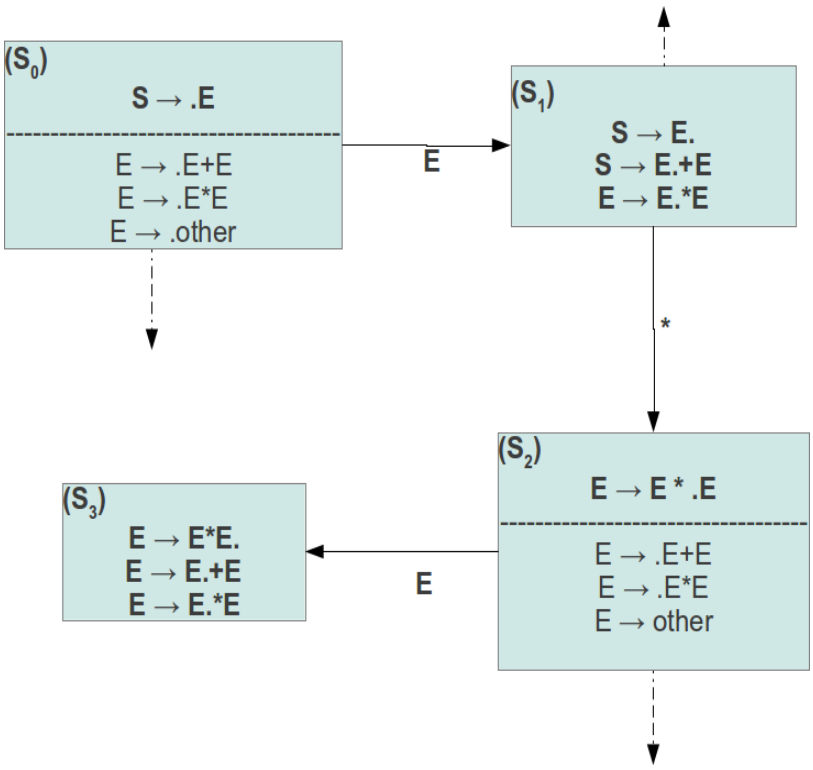
dangling-else it *is* the right solution since it leads to the else part being connected to the nearest preceding if that doesn't already have an else part.

Precedence and Associativity

Consider the ambiguous grammar:

S → E
E → E + E | E * E | other

Let's see what happens in the LR(0) DFA:



Now, since FOLLOW(E) includes + and *, we will have two shift/reduce conflicts in S_3:

1. On * shift or reduce E → E*E this conflict comes from the grammar's failure to express associativity of *
2. On + shift or reduce E → E*E this conflict comes from the grammar's failure to express precedence between + and *

Yacc and similar LR parser generators accept declarations to resolve these kinds of conflicts:

%left <token>

When there is a choice between reducing a rule containing the <token> or shifting that <token>, the

parser will choose the reduction, which effectively makes the <token> left-associative. So "%left *" would solve ambiguity number 1 above. Similarly, %right <token> causes the parser choose to shift instead of reduce, making the <token> right-associative. The order of the %left and %right declarations cause the tokens to be given different precedences:

%left +
%left *

means that * is given higher precedence than +: when there is a choice between reducing a rule involving one token or shifting another token, the choice involving the higher-precedence token is the one taken.

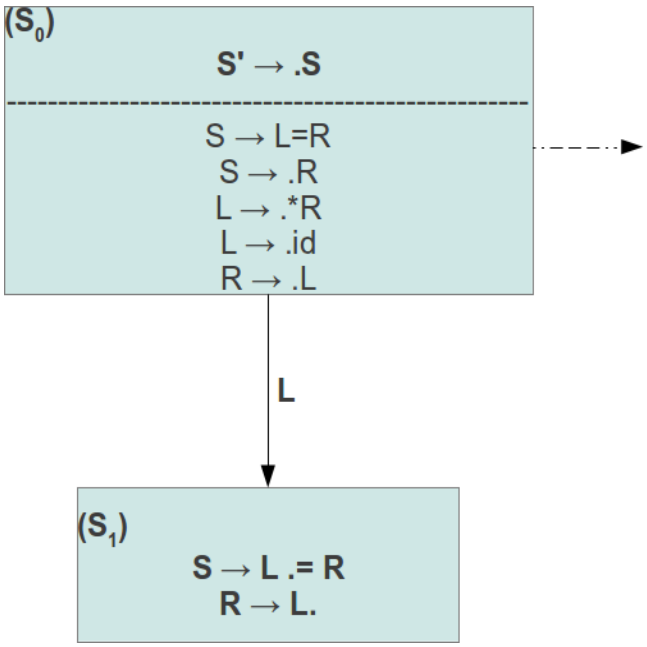
LR(1)

SLR is stronger than LR(0) since it uses lookahead to decide whether to reduce or not, which eliminates some shift/reduce and reduce/reduce conflicts.

Example: The grammar in Example 4.48 of the Purple Dragon book:

S' → S
S → L=R | R
L → *R | id
R → L

LR(0) DFA:



Now, FOLLOW(R) = { =, $ } (because S → L=R and L → *R), so SLR wants to shift on = and also

reduce R → L on =.

The problem with SLR is that the lookahead information used is FOLLOW, which is a global property for *all* uses of the non-terminal. It is in general an over-approximation for a specific DFA state with a complete item.

LR(1) changes the NFA construction to explicitly include and propagate accurate lookahead information:

1. Items are pairs <LR(0) item, lookahead terminal symbol>
2. $\delta[q_0, \varepsilon] = \{ \langle S' \rightarrow .S, \$ \rangle \}$
3. $\delta[\langle A \rightarrow \alpha . B \beta, Z \rangle, \varepsilon] = \{ \langle B \rightarrow . \gamma, y \rangle \mid B \rightarrow \gamma$ is a production, and $y \in FIRST(\beta Z)\}$
   B is followed by β in this context, so FIRST(β) provides the possible lookahead symbols for B's initial items. If β derives the empty string, the context's lookahead Z is propagated instead. Hence FIRST(β Z).
4. $\delta[\langle A \rightarrow \alpha . X \beta, Z \rangle, X] = \{ \langle A \rightarrow \alpha X . \beta, Z \rangle \}$ moving the dot doesn't change the lookahead symbol

In the action table:
Action[q, X] = reduce A → α if <A → α . , X> is an item in q. The item provides its own lookahead symbol, and the reduction is only done if the input matches the lookahead.

LR(1) is much stronger than SLR or LR(0), in fact, LR(1) corresponds exactly to the class of "deterministic push-down automata".

Unfortunately, the number of LR(1) states can be orders of magnitude larger than the number of SLR/LR(0) states, so LR(1) parsers are rarely used in practice.

LALR(1)

LALR(1) stands for "lookahead LR(1)". In principle:

1. Build the LR(1) DFA
2. Merge states with identical LR(0) items, and use the union of their lookaheads

The result is a DFA with the same number of states as SLR and LR(0) would have, but with items <LR(0) item, set of lookahead symbols>.

In practice, the LALR(1) DFA is constructed from the LR(0) DFA by a process of lookahead propagation.

LALR(1) is implemented by tools like Yacc and its clones (Bison, ML-Yacc), and is very popular among language designers and compiler implementors.