More on Grammars

Recap: $LR(0)$ parsing

- We have seen the $LR(0)$ parsing automaton
  - is a bottom-up automaton
  - no lookahead!
  - is deterministic if grammar is $LR(0)$
  - provides the (reversed) rightmost analysis of the input
- Non-$LR(0)$ grammars will cause conflicts that are visible in the states (item sets) of the goto automaton:
  - reduce/reduce conflict if an item set contains two items $[A \to \alpha\ \cdot]$ and $[B \to \beta\ \cdot]$
  - shift/reduce conflict if an item sets contains two items $[A \to \alpha_1 \cdot a\alpha_2]$ and $[B \to \beta\ \cdot]$

Removing conflicts

- Let's look again at the shift-reduce conflict:
$$I = \{[A \to \alpha_1 \cdot a\alpha_2], [B \to \beta \cdot]\}$$
- If the parser is in the state $(aw, \ldots I, \ldots)$ it is quite clear that we should **shift** because the next input symbol is a
  - Of course, this only works if $a \notin follow_1(B)$
  - For example, if the grammar contains a rule like $C \to Ba$, then the conflict cannot be avoided

- Same for the reduce-reduce conflict:
$$I = \{[A \to \alpha \cdot], [B \to \beta \cdot]\}$$
- If the parser is in the state $(aw, \ldots I, \ldots)$ we can decide whether to reduce to A or to B by checking whether $a \in follow_1(A)$ or $a \in follow_1(B)$
  - Again, this will not work if $a \in follow_1(A) \cap follow_1(B)$

$SLR(1)$ parsing

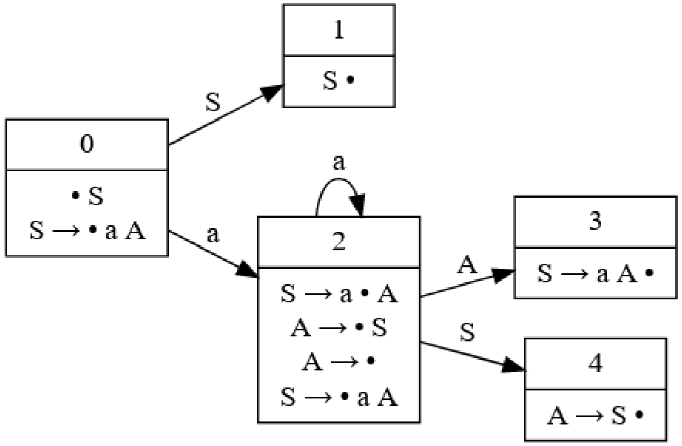
- An $SLR(1)$ ("Simple LR(1)") parser automaton is doing what we have seen on the previous slide
- Using a *lookahead* of one input symbol, we extend the $LR(0)$ automaton to an $SLR(1)$ automaton:
  - **Shift** $(aw, \alpha I, z) \rightarrow (w, \alpha IJ, z)$ if $[A \rightarrow \alpha_1 \cdot a\alpha_2] \in I$ and $I \underset{goto}{\overset{a}{\rightarrow}} J$
  - **Reduce** $(aw, \alpha I I_1 \dots I_n, z) \rightarrow (aw, \alpha IJ, zi)$ with rule $i \neq 0$ $A \rightarrow Y_1 \dots Y_n$ if $[A \rightarrow Y_1 \dots Y_n \cdot] \in I_n$ and $I \underset{goto}{\overset{A}{\rightarrow}} J$ and $a \in follow_1(A)$
  - **Accepting** state $(\varepsilon, I_0 I, z) \rightarrow (\varepsilon, \varepsilon, z0)$ if $[S' \rightarrow S \cdot] \in I$
  - **Error** otherwise

- A CFG is an $SLR(1)$ grammar if there is no conflict with the above actions

Action table of $SLR(1)$ automaton

- Grammar with Shift-Reduce conflict in $LR(0)$ parser:

$$S' \to S \qquad (0)$$
$$S \to aA \qquad (1)$$
$$A \to S \mid \varepsilon \qquad (2,3)$$



- Action table of $LR(0)$ parser

Item set	action
I_0	shift
I_1	accept
I_2	shift - reduce 3
I_3	reduce 1
I_4	reduce 2

Action table of $SLR(1)$ parser

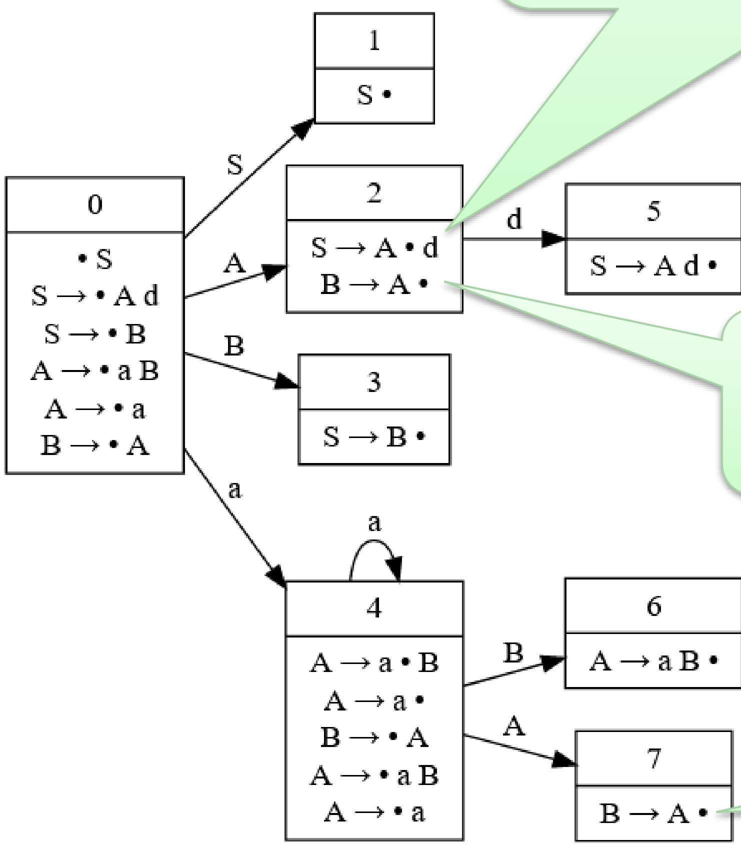
Item set	action with lookahead	
	a	$ (end of input)
I_0	shift	
I_1		accept
I_2	shift	reduce 3
I_3		reduce 1
I_4		reduce 2

Limitation of $SLR(1)$: Example

- Here is a grammar that has a shift/reduce conflict with an $SLR(1)$ parser:

$$S \rightarrow A\ d \mid B$$
$$A \rightarrow a\ B \mid a$$
$$B \rightarrow A$$

The SLR(1) parser doesn't know whether it should **shift** d or **reduce** $B \rightarrow A$ if the next input symbol is d because $d \in follow_1(B)$

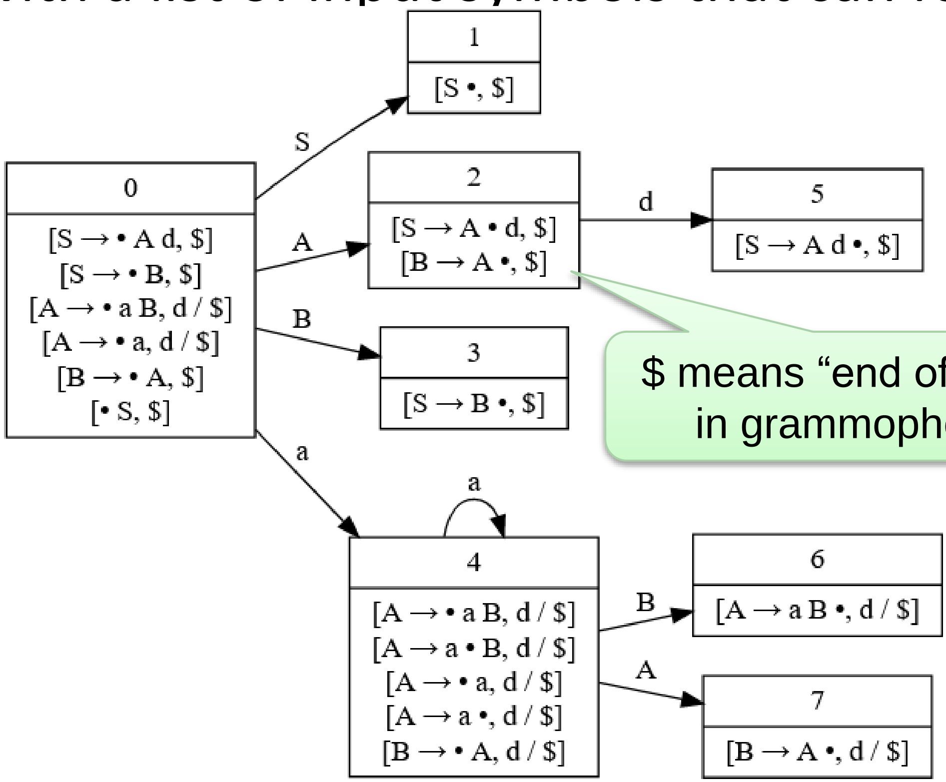


d can never follow B in this situation $S \Rightarrow B$

Only here, d could follow B

$LR(1)$ parsing

- $SLR(1)$ only checks the follow set of a non-terminal symbol N, it doesn't check whether the next input symbol can follow N *for that specific occurrence* of N
  - That's why $SLR(1)$ is called "Simple LR(1)". It doesn't take full advantage of the lookahead (but it's easy to implement)
- For a full $LR(1)$ parser, we extend *each item* in the the goto automaton with a list of input symbols that can follow



$ means "end of input"
in grammophone

More parsers...

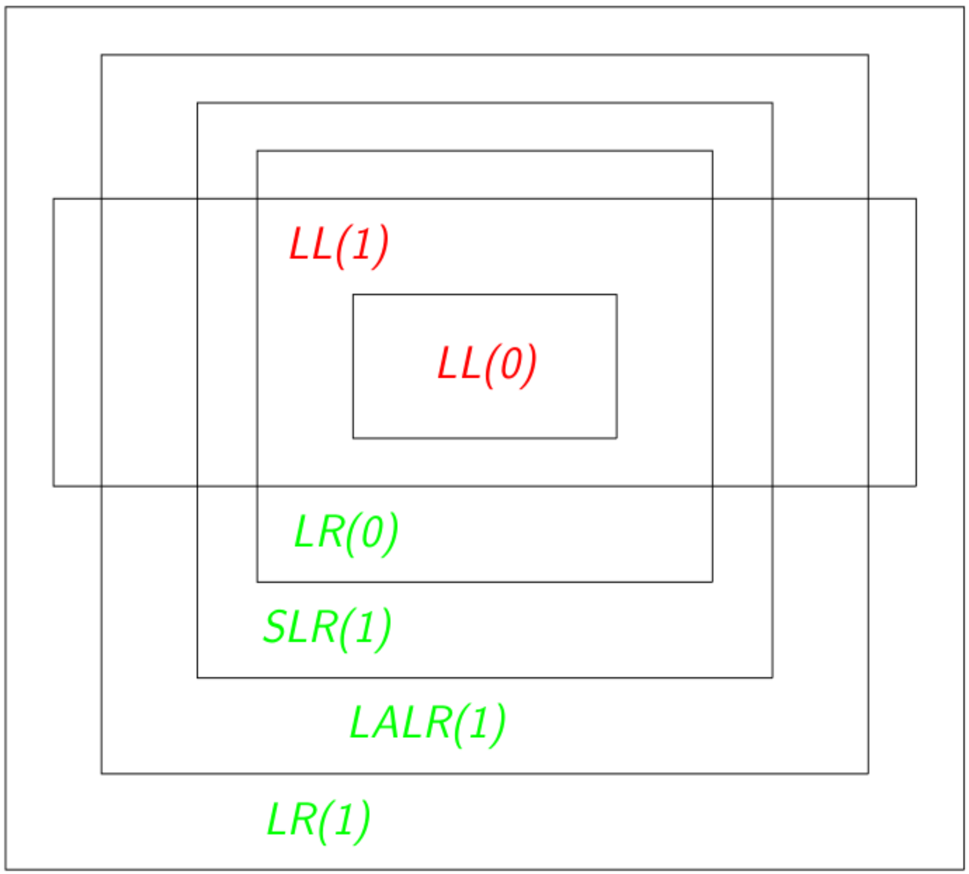
- $LR(k)$ parsers can parse any context free language that can be parsed with a deterministic push-down automaton
  - Complexity $O(n)$
  - But can result in large automata with many table entries
- $LALR(1)$ reduces the table size by merging similar states, i.e., two $LR(1)$ item sets $I_x = \{[A \to B \cdot, a]\}$ and $I_y = \{[A \to B \cdot, b]\}$ could be merged to a single state $I_{xy} = \{[A \to B \cdot, a/b]\}$
  - A compromise between $LR(1)$ and $SLR(1)$
  - Less powerful than $LR(1)$
  - More powerful than $SLR(1)$
  - Very popular in parser generator tools, used in the yacc/bison parser

Parsers for general CFG

- $GLR(0)$
  - non-deterministic push-down automaton, tries all possibilities in parallel, but uses clever data structures to reduce memory consumption
  - $O(n^3)$ for general CFG, $O(n)$ if grammar is deterministic
- GLL
  - like GLR but with top-down automaton
  - $O(n^3)$
- Earley
  - $O(n^3)$
- ANTLR
  - $LL(k) + \text{backtracking} + \text{caching for better performance}$
  - $O(n^4)$

Comparison of grammars

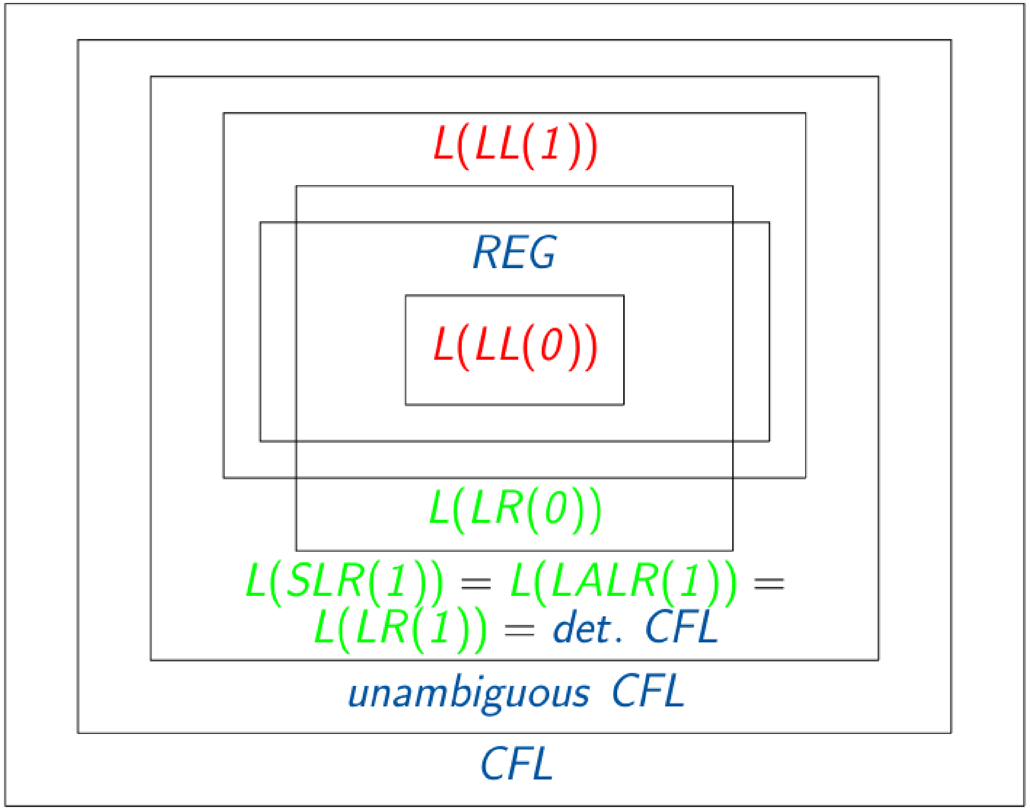
- $LR(0) \subset SLR(1) \subset LALR(1) \subset LR(1) \subset LR(2) \subset \cdots$
- $LL(0) \subset LL(1) \subset LL(2) \subset \cdots$
- $LL(k) \subset LR(k) \subset CFG$



Source: RWTH Aachen

Expressiveness of grammars

- $L(C)$:= All languages generated by the grammars of class C
- $L(LR(1)) = L(SLR(1)) = L(LALR(1)) = L(LR(k)) \subset CFL$
- $L\big(LL(0)\big) \subset L(RE) \subset L\big(LL(1)\big) \subset \cdots \subset L(LL(k)) \subset L(LR(1))$
- Note that $L(RE)$ is not completely in $L(LR(0))$



Source: RWTH Aachen

Chomsky Hierarchy of Grammars

- Type 3: Regular Grammars (Regular Expressions)
  - Parsed with DFA
- Type 2: Context-Free Grammars
  - Allows recursion, e.g. $A \rightarrow (\ A\ )\ \mid\ \varepsilon$
  - Parsed with Nondeterministic Pushdown Automaton
- Type 1: Context-Sensitive Grammars
  - Allows rules like $xAy \rightarrow aBCb$, i.e., the parser must consider the context $x \dots y$ around A when selecting a rule to expand A
  - Parsed with Finite-tape Turing Machine
- Type 0: Everything allowed that you can program in the parser ☺

- Type 0 and type 1 grammars are mostly useless in practice
- Most compilers use a hand-written or generated $LL(1)$ or $xLR(1)$ parser plus extra code to resolve ambiguities
  - Example C: `(x)*y` Multiplication or Pointer-dereferencing with cast?