# Compiler Construction

## Lecture 14: Syntactic Analysis IX (Practical Issues)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH Aachen University

noll@cs.rwth-aachen.de

http://www-i2.informatik.rwth-aachen.de/i2/cc10/

Winter semester 2010/11

# Outline

# $LALR(1)$ Sets I

## Corollary

*For every* $G \in CFG_\Sigma$, $|LR(1)(G)/\sim_0| = |LR(0)(G)|$.

**Idea:** merge $LR(0)$-equivalent $LR(1)$ sets (maintaining the lookahead information, but possibly introducing conflicts)

## Definition ($LALR(1)$ sets)

Let $G \in CFG_\Sigma$.

- An information $I \in LR(1)(G)$ determines the $LALR(1)$ set
$$\bigcup [I]_{\sim_0} = \bigcup \{I' \in LR(1)(G) \mid I' \sim_0 I\}.$$
- The set of all $LALR(1)$ sets of $G$ is denoted by $LALR(1)(G)$.

**Remark:** by Corollary 13.16, $|LALR(1)(G)| = |LR(0)(G)|$
(but $LALR(1)$ sets provide additional lookahead information)

# $LALR(1)$ Sets II

## Example (cf. Example 13.15)

$G_{LR} : S' \to S \quad S \to L\texttt{=}R \mid R \quad L \to \texttt{*}R \mid \texttt{a} \quad R \to L$

$LR(0)(G_{LR})$ :

$I_0(\varepsilon)$ : $[S' \to \cdot S]$ $[S \to \cdot L\texttt{=}R]$ $[S \to \cdot R]$ $[L \to \cdot\texttt{*}R]$ $[L \to \cdot\texttt{a}]$ $[R \to \cdot L]$

$I_1(S)$ : $[S' \to S\cdot]$

$I_2(L)$ : $[S \to L \cdot \texttt{=}R]$ $[R \to L\cdot]$

$I_3(R)$ : $[S \to R\cdot]$

$I_4(\texttt{*})$ : $[L \to \texttt{*} \cdot R]$ $[R \to \cdot L]$ $[L \to \cdot\texttt{*}R]$ $[L \to \cdot\texttt{a}]$

$I_5(\texttt{a})$ : $[L \to \texttt{a}\cdot]$

$I_6(L\texttt{=})$ : $[S \to L\texttt{=} \cdot R]$ $[R \to \cdot L]$ $[L \to \cdot\texttt{*}R]$ $[L \to \cdot\texttt{a}]$

$I_7(\texttt{*}R)$ : $[L \to \texttt{*}R\cdot]$

$I_8(\texttt{*}L)$ : $[R \to L\cdot]$

$I_9(L\texttt{=}R)$ : $[S \to L\texttt{=}R\cdot]$

$LALR(1)(G_{LR})$ :

$I''_0 := I'_0$ : $[S' \to \cdot S, \varepsilon]$ $[S \to \cdot L\texttt{=}R, \varepsilon]$ $[S \to \cdot R, \varepsilon]$ $[L \to \cdot\texttt{*}R, \texttt{=}/\varepsilon]$ $[L \to \cdot\texttt{a}, \texttt{=}/\varepsilon]$ $[R \to \cdot L, \varepsilon]$

$I''_1 := I'_1$ : $[S' \to S\cdot, \varepsilon]$

$I''_2 := I'_2$ : $[S \to L \cdot \texttt{=}R, \varepsilon]$ $[R \to L\cdot, \varepsilon]$

$I''_3 := I'_3$ : $[S \to R\cdot, \varepsilon]$

$I''_4 := I'_4 \cup I'_{11}$ : $[L \to \texttt{*} \cdot R, \texttt{=}/\varepsilon]$ $[R \to \cdot L, \texttt{=}/\varepsilon]$ $[L \to \cdot\texttt{*}R, \texttt{=}/\varepsilon]$ $[L \to \cdot\texttt{a}, \texttt{=}/\varepsilon]$

$I''_5 := I'_5 \cup I'_{12}$ : $[L \to \texttt{a}\cdot, \texttt{=}/\varepsilon]$

$I''_6 := I'_6$ : $[S \to L\texttt{=} \cdot R, \varepsilon]$ $[R \to \cdot L, \varepsilon]$ $[L \to \cdot\texttt{*}R, \varepsilon]$ $[L \to \cdot\texttt{a}, \varepsilon]$

$I''_7 := I'_7 \cup I'_{13}$ : $[L \to \texttt{*}R\cdot, \texttt{=}/\varepsilon]$

$I''_8 := I'_8 \cup I'_{10}$ : $[R \to L\cdot, \texttt{=}/\varepsilon]$

$I''_9 := I'_9$ : $[S \to L\texttt{=}R\cdot, \varepsilon]$

# Outline

# $LALR(1)$ Conflicts

**But:** merging of $LR(1)$ sets can produce new conflicts (also see exercises):

## Example 14.1

$G : S' \to S \quad S \to \mathtt{a}A\mathtt{d} \mid \mathtt{b}B\mathtt{d} \mid \mathtt{a}B\mathtt{e} \mid \mathtt{b}A\mathtt{e} \quad A \to \mathtt{c} \quad B \to \mathtt{c}$

| | | | | |
|---|---|---|---|---|
| $LR(1)(\varepsilon)$ : | $[S' \to \cdot S, \varepsilon]$ | $[S \to \cdot\mathtt{a}A\mathtt{d}, \varepsilon]$ | $[S \to \cdot\mathtt{b}B\mathtt{d}, \varepsilon]$ | $[S \to \cdot\mathtt{a}B\mathtt{e}, \varepsilon]$ |
| | $[S \to \cdot\mathtt{b}A\mathtt{e}, \varepsilon]$ | | | |
| $LR(1)(S)$ : | $[S' \to S\cdot, \varepsilon]$ | | | |
| $LR(1)(\mathtt{a})$ : | $[S \to \mathtt{a} \cdot A\mathtt{d}, \varepsilon]$ | $[S \to \mathtt{a} \cdot B\mathtt{e}, \varepsilon]$ | $[A \to \cdot\mathtt{c}, \mathtt{d}]$ | $[B \to \cdot\mathtt{c}, \mathtt{e}]$ |
| $LR(1)(\mathtt{b})$ : | $[S \to \mathtt{b} \cdot B\mathtt{d}, \varepsilon]$ | $[S \to \mathtt{b} \cdot A\mathtt{e}, \varepsilon]$ | $[B \to \cdot\mathtt{c}, \mathtt{d}]$ | $[A \to \cdot\mathtt{c}, \mathtt{e}]$ |
| $LR(1)(\mathtt{a}A)$ : | $[S \to \mathtt{a}A \cdot \mathtt{d}, \varepsilon]$ | | $LR(1)(\mathtt{a}B)$ : | $[S \to \mathtt{a}B \cdot \mathtt{e}, \varepsilon]$ |
| $LR(1)(\mathtt{ac})$ : | $[A \to \mathtt{c}\cdot, \mathtt{d}]$ | $[B \to \mathtt{c}\cdot, \mathtt{e}]$ | | |
| $LR(1)(\mathtt{b}B)$ : | $[S \to \mathtt{b}B \cdot \mathtt{d}, \varepsilon]$ | | $LR(1)(\mathtt{b}A)$ : | $[S \to \mathtt{b}A \cdot \mathtt{e}, \varepsilon]$ |
| $LR(1)(\mathtt{bc})$ : | $[B \to \mathtt{c}\cdot, \mathtt{d}]$ | $[A \to \mathtt{c}\cdot, \mathtt{e}]$ | | |
| $LR(1)(\mathtt{a}A\mathtt{d})$ : | $[S \to \mathtt{a}A\mathtt{d}\cdot, \varepsilon]$ | | $LR(1)(\mathtt{a}B\mathtt{e})$ : | $[S \to \mathtt{a}B\mathtt{e}\cdot, \varepsilon]$ |
| $LR(1)(\mathtt{b}B\mathtt{d})$ : | $[S \to \mathtt{b}B\mathtt{d}\cdot, \varepsilon]$ | | $LR(1)(\mathtt{b}A\mathtt{e})$ : | $[S \to \mathtt{b}A\mathtt{e}\cdot, \varepsilon]$ |

no conflicts $\implies G \in LR(1)$

$LR(1)(\mathtt{ac}) \sim_0 LR(1)(\mathtt{bc})$, but $LR(1)(\mathtt{ac}) \cup LR(1)(\mathtt{bc})$ has conflicts
$\implies G \notin LALR(1)$

# Efficient Construction of $LALR(1)$ Parsers

Naive algorithm to construct $LALR(1)$ parser for $G \in CFG_\Sigma$:

1. Construct $LR(1)(G)$
2. Determine and merge $LR(0)$-equivalent $LR(1)$ sets

**Problem:** no reduction of peak space requirement

Idea of improved algorithm (see Aho/Lam/Sethi/Ullman: *Compilers: Principles, Techniques, and Tools*, 2nd ed., p. 270ff):

1. Represent each set of items by its kernel, i.e., by the items of the form $[S' \to \cdot S, \varepsilon]$ or $[A \to \beta_1 \cdot \beta_2, x]$ where $\beta_1 \neq \varepsilon$
2. Construct $LALR(1)$ kernels from $LR(0)$ kernels similarly to $LR(1)$ items
3. Compute $LALR(1)$ sets by taking the $\varepsilon$-closure

(applied in `yacc` parser generator)

# Outline

# Ambiguous Grammars

**Reminder (Definition 6.6):** a context-free grammar $G \in CFG_\Sigma$ is called unambiguous if every word $w \in L(G)$ has exactly one syntax tree. Otherwise it is called ambiguous.

## Lemma 14.2

*If $G \in CFG_\Sigma$ is ambiguous, then $G \notin \bigcup_{k \in \mathbb{N}} LR(k)$.*

## Proof.

Assume that there exist $k \in \mathbb{N}$ and $G \in LR(k)$ such that $G$ is ambiguous. Hence there exists $w \in L(G)$ with different right derivations. Let $\alpha A v$ be the last common sentence of the two derivations (i.e., $\beta \neq \beta'$):

$$S \Rightarrow_r^* \alpha A v \begin{cases} \Rightarrow_r \alpha\beta v \Rightarrow_r^* w \\ \Rightarrow_r \alpha\beta' v \Rightarrow_r^* w \end{cases}$$

But since $\text{first}_k(v) = \text{first}_k(v)$ for every $v \in \Sigma^*$, Definition 10.9 yields that $\beta = \beta'$. Contradiction □

However ambiguity is a natural specification method which generally avoids involved syntactic constructs.

# Bottom-Up Parsing of Ambiguous Grammars I

## Example 14.3 (Simple arithmetic expressions)

$G : E' \to E \quad E \to E\texttt{+}E \mid E\texttt{*}E \mid \texttt{a}$

Precedence: $\texttt{*} > \texttt{+}$ Associativity: left

(thus: $\texttt{a+a*a+a} := \texttt{(a+(a*a))+a}$)

$LR(0)(G)$:

$I_0 := LR(0)(\varepsilon) : \quad [E' \to \cdot E] \quad [E \to \cdot E\texttt{+}E] \quad [E \to \cdot E\texttt{*}E] \quad [E \to \cdot \texttt{a}]$

$I_1 := LR(0)(E) : \quad [E' \to E\cdot] \quad [E \to E \cdot \texttt{+}E] \quad [E \to E \cdot \texttt{*}E]$

$I_2 := LR(0)(\texttt{a}) : \quad [E \to \texttt{a}\cdot]$

$I_3 := LR(0)(E\texttt{+}) : \quad [E \to E\texttt{+} \cdot E] \quad [E \to \cdot E\texttt{+}E] \quad [E \to \cdot E\texttt{*}E] \quad [E \to \cdot \texttt{a}]$

$I_4 := LR(0)(E\texttt{*}) : \quad [E \to E\texttt{*} \cdot E] \quad [E \to \cdot E\texttt{+}E] \quad [E \to \cdot E\texttt{*}E] \quad [E \to \cdot \texttt{a}]$

$I_5 := LR(0)(E\texttt{+}E) : \quad [E \to E\texttt{+}E\cdot] \quad [E \to E \cdot \texttt{+}E] \quad [E \to E \cdot \texttt{*}E]$

$I_6 := LR(0)(E\texttt{*}E) : \quad [E \to E\texttt{*}E\cdot] \quad [E \to E \cdot \texttt{+}E] \quad [E \to E \cdot \texttt{*}E]$

Conflicts: $I_1$: $SLR(1)$-solvable (reduce on $\varepsilon$, shift on $\texttt{+}/\texttt{*}$)

$I_5$, $I_6$: not $SLR(1)$-solvable ($\texttt{+}, \texttt{*} \in \mathrm{fo}(E)$)

Solution:

$I_5$: $\texttt{*} > \texttt{+} \implies \mathrm{act}(I_5, \texttt{*}) := \mathsf{shift}$, $\texttt{+}$ left assoc. $\implies \mathrm{act}(I_5, \texttt{+}) := \mathsf{red}\,1$

$I_6$: $\texttt{*} > \texttt{+} \implies \mathrm{act}(I_6, \texttt{+}) := \mathsf{red}\,2$, $\texttt{*}$ left assoc. $\implies \mathrm{act}(I_6, \texttt{*}) := \mathsf{red}\,2$

# Bottom-Up Parsing of Ambiguous Grammars II

## Example 14.4 ("Dangling `else`")

$G : S' \to S \quad S \to \texttt{i}S\texttt{e}S \mid \texttt{i}S \mid \texttt{a}$

Ambiguity: `iiaea` := (1) `i(iaea)` (common) or (2) `i(ia)ea`

$LR(0)(G)$:

$I_0 := LR(0)(\varepsilon)$ : $[S' \to \cdot S]$ $[S \to \cdot\texttt{i}S\texttt{e}S]$ $[S \to \cdot\texttt{i}S]$ $[S \to \cdot\texttt{a}]$

$I_1 := LR(0)(S)$ : $[S' \to S\cdot]$

$I_2 := LR(0)(\texttt{i})$ : $[S \to \texttt{i} \cdot S\texttt{e}S]$ $[S \to \texttt{i} \cdot S]$ $[S \to \cdot\texttt{i}S\texttt{e}S]$ $[S \to \cdot\texttt{i}S]$ $[S \to \cdot\texttt{a}]$

$I_3 := LR(0)(\texttt{a})$ : $[S \to \texttt{a}\cdot]$

$I_4 := LR(0)(\texttt{i}S)$ : $[S \to \texttt{i}S \cdot \texttt{e}S]$ $[S \to \texttt{i}S\cdot]$

$I_5 := LR(0)(\texttt{i}S\texttt{e})$ : $[S \to \texttt{i}S\texttt{e} \cdot S]$ $[S \to \cdot\texttt{i}S\texttt{e}S]$ $[S \to \cdot\texttt{i}S]$ $[S \to \cdot\texttt{a}]$

$I_6 := LR(0)(\texttt{i}S\texttt{e}S)$ : $[S \to \texttt{i}S\texttt{e}S\cdot]$

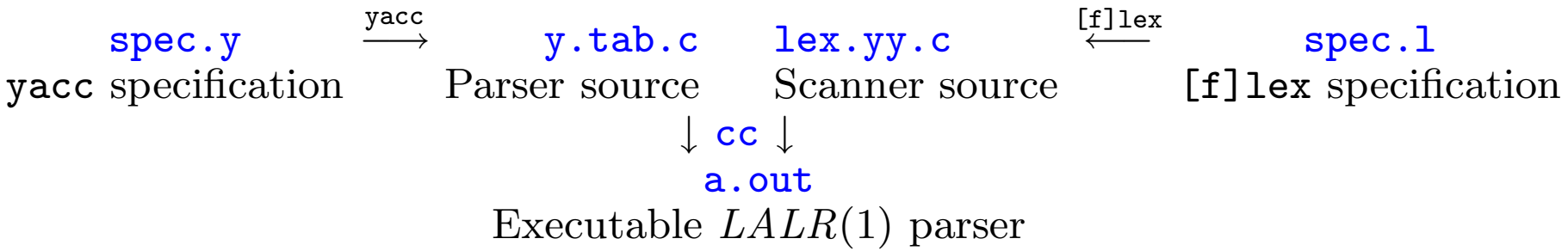
Conflict in $I_4$: $\texttt{e} \in \text{fo}(S) \implies$ not $SLR(1)$-solvable

Solution (1): $\text{act}(I_4, \texttt{e}) :=$ shift

# Outline

# The yacc Tool

Usage of yacc ("yet another compiler compiler"):




Like for `[f]lex`, a yacc specification is of the form

*Declarations (optional)*
%%
*Rules*
%%
*Auxiliary procedures (optional)*

# yacc Specifications

**Declarations:**

- Token definitions: `%token` *Tokens*
- Not every token needs to be declared (`'+'`, `'='`, `...`)
- Start symbol: `%start` *Symbol* (optional)
- C code for declarations etc.: `%{` *Code* `%}`

**Rules:** context-free productions and semantic actions

- $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ represented as

$$\begin{array}{lll} A & : & \alpha_1 \quad \{Action_1\} \\ & | & \alpha_2 \quad \{Action_2\} \\ & \vdots & \\ & | & \alpha_n \quad \{Action_n\}; \end{array}$$

- Semantic actions = C statements for computing attribute values
- `$$` = attribute value of $A$
- `$i` = attribute value of $i$th symbol on right-hand side
- Default action: `$$ = $1`

**Auxiliary procedures:** scanner (if not `[f]lex`), error routines, ...

# Example: Simple Desk Calculator I

```
%{/* SLR(1) grammar for arithmetic expressions (Example 12.9) */
  #include <stdio.h>
  #include <ctype.h>
%}
%token DIGIT
%%
line   : expr '\n'          { printf("%d\n", $1); };
expr   : expr '+' term      { $$ = $1 + $3; }
       | term               { $$ = $1; };
term   : term '*' factor    { $$ = $1 * $3; }
       | factor             { $$ = $1; };
factor : '(' expr ')'       { $$ = $2; }
       | DIGIT              { $$ = $1; };
%%
yylex() {
  int c;
  c = getchar();
  if (isdigit(c)) yylval = c - '0'; return DIGIT;
  return c;
}
```

# Example: Simple Desk Calculator II

```
> yacc calc.y
> cc y.tab.c -ly
> a.out
2+3
5
> a.out
2+3*5
17
```

# An Ambiguous Grammar I

```
%{/* Ambiguous grammar for arithm. expressions (Ex. 14.3) */
  #include <stdio.h>
  #include <ctype.h>
%}
%token DIGIT
%%
line   : expr '\n'          { printf("%d\n", $1); };
expr   : expr '+' expr      { $$ = $1 + $3; }
       | expr '*' expr      { $$ = $1 * $3; }
       | DIGIT              { $$ = $1; };
%%
yylex() {
  int c;
  c = getchar();
  if (isdigit(c)) {yylval = c - '0'; return DIGIT;}
  return c;
}
```

# An Ambiguous Grammar II

Invoking `yacc` with the option `-v` produces a report `y.output`:

```
  ...
State 8

    2 expr: expr . '+' expr
    2     | expr '+' expr .
    3     | expr . '*' expr

    '+'  shift and goto state 6
    '*'  shift and goto state 7

    '+'        [reduce with rule 2 (expr)]
    '*'        [reduce with rule 2 (expr)]


State 9

    2 expr: expr . '+' expr
    3     | expr . '*' expr
    3     | expr '*' expr .

    '+'  shift and goto state 6
    '*'  shift and goto state 7

    '+'        [reduce with rule 3 (expr)]
    '*'        [reduce with rule 3 (expr)]
```

# Conflict Handling in yacc

Default conflict resolving strategy in `yacc`:

reduce/reduce: choose first conflicting production in specification

shift/reduce: prefer shift

- resolves dangling-else ambiguity (Example 14.4) correctly
- also adequate for strong following weak operator (`*` after `+`; Example 14.3) and for right-associative operators
- not appropriate for weak following strong operator and for left-associative binary operators
  ($\Longrightarrow$ reduce; see Example 14.3)

For ambiguous grammar:

```
> yacc ambig.y
conflicts: 4 shift/reduce
> cc y.tab.c -ly
> a.out
2+3*5
17
> a.out
2*3+5
16
```

# Precedences and Associativities in yacc I

General mechanism for resolving conflicts:

$$\begin{array}{c} \texttt{\%[left|right]}\ \mathit{Operators}_1 \\ \vdots \\ \texttt{\%[left|right]}\ \mathit{Operators}_n \end{array}$$

- operators in one line have given associativity and same precedence
- precedence increases over lines

**Example 14.5**

```
%left '+' '-'
%left '*' '/'
%right '^'
```

`^` (right associative) binds stronger than `*` and `/` (left associative), which in turn bind stronger than `+` and `-` (left associative)

# Precedences and Associativities in yacc II

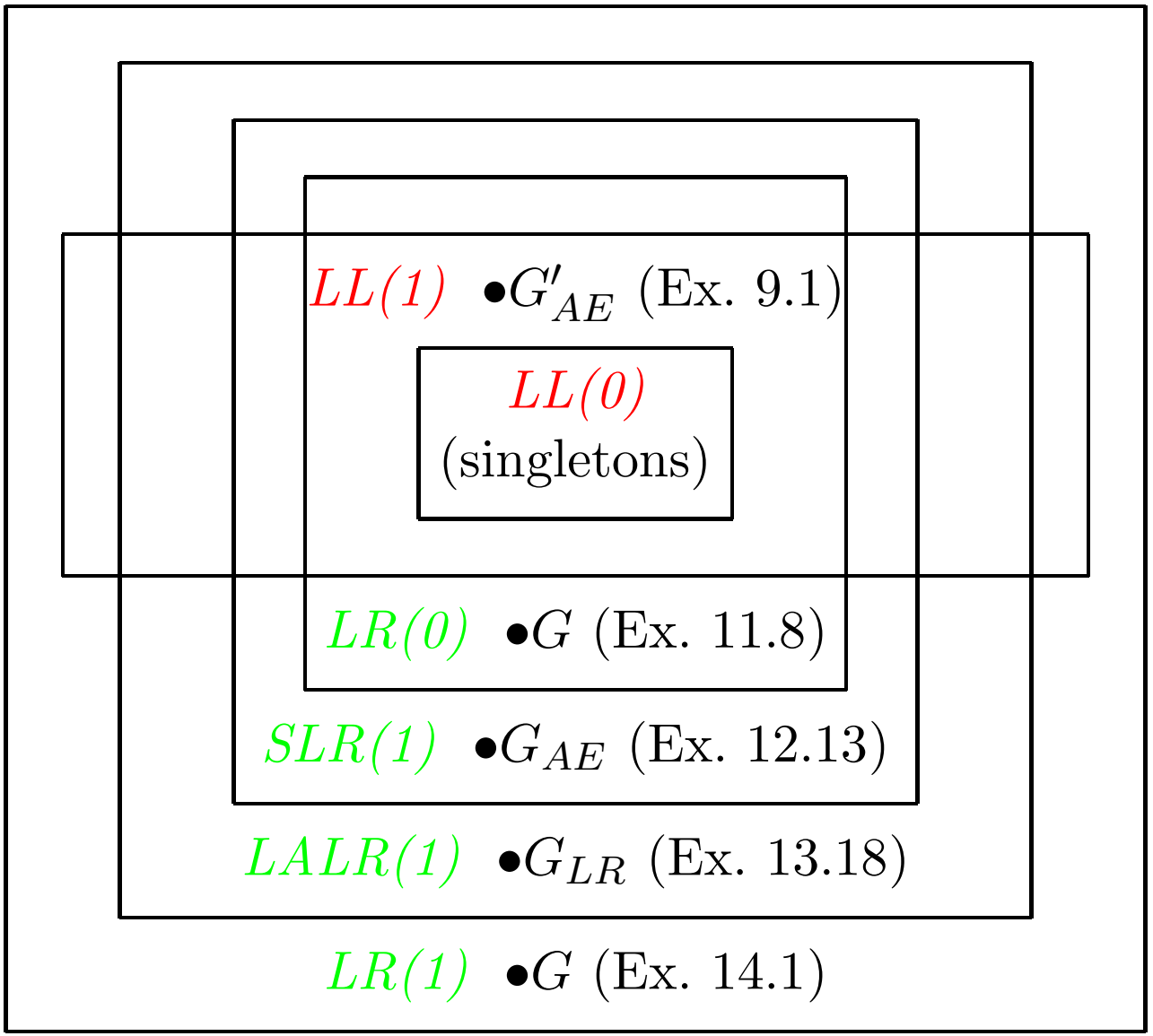
```
%{/* Ambiguous grammar for arithmetic expressions
     with precedences and associativities */
  #include <stdio.h>
  #include <ctype.h>
%}
%token DIGIT
%left '+'
%left '*'
%%
line   : expr '\n'  { printf("%d\n", $1); };
expr   : expr '+' expr    { $$ = $1 + $3; }
       | expr '*' expr    { $$ = $1 * $3; }
       | DIGIT            { $$ = $1; };
%%
yylex() {
  int c;
  c = getchar();
  if (isdigit(c)) {yylval = c - '0'; return DIGIT;}
  return c;
}
```

# Precedences and Associativities in yacc III

```
> yacc ambig-prio.y
> cc y.tab.c -ly
> a.out
2*3+5
11
> a.out
2+3*5
17
```

# Outline

# Overview of Grammar Classes

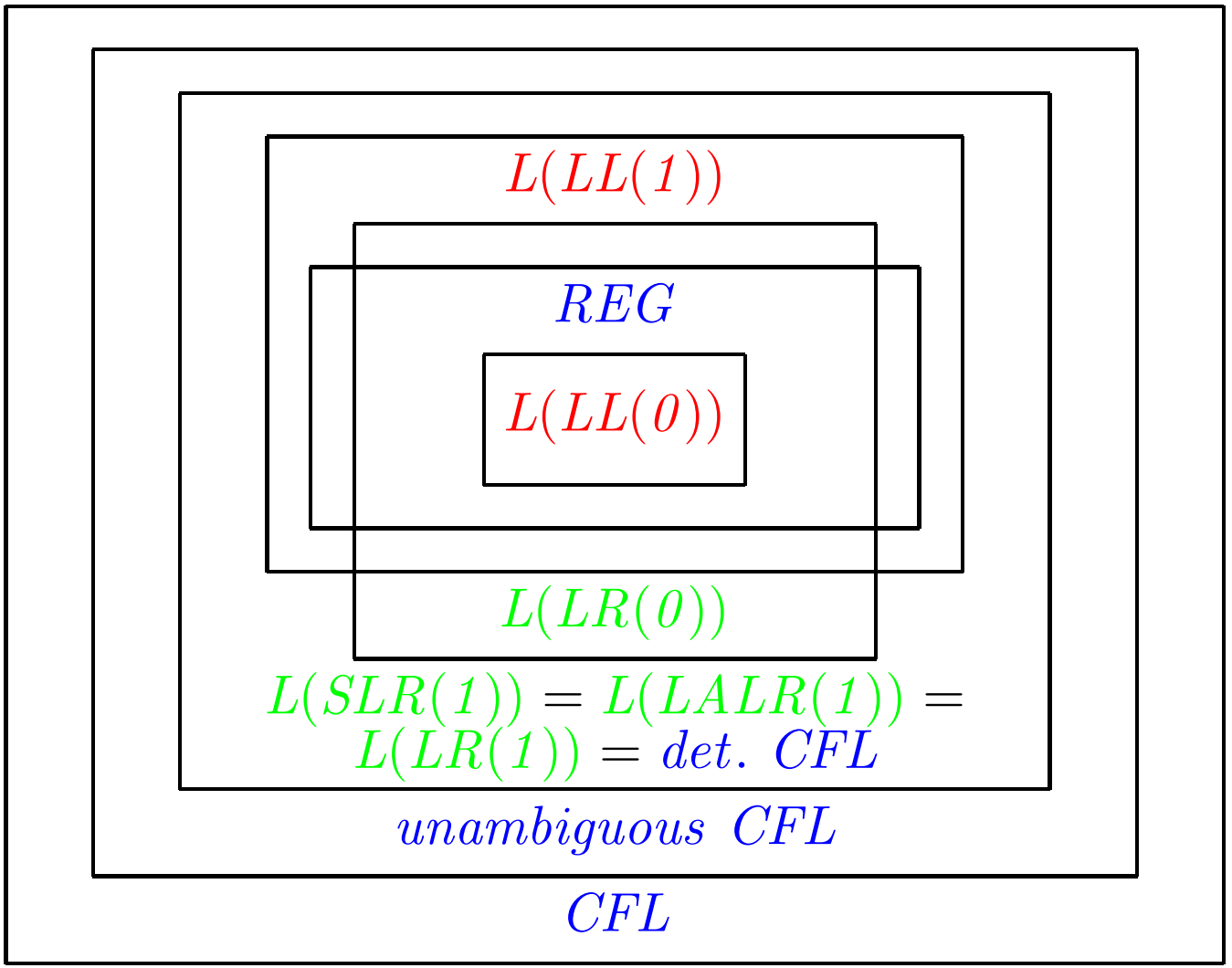



**Moreover:**

- $LL(k) \subsetneq LL(k+1)$ for every $k \in \mathbb{N}$
- $LR(k) \subsetneq LR(k+1)$ for every $k \in \mathbb{N}$
- $LL(k) \subseteq LR(k)$ for every $k \in \mathbb{N}$

# Overview of Language Classes

(cf. O. Mayer: *Syntaxanalyse*, BI-Verlag, 1978, p. 409ff)




**Moreover:**

- $L(LL(k)) \subsetneq L(LL(k+1)) \subsetneq L(LR(1))$ for every $k \in \mathbb{N}$
- $L(LR(k)) = L(LR(1))$ for every $k \geq 1$

# Outline

# LL and LR Parsing in Practice

In practice: use of $LL(1)$ or $LALR(1)$

Detailed comparison (cf. Fischer/LeBlanc: *Crafting a Compiler*, Benjamin/Cummings, 1988):

Simplicity : LL wins

- LL parsing technique easier to understand
- recursive-descent parser easier to debug than LALR action tables

Generality : LALR wins

- "almost" $LL(1) \subseteq LALR(1)$ (only pathological counterexamples)
- LL requires elimination of left recursion and left factorization

Semantic actions : (see semantic analysis) LL wins

- actions can be placed anywhere in LL parsers without causing conflicts
- in LALR: implicit $\varepsilon$-productions