Compiler Construction

Lecture 11: Syntax Analysis VII (Practical Issues)

Summer Semester 2016

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

https://moves.rwth-aachen.de/teaching/ss-16/cc/

Recap: $LR(1)$ and $LALR(1)$ Parsing

$LR(1)$ Items and Sets

Observation: not every element of $\mathrm{fo}(A)$ can follow every occurrence of A

$\Longrightarrow$ refinement of $LR(0)$ items by adding possible lookahead symbols

Definition ($LR(1)$ items and sets)

Let $G = \langle N, \Sigma, P, S \rangle \in CFG_\Sigma$ be start separated by $S' \to S$.

- If $S' \Rightarrow_r^* \alpha A a w \Rightarrow_r \alpha\beta_1\beta_2 a w$, then $[A \to \beta_1 \cdot \beta_2, a]$ is called an $LR(1)$ item for $\alpha\beta_1$.
- If $S' \Rightarrow_r^* \alpha A \Rightarrow_r \alpha\beta_1\beta_2$, then $[A \to \beta_1 \cdot \beta_2, \varepsilon]$ is called an $LR(1)$ item for $\alpha\beta_1$.
- Given $\gamma \in X^*$, $LR(1)(\gamma)$ denotes the set of all $LR(1)$ items for γ, called the $LR(1)$ set (or: $LR(1)$ information) of γ.
- $LR(1)(G) := \{LR(1)(\gamma) \mid \gamma \in X^*\}$.

$LR(1)$ Conflicts

Definition ($LR(1)$ conflicts)

Let $G = \langle N, \Sigma, P, S \rangle \in CFG_\Sigma$ and $I \in LR(1)(G)$.

- I has a shift/reduce conflict if there exist $A \to \alpha_1 a \alpha_2, B \to \beta \in P$ and $x \in \Sigma_\varepsilon$ such that

$$[A \to \alpha_1 \cdot a\alpha_2, x], [B \to \beta\cdot, a] \in I.$$

- I has a reduce/reduce conflict if there exist $x \in \Sigma_\varepsilon$ and $A \to \alpha, B \to \beta \in P$ with $A \neq B$ or $\alpha \neq \beta$ such that

$$[A \to \alpha\cdot, x], [B \to \beta\cdot, x] \in I.$$

Lemma

$G \in LR(1)$ iff no $I \in LR(1)(G)$ contains conflicting items.

The $LR(1)$ Action Function

Definition ($LR(1)$ action function)

The $LR(1)$ action function

$$\mathrm{act} : LR(1)(G) \times \Sigma_\varepsilon \to \{\text{red } i \mid i \in [p]\} \cup \{\text{shift}, \text{accept}, \text{error}\}$$

is defined by

$$\mathrm{act}(I, x) := \begin{cases} \text{red } i & \text{if } i \neq 0, \pi_i = A \to \alpha \text{ and } [A \to \alpha\cdot, x] \in I \\ \text{shift} & \text{if } [A \to \alpha_1 \cdot x\alpha_2, y] \in I \text{ and } x \in \Sigma \\ \text{accept} & \text{if } [S' \to S\cdot, \varepsilon] \in I \text{ and } x = \varepsilon \\ \text{error} & \text{otherwise} \end{cases}$$

Corollary

For every $G \in CFG_\Sigma$, $G \in LR(1)$ iff its $LR(1)$ action function is well defined.

$LR(0)$ Equivalence

Observation: potential redundancy by containment of $LR(0)$ sets in $LR(1)$ sets (cf. Corollary 10.3)

Definition ($LR(0)$ equivalence)

Let $\mathrm{lr}_0 : LR(1)(G) \to LR(0)(G)$ be defined by

$$\mathrm{lr}_0(I) := \{[A \to \beta_1 \cdot \beta_2] \mid [A \to \beta_1 \cdot \beta_2, x] \in I\}.$$

Two sets $I_1, I_2 \in LR(1)(G)$ are called $LR(0)$-equivalent (notation: $I_1 \sim_0 I_2$) if $\mathrm{lr}_0(I_1) = \mathrm{lr}_0(I_2)$.

$LALR(1)$ 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 10.16, $|LALR(1)(G)| = |LR(0)(G)|$
(but $LALR(1)$ sets provide additional lookahead information)

Ambiguous Grammars

Reminder (Definition 5.5): 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 11.1

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 $\mathrm{first}_k(v) = \mathrm{first}_k(v)$ for every $v \in \Sigma^*$, Definition 8.8 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 11.2 (Simple arithmetic expressions)

$G : E' \to E \quad (0) \qquad E \to E{+}E \mid E{*}E \mid \mathtt{a} \quad (1,2,3)$

Precedence: $* > +$ Associativity: left (thus: $\mathtt{a{+}a{*}a{+}a} := \mathtt{(a{+}(a{*}a)){+}a}$)

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

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

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

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

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

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

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

Conflicts: I_1: $SLR(1)$-solvable (reduce on ε, shift on $+/*$)

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

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

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

Bottom-Up Parsing of Ambiguous Grammars II

Example 11.3 (“Dangling `else`”)

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

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

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

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

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

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

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

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

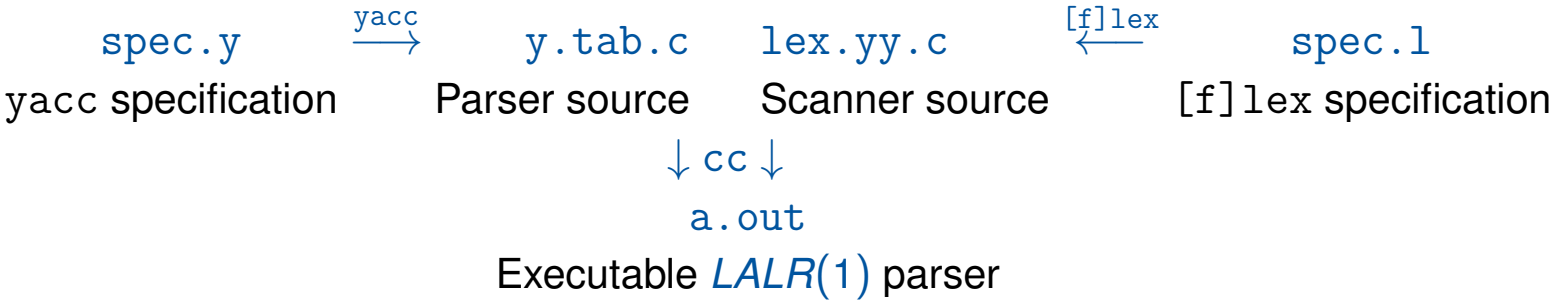
$I_6 := LR(0)(\mathtt{i}S\mathtt{e}S) : \quad [S \to \mathtt{i}S\mathtt{e}S\cdot]$

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

Solution (1): $\mathrm{act}(I_4, \mathtt{e}) :=$ shift

The `yacc` and `bison` Tools

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



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

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

`bison` : upward-compatible GNU implementation of `yacc`
(more flexible w.r.t. file names, ...)

`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

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

- 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 9.11) */
  #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 arithmetic expressions (Example 11.2) */
  #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 11.3) correctly
- also adequate for strong following weak operator (∗ after +; Example 11.2) and for right-associative operators
- not appropriate for weak following strong operator and for left-associative binary operators ($\Longrightarrow$ reduce; see Example 11.2)

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:

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

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

Example 11.4

```
%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

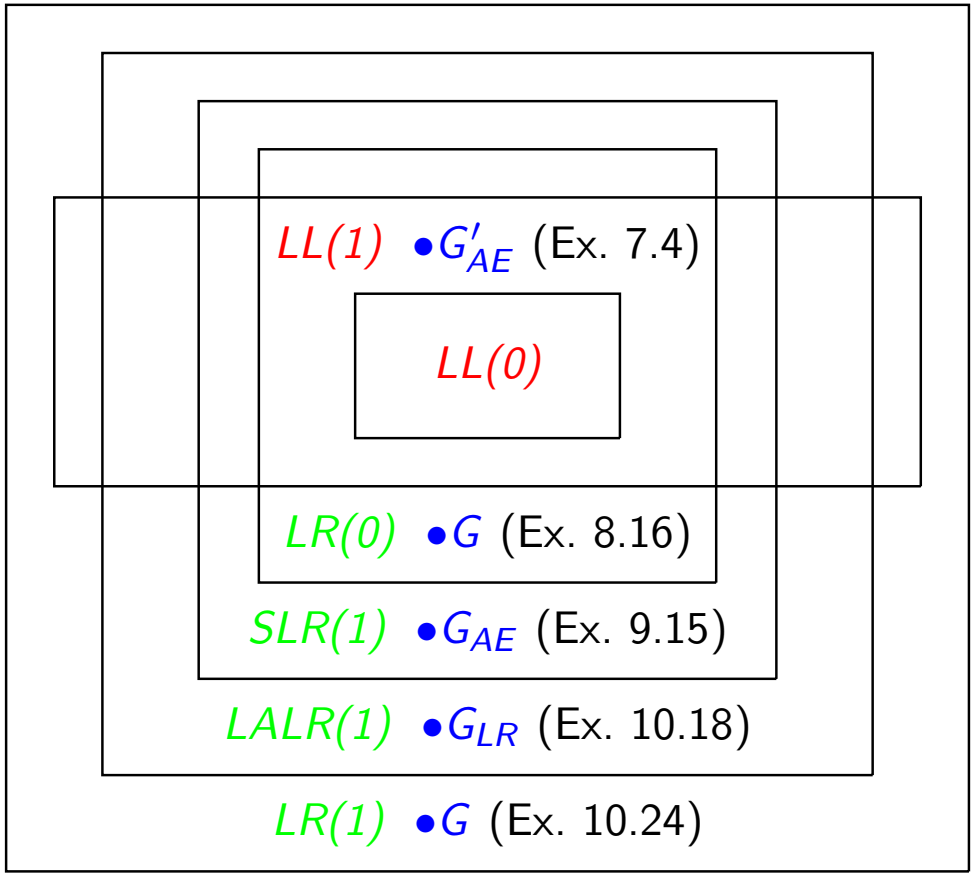
```
%{/* Ambiguous grammar for arithmetic expressions
     with precedences and associativities */
  #include <stdio.h>
  #include <ctype.h>
%}
%token DIGIT
%left '+'
%left '*'
%%
line   : expr '\n'  { printf("%d\n", $1); };
expr   : expr '+' expr    { $$ = $1 + $3; }
       | expr '*' expr    { $$ =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

Expressiveness of LL and LR Grammars

Overview of Grammar Classes



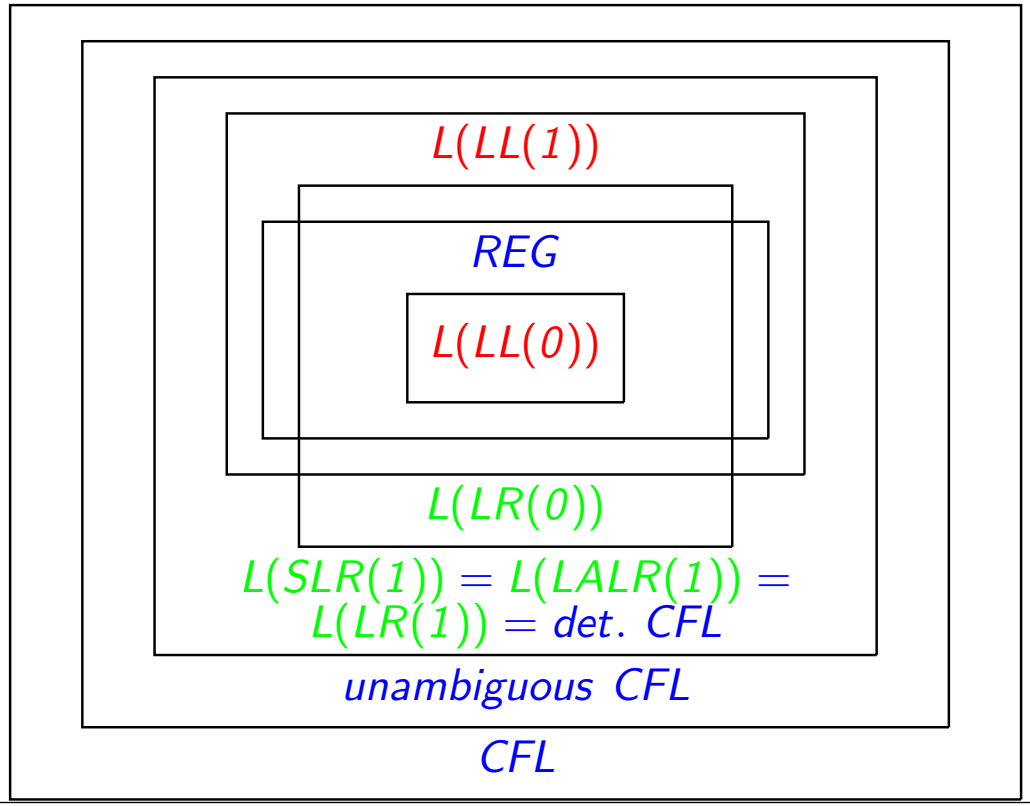
Moreover:

- $LL(k) \subsetneq LL(k+1)$ 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 $k \in \mathbb{N}$

Expressiveness of LL and LR Grammars

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 $k \geq 1$

Expressiveness of LL and LR Grammars

Why $REG \not\subseteq L(LR(0))$?

Definition 11.5

A language $L \subseteq \Sigma^*$ is called prefix-free if $L \cap L \cdot \Sigma^+ = \emptyset$, i.e., if no proper prefix of an element of L is again in L.

Lemma 11.6

$G \in LR(0) \implies L(G)$ *prefix-free*

Proof.

on the board □

Corollary 11.7

$\{a, aa\} \in REG \setminus L(LR(0))$

Conjecture: $L \in REG \setminus L(LR(0)) \implies L(G)$ not prefix-free?

Expressiveness of LL and LR Grammars

Why $REG \subseteq L(LL(1))$?

Lemma 11.8 (cf. Lecture 2)

Every $L \in REG$ can be recognized by a DFA.

Lemma 11.9

Every DFA can be transformed into an equivalent $LL(1)$ grammar.

Proof.

on the board □

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 ε-productions
  $\Longrightarrow$ may generate conflicts

Error handling: LL wins
- top-down approach provides context information
  $\Longrightarrow$ better basis for reporting and/or repairing errors

Parser size: comparable
- LL: action table
- LALR: action/goto table