Formal Languages and Compilers
Lecture VII—Part 4: Syntactic Analysis

Alessandro Artale

Free University of Bozen-Bolzano
Faculty of Computer Science – POS Building, Room: 2.03
artale@inf.unibz.it
http://www.inf.unibz.it/~artale/

Formal Languages and Compilers — BSc course

2016/17 – First Semester
Summary of Lecture IV—Part 4

- **Inadequacy of SLR Parsing**
- **LR(1) Parsing**
  - Closure and Goto Operations, Canonical Collection
  - LR(1) Parsing Tables
- **LALR(1) Parsing**
  - Conflicts in LALR Parsers
  - LALR(1) Parsing Tables
  - LALR Vs. LR Parsers
- **Dealing with Ambiguous Grammars**
Example. Consider the following Grammar and the Canonical Collection:

\[
\begin{align*}
I_0 : & \quad S' \rightarrow .S \\
S & \rightarrow .L = R \\
S & \rightarrow .R \\
L & \rightarrow .*R \\
L & \rightarrow .id \\
R & \rightarrow .L \\
I_5 : & \quad L \rightarrow .id. \\
I_6 : & \quad S \rightarrow L = .R \quad R \rightarrow .L \\
L & \rightarrow .*R \\
L & \rightarrow .id \\
I_7 : & \quad L \rightarrow .*R. \\
L & \rightarrow L. \\
I_8 : & \quad R \rightarrow L. \\
I_9 : & \quad S \rightarrow L = R. \\
I_1 : & \quad S' \rightarrow S. \\
I_2 : & \quad S \rightarrow L = R. \\
I_3 : & \quad S \rightarrow R. \\
I_4 : & \quad L \rightarrow .*R. \\
L & \rightarrow .L \\
L & \rightarrow .*R \\
L & \rightarrow .id \\
\end{align*}
\]
Consider now the set of items in $I_2$:

1. If we consider the first item, then, $\text{action}[2, \text{=}] = \text{shift } 6$.
2. If we consider the second item, since $\text{=} \in \text{Follow}(R)$, then, $\text{action}[2, \text{=}] = \text{reduce } R \rightarrow L$.

- **Shift/Reduce Conflict.** There is both a shift and a reduce action thus the entry for $\text{action}[2, \text{=}]$ is multiply defined!
- The SLR parsing technique is not powerful enough!
The problem can be modeled as follow:

1. The stack contains the string $\beta\alpha$;
2. The item set $I_i$ contains the item $[A \rightarrow \alpha.]$;
3. Lookahead symbol is $a$, and $a \in \text{Follow}(A)$.

**Problem.** The state $s_i$ is on top of the stack BUT reducing with $A \rightarrow \alpha$ is such that $\beta A$ cannot be followed by “$a$” in a right-sentential form.

**Remark.** In general, $\text{Follow}(A)$ is not limited to right-most derivations.
If we are in state $I_2$ with lookahead “=” then the stack contains $0L2$ with $L$ on top. If we reduce with $R \rightarrow L$ then the stack is $0R3$. In state $I_3$ we can only reduce with $S \rightarrow R$ and lookahead $. Since the current lookahead is “=” then the parsing fails: $\text{action}[3,=] = \text{error}$, and the parser must Backtrack!
Summary

- Inadequacy of SLR Parsing
- LR(1) Parsing
  - Closure and Goto Operations, Canonical Collection
  - LR(1) Parsing Tables
- LALR(1) Parsing
  - Conflicts in LALR Parsers
  - LALR(1) Parsing Tables
  - LALR Vs. LR Parsers
- Dealing with Ambiguous Grammars
Towards LR Parsing: A Solution to SLR failure

- Add more information in the state to rule out invalid reductions.
- Each state of an LR(1) Parser indicates what symbol can follow a handle for which there is a possible reduction.

**Definition.** An LR(1) Item is then a pair \([A \rightarrow \alpha. \beta, a]\), where \(a \in V_T\) is called the **Lookahead of the Item**. Furthermore, \(a \in \text{Follow}(A)\).
LR(1) Items

- \([A \rightarrow \alpha \cdot \beta, a]\) describes a “context” of the parser:
  - We are trying to find an \(A\) followed by an \(a\), and
  - We have \(\alpha\) already on top of the stack
  - Thus we need to see next something derivable from \(\beta a\) along right-most derivations.

- The lookahead has no effect in items of the form \([A \rightarrow \alpha \cdot \beta, a]\), with \(\beta \neq \epsilon\), BUT

- On items \([A \rightarrow \alpha, a]\), the parser reduces with \(A \rightarrow \alpha\) only if the next input symbol is \(a\).
Summary

- Inadequacy of SLR Parsing
- LR(1) Parsing
  - Closure and Goto Operations, Canonical Collection
  - LR(1) Parsing Tables
- LALR(1) Parsing
  - Conflicts in LALR Parsers
  - LALR(1) Parsing Tables
  - LALR Vs. LR Parsers
- Dealing with Ambiguous Grammars
To construct the Collection of sets of Items we proceed as in the construction of the LR(0) Canonical Collection but with new closure and goto operations.

**Algorithm.** *Closure(I).*
Let \( I \) be a set of LR(1) items for an augmented Grammar \( G' \), then \( \text{closure}(I) \) is a set of items such that:

1. Initially every item in \( I \) is added to \( \text{closure}(I) \);
2. If \( [A \rightarrow \alpha. B \beta, a] \in \text{closure}(I) \) and \( B \rightarrow \gamma \), then we add the item \( [B \rightarrow . \gamma, b] \) to \( \text{closure}(I) \), where \( b \in \text{First}(\beta a) \). Go to step 1 until no more items can be added to \( \text{closure}(I) \).

**Intuition.** Considering the addition of the productions involving \( B \) the parser must take into account the “context” of \( B \).
The Goto Operation and LR(1) Item Collection

**Definition. (Goto)** If $I$ is a set of items and $X$ is a grammar symbol, then, \( \text{goto}(I, X) \) is the closure of the set of all items \([A \to \alpha X. \beta, a]\) such that \([A \to \alpha. X \beta, a]\) is in $I$.

**LR(1) Item Collection.** We proceed as in the construction of the LR(0) Canonical Collection but with new closure and goto operations. The initial condition is: \( C = \{\text{closure}(\{[S' \to S, $]\})\} \).
**Example.** Build the LR(1) Item Collection for the following augmented Grammar:

\[
S' \rightarrow S, S \\
S \rightarrow CC, S \\
S \rightarrow cC, $c/d$ \\
S \rightarrow d, $c/d$ \\
C \rightarrow cC, S \\
C \rightarrow cC, $c/d$ \\
C \rightarrow d, $c/d$ \\
C \rightarrow cC, $c/d$ \\
C \rightarrow d, $c/d$ \\
S \rightarrow S', $S$ \\
C \rightarrow cC, $c/d$ \\
C \rightarrow d, $c/d$ \\
S \rightarrow CC, $S$ \\
C \rightarrow cC, $c/d$ \\
C \rightarrow d, $c/d$ \\
S \rightarrow cC, $c/d$ \\
C \rightarrow d, $c/d$ \\
S \rightarrow S', $S$ \\
C \rightarrow cC, $c/d$ \\
C \rightarrow d, $c/d$
\]
Inadequacy of SLR Parsing

LR(1) Parsing
  - Closure and Goto Operations, Canonical Collection
  - LR(1) Parsing Tables

LALR(1) Parsing
  - Conflicts in LALR Parsers
  - LALR(1) Parsing Tables
  - LALR Vs. LR Parsers

Dealing with Ambiguous Grammars
Algorithm. LR(1) Parsing Tables Action and Goto.

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the LR(1) item collection for the augmented grammar $G'$.

2. To each item set $I_i$ we create a new state $s_i$. Then the action table is:
   - $\text{action}[s_i, a] = \text{"shift } s_j\text{"}$, if $[A \rightarrow \alpha \cdot a \beta, b] \in I_i$, and $\text{goto}(I_i, a) = I_j$.
   - $\text{action}[s_i, a] = \text{"reduce } A \rightarrow \alpha\text{"}$, for all $[A \rightarrow \alpha \cdot a] \in I_i$. Here $A \neq S'$.
   - $\text{action}[s_i, \$$] = \text{"accept\"}$, if $[S' \rightarrow \cdot S, \$$] \in I_i$.

3. $\text{goto}[s_i, A] = s_j$, if $\text{goto}(I_i, A) = I_j$.

4. All the entries not defined by rules (2) and (3) are made "error".

5. The initial state is the one constructed from the closure of $[S' \rightarrow \cdot S, \$$]$. 
The parsing table produced by the above algorithm is called the **Canonical LR(1) Parsing Table**;

A parser using this table is called a **Canonical LR(1) Parser**;

If the table has no multiply-defined entries then the Grammar is called an **LR(1) Grammar**.
Summary

- Inadequacy of SLR Parsing
- LR(1) Parsing
  - Closure and Goto Operations, Canonical Collection
  - LR(1) Parsing Tables
- LALR(1) Parsing
  - Conflicts in LALR Parsers
  - LALR(1) Parsing Tables
  - LALR Vs. LR Parsers
- Dealing with Ambiguous Grammars
LALR stands for LookAhead-LR Parser.

It is often used in practice since the parsing tables are considerably smaller than the canonical LR tables.

(SLR and LALR) Vs. LR. The comparison is in term of size. For a language like Pascal we go from hundreds of states to thousand of states.

Main Idea: Merge the states (Item sets) whose items differ only in the lookahead.

- We say that such states have the same core.
Example. The Grammar:

\[
S' \rightarrow S \\
S \rightarrow CC \\
C \rightarrow cC | d
\]

Has items \(I_3 - I_6, I_4 - I_7,\) and \(I_8 - I_9\) with the same core. By merging items with the same core we obtain:

\[
I_{36} : \quad C \rightarrow .c.C, \quad c/d/$$ \\
I_{47} : \quad C \rightarrow .d, \quad c/d/$$
\]

Consider the merged state \(I_{47}\). The LALR parser will reduce \(d\) to \(C\) on any input, even in situations where the original LR parser would generate an error!

In any case, the LALR parser will communicate the same error but at a later stage.
Summary

- Inadequacy of SLR Parsing
- LR(1) Parsing
  - Closure and Goto Operations, Canonical Collection
  - LR(1) Parsing Tables
- LALR(1) Parsing
  - Conflicts in LALR Parsers
  - LALR(1) Parsing Tables
  - LALR Vs. LR Parsers
- Dealing with Ambiguous Grammars
Theorem. Starting from an LR(1) Grammar and merging states with the same core we can only introduce reduce/reduce conflicts.

We first show that no shift/reduce conflicts are present. Suppose there is a merged state with a shift/reduce conflict on input \( a \):

\[
\begin{align*}
I_{LALR} : \quad A & \rightarrow \alpha., \quad a/\ldots \\
B & \rightarrow \beta.a\gamma, \quad b/\ldots
\end{align*}
\]

Then, there is an LR(1) state such that:

\[
\begin{align*}
I_{LR} : \quad A & \rightarrow \alpha., \quad a \\
B & \rightarrow \beta.a\gamma, \quad c \text{ (for some } c) \\
\end{align*}
\]

And \( I_{LR} \) presents a shift/reduce conflict on input \( a \). Absurd!
Introducing Conflict by Merging States (Cont.)

We now show how it is possible to introduce reduce/reduce conflicts.

Consider the following LR(1) states:

\[
\begin{align*}
I_i : & \quad A \rightarrow c.\ , \ d \\
B \rightarrow & \quad c.\ , \ e
\end{align*}
\]

\[
\begin{align*}
I_j : & \quad A \rightarrow c.\ , \ e \\
B \rightarrow & \quad c.\ , \ d
\end{align*}
\]

Neither of these sets generates a conflict, however, their merge:

\[
\begin{align*}
I_{ij} : & \quad A \rightarrow c.\ , \ d/e \\
B \rightarrow & \quad c.\ , \ d/e
\end{align*}
\]

generates a reduce/reduce conflict on both input d and e.

**Note.** In case of a reduce/reduce conflict the Grammar is no more LALR. Thus, LALR is a simpler Grammar than an LR Grammar.
Summary

- Inadequacy of SLR Parsing
- LR(1) Parsing
  - Closure and Goto Operations, Canonical Collection
  - LR(1) Parsing Tables
- LALR(1) Parsing
  - Conflicts in LALR Parsers
  - LALR(1) Parsing Tables
  - LALR Vs. LR Parsers
- Dealing with Ambiguous Grammars
The algorithm we present serves primarily for defining LALR Grammars—there are more efficient techniques.

Algorithm. LALR(1) Parsing Tables Action and Goto.

1. Construct \( C = \{ I_0, I_1, \ldots, I_n \} \), the LR(1) item collection for the augmented grammar \( G' \).

2. Find all LR(1) states having the same core and merge them. Let \( C' = \{ J_0, J_1, \ldots, J_m \} \) the resulting item collection.

3. To each item set \( J_i \) we create a new state \( s_i \). Then the action table is constructed as in case of LR(1) parsers.

4. The goto function (and thus, the goto table) is constructed as follows. If \( J = I_1 \cup \ldots \cup I_k \), then the cores of \( \text{goto}(I_1, X), \ldots, \text{goto}(I_k, X) \) are all the same. Let \( K \) the union of all sets of items having the same core as \( \text{goto}(I_1, X) \), i.e., \( K = \bigcup_{i=1,\ldots,k} \text{goto}(I_i, X) \), then \( \text{goto}(J, X) = K \).
If there is a reduce/reduce conflict in the above construction the Grammar is not LALR(1);

The parsing table produced by the above algorithm is called the \textit{LALR(1) Parsing Table};

A parser using this table is called an \textit{LALR(1) Parser}.
**Example.** The Grammar:

\[
\begin{align*}
S' & \rightarrow S \\
S & \rightarrow CC \\
C & \rightarrow cC \mid d
\end{align*}
\]

Has the following LALR(1) Item Collection: \( C = \{l_0, l_1, l_2, l_5, l_{36}, l_{47}, l_{89}\} \), where:

\[
\begin{align*}
l_{36} & : C \rightarrow c. C, \ c/d/$ \\
l_{47} & : C \rightarrow d., \ c/d/$ \\
C \rightarrow & .cC, \ c/d/$ \\
C \rightarrow & .d, \ c/d/$ \\
l_{89} & : C \rightarrow cC., \ c/d/$
\end{align*}
\]

The LALR Parsing Table is:

\[
\begin{array}{cccc}
\text{c} & \text{d} & \$ & \text{S} \text{ C} \\
0 & s36 & s47 & 1 \ 2 \\
1 & & acc & \\
2 & s36 & s47 & 5 \\
36 & s36 & s47 & 89 \\
47 & r3 & r3 & r3 \\
5 & & r1 & \\
89 & r2 & r2 & r2
\end{array}
\]
Summary

- Inadequacy of SLR Parsing
- LR(1) Parsing
  - Closure and Goto Operations, Canonical Collection
  - LR(1) Parsing Tables
- LALR(1) Parsing
  - Conflicts in LALR Parsers
  - LALR(1) Parsing Tables
  - LALR Vs. LR Parsers
- Dealing with Ambiguous Grammars
Behavior of LR Vs. LALR

- On a correct input both parsers make the same sequence of shift and reduce, only the state’s names change.
- On erroneous input the LALR parser makes more moves (reductions) than the LR parser.
Example. Consider as input the string “ccd”.

- The LR parser will push onto the stack: \texttt{0c3c3d4}.
  Ending with an error in state 4 and current input $.$.

- The LALR parser will push onto the stack: \texttt{0c36c36d47}.
  State 47 with input $\$ \text{ reduces with } C \rightarrow d \text{ and the stack is: } \texttt{0c36c36C89}.
  State 89 with input $\$ \text{ reduces with } C \rightarrow cC \text{ and the stack is: } \texttt{0c36C89}.
  A similar reduction applies and the stack is: \texttt{0C2}.
  Ending finally with an error in state 2 and current input $.$.
Summary of Lecture IV—Part 4

- Inadequacy of SLR Parsing
- LR(1) Parsing
  - Closure and Goto Operations, Canonical Collection
  - LR(1) Parsing Tables
- LALR(1) Parsing
  - Conflicts in LALR Parsers
  - LALR(1) Parsing Tables
  - LALR Vs. LR Parsers
- Dealing with Ambiguous Grammars
• Ambiguous Grammars provide a shorter and more natural specification.
• This is reflected by a parser requiring a minor number of states.
• Ambiguous Grammars are not LR:
  • A parsing table will have multiply-defined entries;
  • A parsing table will contain reduce/reduce and/or shift/reduce conflicts.

• To deal with ambiguous grammars we use disambiguating rules that eliminate all the conflicts allowing for only one Parse-Tree.
The following ambiguous Grammar for arithmetic expressions does not specify the associativity and the precedence of \(+\), \(*\):

\[
E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}
\]

The unambiguous Grammar:

\[
\begin{align*}
E & \rightarrow E + T | T \\
T & \rightarrow T * F | F \\
F & \rightarrow (E) | \text{id}
\end{align*}
\]

Enforces precedence of \(*\) over \(+\), and left-associativity of \(+\) and \(*\).

**Note.** The parser for the unambiguous Grammar is less efficient:

- The time spent for reducing the unit productions \(E \rightarrow T\) and \(T \rightarrow F\) has the sole function to enforce precedence.
Let us consider the following LR(0) items for the ambiguous augmented Grammar (see the Book, Sect. 4.8, for the complete set of states):

\[ I_7 : \quad E \rightarrow E + E, \quad I_8 : \quad E \rightarrow E * E. \]

\[ E \rightarrow E + E, \quad E \rightarrow E * E. \]

Follow(\(E\)) = \{\$, +, \*\}

State \(I_7\) generates a conflict between “reduce with \(E \rightarrow E + E\)”, and “shift with input + and *”.

State \(I_8\) generates a conflict between “reduce with \(E \rightarrow E * E\)”, and “shift with input + and *”.

Both Conflicts can be solved using the Precedence and Associativity for the operations + and *.
Consider the input id + id * id and the following configuration:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>0E1 + 4E7</td>
<td>*id$</td>
</tr>
</tbody>
</table>

Assuming that * takes precedence over +, the parser should **shift** * onto the stack instead of **reducing** with $E \rightarrow E + E$.

Consider the input id + id + id and the following configuration:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>0E1 + 4E7</td>
<td>+id$</td>
</tr>
</tbody>
</table>

Assuming that + is *left-associative*, the parser should **reduce** with $E \rightarrow E + E$ instead of **shift** + onto the stack.
**Summary.** Precedence and Associativity uniquely determine the actions of the parser eliminating the ambiguity:

1. **Left-Associativity of +**: $\text{action}(7, +) = \text{reduce}$ with $E \rightarrow E + E$
2. **Left-Associativity of ***: $\text{action}(8, *) = \text{reduce}$ with $E \rightarrow E * E$
3. **Precedence of * over +**: $\text{action}(7, *) = \text{shift}$ *
4. **Precedence of * over +**: $\text{action}(8, +) = \text{reduce}$ with $E \rightarrow E * E$
Consider the following Grammar for IF-THEN-ELSE statements:

\[ S' \rightarrow S \]
\[ S \rightarrow iSeS | iS | a \]

Where, \( i \) stands for “if expr then”, \( e \) stands for else, and \( a \) stands for “all other productions”.

Let us consider the following LR(0) item for this ambiguous Grammar (see the Book for the complete set of states):

\[ l_4: \quad S \rightarrow iS. eS \quad S \rightarrow iS. \]

Follow(\( S \)) = \{ $, \text{else} \}

Assuming that the stack contains: “if expr then stmt” (\( iS \)) with state 4 on top, and the next input is else there is a shift/reduce conflict.

**Solution.** else *must match the last if*: \( \text{action}(4, \text{else}) = \text{shift} \) else.

Since Yacc solves shift/reduce conflicts in favor of shift the dangling-else is handled correctly.
Summary of Lecture IV—Part 4

- Inadequacy of SLR Parsing
- LR(1) Parsing
  - Closure and Goto Operations, Canonical Collection
  - LR(1) Parsing Tables
- LALR(1) Parsing
  - Conflicts in LALR Parsers
  - LALR(1) Parsing Tables
  - LALR Vs. LR Parsers
- Dealing with Ambiguous Grammars