# 7 Parsing 3 - Shift reduce, SLR parsing

All LR parsers (SLR, canonical LR and LALR) work in the same way. They differ in the range of grammars which they can use and the manner in which their driver tables are generated.

As with operator precedence parsers, the basic model of an LR parser contains a stack of grammar symbols and a string waiting to be input. Unlike the operator precedence stack, however, the LR parser's stack has a state symbol associated with each grammar symbol. In fact only the states are significant, the grammar symbols do not drive the parser.

The actions of the parser are based on using the combination of the topmost state on the stack and the next input symbol to index the action table. Each entry in the action table specifies one of

1. shift and add state i to the stack
2. reduce by production j
3. accept
4. error

If we start with configuration

$$\underbrace{(S_0\ X_1\ S_1\ X_2\ S_2 \ldots X_m\ S_m}_{\text{stack}} \qquad \underbrace{a_i\ a_{i+1} \ldots a_n\$)}_{\text{input stream}}$$

we may encounter the following:

1.

$$\begin{array}{rcl} Action[S_m, a_i] & = & \text{shift.} \\ & \downarrow & \\ \underbrace{(S_0\ X_1\ S_1\ X_2\ S_2 \ldots X_m\ S_m\ a_i\ S_k}_{\text{stack}} & & \underbrace{a_{i+1} \ldots a_n\$)}_{\text{input stream}} \end{array}$$

The new state $S_k$ is determined by a function $GOTO$, which takes a state and a symbol in the same way as $Action$. Thus $GOTO[S_m, a_i] = S_k$. The $GOTO$ values form a second part of the parser driver tables.

2.

$$\begin{array}{rcl} Action[S_m, a_i] & = & \text{reduce} A \rightarrow \text{b} \\ & \downarrow & \\ \underbrace{(S_0\ X_1\ S_1\ X_2\ S_2 \ldots X_{m-r}\ S_{m-r}\ A\ S_k}_{\text{stack}} & & \underbrace{a_i\ a_{i+1} \ldots a_n\$)}_{\text{input stream}} \end{array}$$

The top $r$ symbols, where $r$ is the length of $b$, are popped from the stack and $A$ is pushed. The state $S_k$ is pushed, where $S = GOTO[S_{m-r}, A]$, i.e. as if, after the popping, $A$ was being shifted. The input is unchanged.

3.

$$Action[S_m, a_i] \quad = \quad \text{accept.}$$

Parsing is complete.

4.

$$Action[S_m, a_i] \quad = \quad \text{error}$$

An error handling routine is called.

**Example** An LR grammar for expressions is:

$$\begin{aligned}
exp &\rightarrow exp + term && (1)\\
exp &\rightarrow term && (2)\\
term &\rightarrow term * factor && (3)\\
term &\rightarrow factor && (4)\\
factor &\rightarrow ( exp ) && (5)\\
factor &\rightarrow \text{id} && (6)
\end{aligned}$$

The numbering of productions defines the reduce actions in the following parse table.

| | *Action* | | | | | | *GOTO* | | |
|---|---|---|---|---|---|---|---|---|---|
| State | id | + | * | ( | ) | $ | *exp* | *term* | *factor* |
| 0 | s5 | | | s4 | | | 1 | 2 | 3 |
| 1 | | S6 | | | | acc | | | |
| 2 | | r2 | S7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | s5 | | | s4 | | | 8 | 2 | 3 |
| 5 | | r6 | r6 | | r6 | r6 | | | |
| 6 | s5 | | | s4 | | | | 9 | 3 |
| 7 | s5 | | | s4 | | | | | 10 |
| 8 | | s6 | | | s11 | | | | |
| 9 | | r1 | S7 | | r1 | r1 | | | |
| 10 | | r3 | r3 | | r3 | r3 | | | |
| 11 | | r5 | r5 | | r5 | r5 | | | |

For terminal $a_i$, the value $GOTO[S_m, a_i]$ follows the *Action* `s` or `r` in the table. For non-terminals it is found in a separate part of the table.

The *GOTO* values embody a finite state automaton which parses this grammar.

Consider parsing the expression `id1 * id2 + id3`

| | Stack | Input | Action |
|---|---|---|---|
| 1 | 0 | id1 * id2 + id3 $ | Shift 5 |
| 2 | 0 id1 5 | * id2 + id3 $ | reduce by 6 |
| 3 | 0 factor 3 | * id2 + id3 $ | reduce by 4 |
| 4 | 0 term 2 | * id2 + id3 $ | Shift 7 |
| 5 | 0 term 2 * 7 | id2 + id3 $ | Shift 5 |
| 6 | 0 term 2 * 7 id2 5 | + id3 $ | reduce by 6 |
| 7 | 0 term 2 * 7 factor 10 | + id3 $ | reduce by 3 |
| 8 | 0 term 2 | + id3 $ | reduce by 2 |
| 9 | 0 exp 1 | + id3 $ | Shift 6 |
| 10 | 0 exp 1 + 6 | id3 $ | Shift 5 |
| 11 | 0 exp 1 + 6 id3 5 | $ | reduce by 6 |
| 12 | 0 exp 1 + 6 factor 3 | $ | reduce by 4 |
| 13 | 0 exp 1 + 6 term 9 | $ | reduce by 1 |
| 14 | 0 exp 1 | $ | accept |

The remaining questions are concerned with defining an acceptable LR grammar and generating a parse table from it.

### Generating SLR parser tables

The generation of any LR parser's tables is totally mechanical. Unfortunately, it is also time consuming and error prone when performed by hand for large (realistic) languages. This meant that little use was made of these techniques until parser generators became available.

The first step in generating an SLR parser's tables is to generate the set of $LR(0)$ items for that grammar. The grammar is first augmented by the additional production.

$$S' \rightarrow S$$

where $S$ is the non-terminal defined in the original start production. The end of parsing is defined by the reduction of $S$ to $S'$. Augmenting our expression grammar gives us

$$\begin{aligned}
exp' &\rightarrow exp \quad (1)\\
exp &\rightarrow exp + term \quad (2)\\
exp &\rightarrow term \quad (3)\\
term &\rightarrow term * factor \quad (4)\\
term &\rightarrow factor \quad (5)\\
factor &\rightarrow (\ exp\ ) \quad (6)\\
factor &\rightarrow \text{id} \quad (7)
\end{aligned}$$

An item is a production rule with a dot between two symbols or at the start or finish. This may be thought of as indicating the position reached in parsing according to that production. Thus, from

$$exp \rightarrow exp + term$$

we get the $LR(0)$ items:

$$
\begin{array}{lcll}
exp & \rightarrow & \cdot\ exp\ +\ term & <1,0> \\
exp & \rightarrow & exp\ \cdot\ +\ term & <1,1> \\
exp & \rightarrow & exp\ +\ \cdot\ term & <1,2> \\
exp & \rightarrow & exp\ +\ term\ \cdot & <1,3>
\end{array}
$$

The SLR parser is based on a deterministic finite state automaton which recognises viable prefixes. A viable prefix is, formally, a prefix (or start sequence) of a right sentential form that does not extend past the right end of its rightmost handle. Less formally, a viable prefix is one of the set of prefixes of right sentential forms that may legally be generated on the shift/reduce stack during parsing.

As long as the parse follows such a course, it means that no detectable error has occurred. The program is correct so far, if the input so far can be reduced to a viable prefix. Thus, an automaton which recognises viable prefixes is a successful parser.

It is also important to note that this deterministic automaton is simulating a non-deterministic automaton. Each state in the deterministic automaton represents one or more states in the non- deterministic automaton, all of which would be reached by the same sequence of potentially ambiguous actions.

To build such an automaton we must calculate the states which can be reached and the edges connecting them. As we intend using a deterministic automaton each edge leaving a state must be labelled with a unique symbol, i.e. finding symbol a in state $L$ must always lead to state $M$.

The items derived from the grammar by inserting a dot represent the states of a non-deterministic finite state automaton, i.e. one where several edges from one state may be marked with the same symbol. These items must be grouped so that each group represents one state of the corresponding deterministic automaton. (See Aho, Sethi and Ullman, section 3.6 for more on this). Essentially each grouped state represents the possibility that any of its sub-states may really be correct.

(An alternative to the insertion of a dot in a right hand sequence is to use a tuple, consisting of the number of the production, followed by the number of symbols to the left of the current point.)

Having produced the list of items for each production, we next compute the closure of this set for the whole grammar. The following rules are used.

1. For a set of items $I$, to compute $closure(I)$, first include every item in $I$.
2. If $A\ rightarrow\ a\ \cdot B\ b$ is already in $closure(I)$ and $B\ \rightarrow\ g$, add the item $B\ \rightarrow\ \cdot\ g$ to the set, unless it is already there. Apply this rule until no new items may be added.

Thus, for the item $exp'\ \rightarrow\ \cdot\ exp$, the closure is

| | | | |
|---|---|---|---|
| (1) | $exp'\ \rightarrow\ \cdot\ exp$ | | by rule 1 |
| (2) | $exp\ \rightarrow\ \cdot\ exp\ +\ term$ | } | by rule 2 |
| (3) | $exp\ \rightarrow\ \cdot\ term$ | } | from (1) |
| (4) | $term\ \rightarrow\ \cdot\ term\ *\ factor$ | } | by rule 2 from |
| (5) | $term\ \rightarrow\ \cdot\ factor$ | } | (2) and (3) |
| (6) | $factor\ \rightarrow\ \cdot\ (\ exp\ )$ | } | by rule 2 |
| (7) | $factor\ \rightarrow\ \cdot\ id$ | } | from(4) and (5) |

For a more concise representation, we could merely keep all the non-terminals from the left hand sides added to the initial list and regenerate the corresponding items when needed.

We must also compute the function $GOTO$ for all the items we are interested in. This is defined as follows. If $I$ is a set of items and $X$ is a symbol (non-terminal or terminal), $GOTO(I, X)$ is the closure of the set of items $A \rightarrow a\ X \cdot b$, for which $A \rightarrow a \cdot Xb$ is found in $I$

If we taken the set of items

$$I = \{[exp' \rightarrow exp\ \cdot], [exp \rightarrow exp \cdot + term]\}$$

$GOTO(I, +)$ gives

$$\begin{array}{rcll} exp & \rightarrow & exp + \cdot term & \} \text{ From our rule for } GOTO \\ term & \rightarrow & \cdot term * factor & \\ term & \rightarrow & \cdot factor & \} \text{ closure of above} \\ factor & \rightarrow & \cdot (\ exp\ ) & \\ factor & \rightarrow & \cdot \text{ id} & \end{array}$$

### Intuitive meanings

Closure can be interpreted as saying that if

$$A \rightarrow a \cdot B\ b$$

is in the closure of $I$, at that point in parsing the sequence $B\ b$ may be our input. Further if $B \rightarrow g$ is a production, a prefixing sub-string of $g$ might also be the next to be input.

$GOTO$ says that if $I$ is a set of items valid for one viable prefix form $g$, then $GOTO(I, X)$ is the set of valid successors if $X$ follows $g$ and $g\ X$ is a viable prefix.

### Sets of items

To construct the canonical collection of sets of $LR(0)$ items for an augmented grammar, we use the algorithm which follows.

For augmented grammar $G'$

```
procedure Items(G');
begin
   C:={closure([S' -> S])};
   repeat
      for each set of items I in C and each grammar symbol
      X, where GOTO(I,X) is not empty and not in C do
         add GOTO(I,X) to C
   until no more sets of items can be added to C
end
```

### Canonical collection of sets of $LR(0)$ items

Here is the complete collection of $LR(0)$ items for our expression grammar.

$$
\begin{array}{rrcl}
I_0: & exp' & \to & \cdot\, exp \\
 & exp & \to & \cdot\, exp + term \\
 & exp & \to & \cdot\, term \\
 & term & \to & \cdot\, term * factor \\
 & term & \to & \cdot\, factor \\
 & factor & \to & \cdot\, (\ exp\ ) \\
 & factor & \to & \cdot\, \mathrm{id}
\end{array}
\qquad
\begin{array}{rrcl}
I_1: & exp' & \to & exp\, \cdot \\
 & exp & \to & exp\, \cdot\, + term
\end{array}
$$

$$
\begin{array}{rrcl}
I_2: & exp & \to & term\, \cdot \\
 & term & \to & term\, \cdot\, * factor
\end{array}
\qquad
I_3: \ term \to factor\, \cdot
$$

$$
\begin{array}{rrcl}
I_4: & factor & \to & (\ \cdot\, exp\ ) \\
 & exp & \to & \cdot\, exp + term \\
 & exp & \to & \cdot\, term \\
 & term & \to & \cdot\, term * factor \\
 & term & \to & \cdot\, factor \\
 & factor & \to & \cdot\, (\ exp\ ) \\
 & factor & \to & \cdot\, \mathrm{id}
\end{array}
$$

$$
I_5: \ factor \to \mathrm{id}\, \cdot
$$

$$
\begin{array}{rrcl}
I_6: & exp & \to & exp + \cdot\, term \\
 & term & \to & \cdot\, term * factor \\
 & term & \to & \cdot\, factor \\
 & factor & \to & \cdot\, (\ exp\ ) \\
 & factor & \to & \cdot\, \mathrm{id}
\end{array}
$$

$$
\begin{array}{rrcl}
I_7: & term & \to & term * \cdot\, factor \\
 & factor & \to & \cdot\, (\ exp\ ) \\
 & factor & \to & \cdot\, \mathrm{id}
\end{array}
$$

$$
\begin{array}{rrcl}
I_8: & factor & \to & (\ exp\, \cdot\ ) \\
 & exp & \to & exp\, \cdot\, + term
\end{array}
$$

$$
\begin{array}{rrcl}
I_9: & exp & \to & exp + term\, \cdot \\
 & term & \to & term\, \cdot\, * factor
\end{array}
$$

$$
I_{10}: \ term \to term * factor\, \cdot
$$

$$
I_{11}\ factor \to (\ exp\ )\, \cdot
$$

### Constructing the tables

1. Construct $\{C = I_0, I_1 ... I_n\}$ as above.
2. State $i$ is constructed from $I_i$. The actions are determined by:
   (a) if $[A \to \alpha \cdot a\, \beta]$ is in $I_i$ and $GOTO(I_i, a) = I_j$, set $ACTION[i, a]$ to "`shift j`", where $a$ must be a terminal
   (b) if $[A \to \alpha \cdot]$ is in $I_i$, set $action[i, a]$ to "`reduce` $A \to \alpha$" for all $a$ in $Follow(A)$, where Follow(A) is the set of symbols which may appear immediately to the right of $A$ in any sentential form except $S'$.
   (c) if $[S' \to S \cdot]$ is in $I_i$, set $ACTION[i, \$]$ to "`accept`".
3. The *goto* transitions for state $i$, for all non-terminals $A$, come from:
   `if` $GOTO(I_i, A) = I_j$ `then` $GOTO[i, A] = j$
4. All entries not yet defined constitute errors in the program being parsed.
5. The start state is made up from the set of items containing $[S' \to \cdot\, S]$.