Summary: LR(1) and LALR(1) Parsing

1 LR(1) Parsing

The SLR(1) approach restricts the availability of reduce actions: they are allowed only if the next input symbol is in the Follow set of the LHS of the reducing production. However, in a given parsing context (a given state), the full follow set of a nonterminal is not always available (see lecture 8 slides for an example).

An LR(1) parser keeps track of which terminals are actual permitted followers of a given symbol in each given parsing state. It thus recognizes fewer valid reduce operations, and thus some shift-reduce and reduce-reduce conflicts are avoided.

1.1 Change to the Augmented Rule

In LR(0) and SLR(1), we add a rule, called the augmented rule, for recognition of the Start symbol of the grammar:

\[ S' :\cdot S \quad \$ \]

In LR(1) and LALR(1), the format of this rule does not include the $:

\[ S' :\cdot S \]

1.2 Closures with Lookahead sets

When determining the closures for a LR(0) grammar, some extra information is indicated with each item: the set of possible terminals which could follow the item’s LHS. This set of items is called the lookahead set for the item.

For instance, given the normal closure for \( S_0 \):

\[
\begin{align*}
E' & :\cdot E \\
E & :\cdot T \\
E & :\cdot E + T \\
T & :\cdot \text{id} \\
T & :\cdot ( E )
\end{align*}
\]

The first item, the augmented rule, must be followed by $ (meaning that the recognition of the start symbol should consume all the input, leaving $ as the next symbol in the input). Now, the next 2 items are expansions of the ‘E’ in the first item, and since that E is at the end of the rule, then the lookahead set for the two E items is the same as for E’ i.e. \{ $ \}. I.e.,

\[
\begin{align*}
E' & :\cdot E \quad \{ $ \} \\
E & :\cdot T \quad \{ $ \} \\
E & :\cdot E + T \quad \{ $ \}
\end{align*}
\]

The final two productions are expansions of the T in the second production, and T is the final symbol of the item, and thus the T can only be followed by $. The lookahead set of the final items is thus also \{ $ \}. The LR(1) closure for \( S_0 \) is thus:

\[
\begin{align*}
E' & :\cdot E \quad \{ $ \} \\
E & :\cdot T \quad \{ $ \} \\
E & :\cdot E + T \quad \{ $ \} \\
T & :\cdot \text{id} \quad \{ $ \} \\
T & :\cdot ( E ) \quad \{ $ \}
\end{align*}
\]
Note however that the E on the RHS of the 3rd rule is normally ignored, since we previously dealt with E at a higher level. However, for an LR(1) closure, we need to expand this E as well. The E can be followed by a ‘+’, so we add two items to the closure:

\[
E \cdot T \quad \{ + \} \\
E \cdot E + T \quad \{ + \}
\]

We then need to consider expanding the nonterminals after the dots in these new rules. In the case of the second, the added items would be the same as these two shown here (rules for E with lookahead symbol ‘+’), so recursion can stop. In the case of the first item, we now need to consider expanding the T with a lookahead set of { + }:

\[
T \cdot \text{id} \quad \{ + \} \\
T \cdot (E) \quad \{ + \}
\]

So, our final LR(1) closure for \( S_0 \) is as follows:

\[
E' \cdot . E \quad $ \quad \{ \} \\
E \cdot . T \quad \{$ \} \\
E \cdot . E + T \quad \{$ \} \\
T \cdot . \text{id} \quad \{$ \} \\
T \cdot . (E) \quad \{$ \} \\
E \cdot . T \quad \{ + \} \\
E \cdot . E + T \quad \{ + \} \\
T \cdot . \text{id} \quad \{ + \} \\
T \cdot . (E) \quad \{ + \}
\]

We can simplify by merging rules with the same item but different lookahead list:

\[
E' \cdot . E \quad $ \quad \{ \} \\
E \cdot . T \quad \{$, + \} \\
E \cdot . E + T \quad \{$, + \} \\
T \cdot . \text{id} \quad \{$, + \} \\
T \cdot . (E) \quad \{$, + \}
\]

### 1.3 Constructing the LR(1) DFA

In the SLR(1) case, we could unify two states if they shared the same closure. The same rule applies here, but now two states can only be merged if the items and their lookahead sets are identical. This produces a parse table with far more states.

### 1.4 Constructing the Parse Table

In an SLR parser, when determining which input symbols allow a reduce, we use the FOLLOW set for the rule’s LHS.

In an LR(1) parser, we use instead the lookahead set. This set is a subset of the full FOLLOW set of the item’s LHS. In the case of items with a dot after the final symbol, the LR(1) parser is thus more selective as to which next input symbols can cause a reduce action.

The process for constructing the parse table from the DFA is much the same as for SLR(1), except:

- In deciding which columns to put reduce actions in, while previously the FOLLOW set for a rule was used, here we only place a reduce action under each of the terminals in the lookahead set for the production.
1.5 Grammar Limitations

A LR(1) grammar is one where the construction of an LR(1) parse table does not require two action (shift-reduce or reduce-reduce) in any one cell.

Many conflicts in SLR(1) parse tables are avoided if the LR(1) parse approach is used, because the latter approach is more restrictive on where it allows reduce operations. An SLR(1) parse table may allow reduces where the next input token should not allow such.

2 LALR(1) Parsing

For practical grammars of programming languages, LA(1) parse tables can consist of 1000s of times more states than in SLR(1) and LR(0) parsers.

2.1 Building the DFA

The LALR(1) parser basically allows merging of DFA states where the items of the states are the same, ignoring differences in the lookahead set. The lookahead sets of such items are then appended.

For instance, given the previous grammar, using an LR(1) parser, we might have two states:

\[
\begin{align*}
S_4 & : T : ( \cdot E ) \quad \{ $ \} \\
& : E : \cdot T \quad \{ ), + \} \\
& : E : \cdot E + T \quad \{ ), + \} \\
& : T : \cdot id \quad \{ ), + \} \\
& : T : \cdot ( E ) \quad \{ ), + \}
\end{align*}
\]

\[
\begin{align*}
S_8 & : T : ( \cdot E ) \quad \{ + \} \\
& : E : \cdot T \quad \{ ), + \} \\
& : E : \cdot E + T \quad \{ ), + \} \\
& : T : \cdot id \quad \{ ), + \} \\
& : T : \cdot ( E ) \quad \{ ), + \}
\end{align*}
\]

An LALR(1) parser would allow these two states to merge, giving:

\[
\begin{align*}
S_{4'} & : T : ( \cdot E ) \quad \{ $, + \} \\
& : E : \cdot T \quad \{ ), + \} \\
& : E : \cdot E + T \quad \{ ), + \} \\
& : T : \cdot id \quad \{ ), + \} \\
& : T : \cdot ( E ) \quad \{ ), + \}
\end{align*}
\]

2.2 Constructing the Parse Table

Exactly the same as for an LR(1) parser.

Allowing more state mergers reduces the number of states back to that of an SLR(1) parser. However, the number of reduce operations in the parse table is still substantially lessened (due to tracking the lookahead symbols of each item), and thus many shift-reduce and reduce-reduce conflicts are avoided.
2.3 Using the Parse Table

If correct input is given, the LALR(1) parse table will follow exactly the same sequence of actions, although of course state numbers may differ between the two approaches.

For erroneous input, there may be cases where the next input token allows a reduce to take place where it may not in fact lead to a successful parse (if two states were merged, and from one of the states the reduce was allowed, but not from the other). However, it can be demonstrated that in such cases the parser will detect the error at a later point, and before another shift operation takes place (see Aho, Sethi, Ullman p 237).

2.4 Grammar Limitations

An LALR(1) grammar is one which allows a LALR(1) parse table to be constructed with never more than one action per cell.

LALR(1) grammars allow a wider set of structures than SLR(1) grammars, including some forms common to programming languages (e.g., the if … else structure). While LALR(1) grammars do not handle some cases allowed in LR(1) grammars, in practice these forms are not needed to describe most programming languages.

3 LR(1) Parsing with Lambda productions

Only difference from SLR(1) is the calculation of the lookahead set. TO EXPAND.