# LALR Parsing

An LALR parser is a special kind of DPDA, with output. Its input is a string generated by a context-free grammar $G$, and its output is a rightmost derivation of that string.

We will start with $G$ being the following context free grammar.
1. $E \to E + E$
2. $E \to E - E$
3. $E \to x$
4. $E \to y$

Since this is an algebraic language, I am using $E$ for the start symbol: $E$ stands for "expression." The string $w = x - y + x$ is generated by $G$. Since $G$ is ambiguous, there is more than one parse tree for $w$, but, by the rule that addition and subtraction are left-associative, there is only one parse tree which agrees with the standard meaning of the string. The output is the reverse rightmost derivation of $w$. Since the grammar is ambiguous, there is more than one rightmost derivation, but only one of those corresponds to the "correct" parse tree. The output of the LALR parser for $x - y + z$ is the string 34231. Do you see why?

Here is the "correct" rightmost derivation of $w$.

$E \overset{1}{\Rightarrow} E + E \overset{3}{\Rightarrow} E + x \overset{2}{\Rightarrow} E - E + x \overset{4}{\Rightarrow} E - y + x \overset{3}{\Rightarrow} x - y + x$

## Parts of the LALR Parser

- The parser consists of an input file, a stack, and an output file. The parser is guided by an ACTION table and a GOTO table.
- The input file is initially the input string $w$ followed by an end-of-file symbol: \$. As each symbol is read, it is deleted from the input file.
- The LALR parser can "peek" at the first unread input symbol without reading it, and can "peek" at the top stack symbol without popping it.
- The items on the stack are either grammar symbols or numbers, which are called *stack states*. Grammar symbols and stack states alternate.
  - The bottom of stack symbol is the stack state 0.
  - The top symbol in the stack is the "current" stack state.
  - Above every grammar symbol in the stack is a stack state.
  - Above every stack state in the stack, except for the top symbol, is a grammar symbol.
- The parser is permitted to pop several symbols at one step.

## Computation of the LALR Parser

To guide our construction of the ACTION and GOTO tables, we rewrite the grammar, placing a subscript, which is a stack state, under some of the grammar symbols on the right hand side.

1. $E \to E +_2 E_3$
2. $E \to E -_4 E_5$
3. $E \to x_6$
4. $E \to y_7$

| | ACTION | | | | | GOTO |
|---|---|---|---|---|---|---|
| | x | y | + | - | $ | E |
| 0 | s6 | s7 | | | | 1 |
| 1 | | | s2 | s4 | HALT | |
| 2 | s6 | s7 | | | | 3 |
| 3 | | | r1 | r1 | r1 | |
| 4 | s6 | s7 | | | | 5 |
| 5 | | | r2 | r2 | r2 | |
| 6 | | | r3 | r3 | r3 | |
| 7 | | | r4 | r4 | r4 | |

| stack | | input | action | output |
|---|---|---|---|---|
| ${}_0$ | : | $x - y + x\$$ | | |
| ${}_0x_6$ | : | $-y + x\$$ | shift 6 | |
| ${}_0E_1$ | : | $-y + x\$$ | reduce 3 | 3 |
| ${}_0E_1{-_4}$ | : | $y + x\$$ | shift 4 | 3 |
| ${}_0E_1 -_4 y_7$ | : | $+x\$$ | shift 7 | 3 |
| ${}_0E_1 -_4 E_5$ | : | $+x\$$ | reduce 4 | 34 |
| ${}_0E_1$ | : | $+x\$$ | reduce 2 | 342 |
| ${}_0E_1{+_2}$ | : | $x\$$ | shift 2 | 342 |
| ${}_0E_1 +_2 x_6$ | : | $ | shift 6 | 342 |
| ${}_0E_1 +_2 E_3$ | : | $ | reduce 3 | 3423 |
| ${}_0E_1$ | : | $ | reduce 1 | 34231 |
| ${}_0E_1$ | : | $ | HALT | 34231 |

### Actions

The LALR parser has three kinds of action: shift, reduce, and halt.

1. Choose an action from the ACTION table, using the current stack state and the next input symbol.

2. If the current action is "shift n" then read the next input symbol and push that symbol onto the stack. Then push the stack state n onto the stack.

3. If the current action is "reduce m" then pop the entire right side of production m off the stack, as well as the stack states that follow those symbols. In our example, if $m$ is 3 or 4, we pop one terminal symbol and its stack state. If $m$ is 1 or 2, we need to pop six symbols: the right side of production m as well as all the stack states above those grammar symbols.

   Push the symbol on the left side of the production onto the stack. In our example, this symbol is always the start symbol $E$. We then push a stack state on top of that. The GOTO table tells us which stack state to push. In our example, when we push E onto stack state 0, we push stack state 1; when we push E onto stack state 2, we push stack state 3; and when we push E onto statck state 4, we push stack state 5. Finally, we write $m$ to the output file

4. If the current action is "halt," the parser halts, and the current output file is the reverse rightmost derivation of the input string.