LR(k) Parsing

We learned that the typical strategy of LL(k) parsing is that if a nonterminal
symbol A appears at the stack top, the parser chooses a production rule for A,
say A→α, depending on the k look-ahead contents, if needed, and pops the
nonterminal A at the stack top by α.

In LR(k) parsing, the parser does the opposite, i.e., when the machine sees
(indeed reads from inside out) α at the stack top portion, chooses a production
rule that generates α, say A→α, depending on the k look-ahead contents, if
needed, and replaces α with A (see the figure on the following slide). Notice that
|α| ≥ 0 in general. Hence, LR(k) parser have additional capability of looking
down the stack of some constant depth. The depth is no greater than the
maximum length of the right side of production rules.

If such α does not appear at the stack, the parser shifts in (i.e., reads and pushes)
the next input symbol and again examine if stack top portion can be reduced by
applying a production rule. The parser iteratively applies this strategy until the
whole input string is processed and the stack contains only the start symbol S,
which is the final accepting configuration. If an input string x has been parsed
successfully, the sequence of production rules applied to reduce the stack turns
out to be exactly the reverse order of the rules applied for the rightmost
derivation for x.
A → α | β ...
For the basic strategy of LR(k) parsing consider the following simple grammar.

\[
S \rightarrow A | B \quad A \rightarrow abc \quad B \rightarrow abd
\]

We can easily construct an LL(3) parser for this grammar. With LR(k) parsing strategy, we can build LR(0) parser for the grammar, i.e., the parser does not need to look ahead. For input string abc, the parser parses the input as follows. The parser shifts in the input onto the stack until it sees abc (read the stack inside out) at the stack top portion. This stack top portion is replaced (popped out and then pushed) with A. String abd can also be parsed similarly.

\[
(q_0, \text{abc}, Z_0) \Rightarrow \ldots \Rightarrow (q_1, \varepsilon, \text{cba}Z_0) \Rightarrow (q_1, \varepsilon, AZ_0) \Rightarrow (q_1, \varepsilon, SZ_0)
\]

As in LL(k) parsing, the parser may encounter an uncertainty in reducing the stack top portion. If the grammar is LR(k), the parser can resolve the ambiguity by looking the input string ahead no more than k cells. We shall see more such examples.
Example 1. For the following CFG we will construct an LR(k) parser with minimum k.

\[
\begin{array}{c|c}
(1) & S \rightarrow ADC | \text{aaadd} \\
(2) & \text{A} \rightarrow \text{aaa} \\
(3) & \text{D} \rightarrow \text{ddd} \\
(4) & \text{C} \rightarrow \text{Cc} | \text{c} \\
(5) & \text{ADCcc} \\
(6) & \text{ADccc} \\
\end{array}
\]

Notice that the language of this grammar is \(\{\text{aaadd}\} \cup \{\text{aaaddc}^i | i > 0\}\). We first examine how string \text{aaabbbccc} can be parsed according to LR(k) strategy. The rightmost derivation for this string is shown below. We will show how our parser parses this string by applying the production rules in reverse order of the rules applied for the rightmost derivation, i.e., (3)(4)(6)(5)(5)(1).

\[
\begin{align*}
S \Rightarrow & \ ADC \Rightarrow \ ADCc \Rightarrow \ ADCcc \Rightarrow \ ADccc \Rightarrow \ Adddccc \Rightarrow \ aaadddcc
\end{align*}
\]
Initially there is no string in the stack that can be reduced by applying a production rule. The parser, reading the input symbols, shifts them into the stack until it sees a stack top portion $aaa$ that can possibly be produced by the last step of the rightmost derivation, which is rule (3). So the parser does the following.

$$
(q_0, \text{aaaddcc, } Z_0) \Rightarrow (q_1, \text{aaddccc, } aZ_0) \Rightarrow (q_1, \text{addccc, } aaZ_0) \Rightarrow (q_1, \text{dddcc, } aaaZ_0) \Rightarrow ?
$$

Can we let the parser simply replace the stack top portion $aaa$ by $A$? No, because if the input were $aaaddd$, the parser will fail to parse the string, since $A$ is not used to derive it. For this input it is too early to apply a rule. So the parser needs some information from the input to resolve this problem.
Notice that if the input were aaabbb, the parser should have shifted the whole input string into the stack and reduce it by S, which results in a successful parsing as follows. (Recall that the parser reads the stack top portion inside out.)

\[(q_0, \text{aaabbb}, Z_0) \Rightarrow (q_1, \text{aaabbb}, aZ_0) \Rightarrow \ldots \Rightarrow (q_1, \varepsilon, \text{dddaaaZ}_0) \Rightarrow (q_1, \varepsilon, SZ_0)\]

However, for input string aaaddccc, the parser should not shift in the d’s, because it will lose the chance for applying rule (3) to reduce aaa by A at the stack top. To choose the right step, the parser needs to look ahead at least 4 cells to see if there appears c after 3 d’s. For the given input the parser looks ahead and sees dddc, and decides that it is right time to reduce (i.e., replace) the stack top portion aaa with A applying rule (3) as follows.

\[(q_0, \text{aaaddccc}, Z_0) \Rightarrow \ldots \Rightarrow (q_1, \text{aaaddccc}, aZ_0) \Rightarrow (q_1, \text{dddc}, \text{aaaZ}_0) \Rightarrow (q_1, \text{dddcc}, AZ_0) \Rightarrow ?\]
Now the parser sees no stack top portion that is reducible until three d’s are shifted in. This string ddd is reduced by applying rule (4) without looking ahead, because this is the only case. When the next c is shifted in onto the stack, it is easily identified that rule (6) must be applied, because the leftmost c is produced by this rule. Thus, the c at the stack top is reduced by C. Notice that the parser does not need to looking ahead for this reduction.

Now, there is ADC in the stack top portion which could be reducible by S. However, since there are more c’s in the input, it is too early to apply this rule. To resolve this problem the parser needs 1 look-ahead, and shift in the remaining c’s one by one and reduce the stack top portion Cc with C applying rule (5) until all c’s in the input tape are shifted in and reduced.
Only when the last c has been shifted in, and the Cc at the stack top reduced by C, the parser should reduce the ADC by S, and can enter in the successful final configuration \((q_1, \varepsilon, SZ_0)\). Clearly, the sequence of production rules applied by this LR(4) parser \((3)(4)(6)(5)(5)(1)\) is exactly the reverse sequence of the rules applied for the rightmost derivation of the input string. This parser is formally defined with, so called, the reduction table as shown on the next slide.
Reduction Table

<table>
<thead>
<tr>
<th>Stack top portion</th>
<th>dddc</th>
<th>dddB</th>
<th>BBBB</th>
<th>cxxx</th>
<th>xxxx</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA</td>
<td>A</td>
<td>Shift-in</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DDD</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AAAADD</td>
<td></td>
<td></td>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>Cc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>ADC</td>
<td></td>
<td></td>
<td>S</td>
<td>Shift-in</td>
<td></td>
</tr>
</tbody>
</table>

4 look-ahead

x: don’t care
B : blanks
Example 2. Construct an LR(k) parser for the following grammar with minimum k.

\[
\begin{align*}
S & \rightarrow EAF \mid EBF \\
E & \rightarrow aE \mid a \\
A & \rightarrow aaab \\
B & \rightarrow aaac \\
F & \rightarrow aaad
\end{align*}
\]

(1) (2) (3) (4) (5) (6) (7)

It is easy to see that this grammar generates the following language.

\[
\{a^ixaaad \mid i \geq 1, x \in \{ aaab, aaac \} \}
\]

Notice that above grammar is not LL(k) grammar for any constant k, because in the language specification, string x which provides the information for choosing one of rules (1) and (2) can be at arbitrarily far to the right from the first input symbol.

We can construct an LR(4) parser for this grammar. As we did for the previous examples, we will first analyze how we can construct such parser with a typical string of the grammar. Consider string aaaaaabaaad which is generated by the following rightmost derivation.

\[
\begin{align*}
S & \Rightarrow EAF \\
EAF & \Rightarrow EAaaad \\
EAaaad & \Rightarrow Eaaabaaad \\
aEaaabaaad & \Rightarrow aaEaaabaad \\
aaaEaaabaad & \Rightarrow aaaaaabaad
\end{align*}
\]
Since every rightmost derivation of the grammar ends by applying rule (4) \( E \rightarrow a \), the initial objective of the parser to bring in the \( a \) generated by rule (4) from the input tape to the stack top. Then the parser can simply reduces it by \( E \) applying rule (4). Notice that for every string in the language \( \{ a^i x a a d \mid i \geq 1, x \in \{ aaab, aaac \} \} \), the part \( a^i \) is produced by \( E \rightarrow a E \mid a \), with the last \( a \) generated by rule \( E \rightarrow a \) that is followed by either \( aaab \) or \( aaac \). Hence, if the parser shifts in \( a \)'s from the input until it sees ahead either \( aaab \) or \( aaac \), then the \( a \) on top of the stack is the one produced by \( E \rightarrow a \). Thus, the parser does the following based on this observation.

\[
(q_0, \text{aaaaaabaaad}, Z_0) \Rightarrow (q_1, \text{aaaaabaaad}, aZ_0) \Rightarrow \ldots \Rightarrow
\]

\[
(q_1, \text{aabaaad}, \text{aaaZ}_0) \overset{(4)}{\Rightarrow} (q_1, \text{aabaad}, EaaZ_0) \Rightarrow ?
\]
The stack top can be reduced further by applying $E \rightarrow aE$ twice. Notice that the a’s in the stack are generated by this rule.

$$(q_1, \text{aaabaaad}, \text{EaaZ}_0) \Rightarrow (q_1, \text{aaabaaad}, \text{EaZ}_0) \Rightarrow (q_1, \text{aaabaaad}, \text{EZ}_0) \Rightarrow ?$$

Since the stack top cannot be reduced further, input symbols are shifted in until the next reduction is possible. Notice that throughout these last moves, the parser needs no look ahead.

$$(q_1, \text{aabaad}, \text{aEZ}_0) \Rightarrow \ldots \Rightarrow (q_1, \text{aad}, \text{baaaEZ}_0) \Rightarrow (q_1, \text{aad}, \text{AEZ}_0) \Rightarrow \ldots$$

$$\Rightarrow (q_1, \varepsilon, \text{daaaAEZ}_0) \Rightarrow (q_1, \varepsilon, \text{FAEZ}_0)$$

$$(q_1, \varepsilon, \text{SZ}_0)$$

<table>
<thead>
<tr>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
<th>(7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow \text{EAF</td>
<td>EBF}$</td>
<td>$E \rightarrow \text{aE</td>
<td>a}$</td>
<td>$A \rightarrow \text{aab}$</td>
<td>$B \rightarrow \text{aaac}$</td>
<td>$F \rightarrow \text{aad}$</td>
</tr>
</tbody>
</table>
The sequence of rules applied by the LR(4) is (7)(6)(5)(4)(3)(2)(1), which is the sequence applied for the rightmost derivation of the input string. Based on this analysis we can construct the following reduction table of the parser.

<table>
<thead>
<tr>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
<th>(7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S→EAF</td>
<td>EBF</td>
<td>E→aE</td>
<td>a</td>
<td>A→ aaab</td>
<td>B→ aaac</td>
<td>F→ aaad</td>
</tr>
</tbody>
</table>

The sequence of rules applied by the LR(4) is (7)(6)(5)(4)(3)(2)(1), which is the sequence applied for the rightmost derivation of the input string. Based on this analysis we can construct the following reduction table of the parser.

**Reduction Table**

<table>
<thead>
<tr>
<th>Stack top</th>
<th>E</th>
<th>E</th>
<th>Shift-in</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aaab</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aaac</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aaad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EBF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4 look-ahead:

<table>
<thead>
<tr>
<th>aab</th>
<th>aac</th>
<th>aaaa</th>
<th>xxxx</th>
</tr>
</thead>
</table>

168
Example 3. Construct LR(k) parser with minimum k for the following grammar.

\[
\begin{array}{cccc}
(1) & (2) & (3) & (4) \\
S \rightarrow & bS & | & Accc \\
A \rightarrow & bAc & | & bc \\
\end{array}
\]

We construct an LR(3) parser for this grammar. The language of this grammar is

\[
\{b^i b^n c^n ccc \mid i \geq 0, n \geq 1 \}.
\]

Clearly, every derivation of this grammar terminates by rule \(A \rightarrow bc\) which generates \(bc\) at the center of \(b^n c^n\). This \(bc\) can be shifted in from the tape by simply reading the input and pushing it on the stack until \(bc\) appears at the stack top. This is the target of the first reduction. Lets study how an LR(3) parser can parse string \(bbbbbbcccccc\) of the language. The rightmost derivation of this string is

\[
(1) (1) (2) (3) (3) (4) \\
S \Rightarrow & bS \Rightarrow & bbS \Rightarrow & bbAacc \Rightarrow & bbbbAcccc \Rightarrow & bbbbbbccccccc \\
\]

Our LR(3) parser will parse this string applying the sequence of production rules in the reverse order of the rightmost derivation, i.e., \((4)(3)(3)(2)(1)(1)\).
<table>
<thead>
<tr>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S→ bS</td>
<td>Accc</td>
<td>A → bAc</td>
<td>bc</td>
</tr>
</tbody>
</table>

The parser shifts the input in until it sees bc (again read inside out) in the stack as follows.

\[(q_0, bbbbbccccc, Z_0) \Rightarrow \ldots \Rightarrow (q_1, cccccc, cbbbbZ_0) \Rightarrow\]

String bc is reduced by A applying production \(A \rightarrow bc\), and then the next input symbol is shifted in because no further reduction is possible.

\[(q_1, cccccc, cbbbbZ_0) \Rightarrow (q_1, cccccc, AbbbbZ_0) \Rightarrow (q_1, ccc, cAbbbbZ_0) \Rightarrow\]

String bAc is reduced by A, the next c is shifted in, bAc is reduced by A again, and so on, until there remains last three c’s.

\[(q_1, ccc, AbbbZ_0) \Rightarrow (q_1, ccc, cAbbbZ_0) \Rightarrow (q_1, ccc, AbbZ_0) \Rightarrow\]
Now, bS on the top of the stack can be reduced by S for the production $S \rightarrow bS$ until only S remains in the stack, which is the accepting configuration. The reduction table of this LR(3) parser is shown on the next slide.

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
S \rightarrow bS | Accc & A \rightarrow bAc | bc & & \\
\end{array}
\]

Since the last three c’s are generated by $S \rightarrow Accc$, after shifting in the next c into the stack, if there remains only two c’s in the input tape, the parser should stop reducing bAc by A. (For this, it needs 3-look-ahead.) The remaining two c’s in the tape should be shifted in onto the stack, and then Accc is reduced by S applying the production (2) $S \rightarrow Accc$ as follows.

\[
(q_1, cc, cAbbZ_0) \Rightarrow \ldots \Rightarrow (q_1, \varepsilon, cccAbbZ_0) \Rightarrow (q_1, \varepsilon, SbbZ_0) \Rightarrow
\]

Now, bS on the top of the stack can be reduced by S for the production $S \rightarrow bS$ until only S remains in the stack, which is the accepting configuration. The reduction table of this LR(3) parser is shown on the next slide.

\[
(q_1, \varepsilon, SbbZ_0) \Rightarrow (q_1, \varepsilon, SbZ_0) \Rightarrow (q_1, \varepsilon, SZ_0)
\]
Example 4. Consider the following two CFG’s which generate the same language \( \{a^i x \mid i \geq 1, x \in \{b,c\}\} \).

\[
G_1 : S \rightarrow Ab|Bc \quad A \rightarrow Aa|a \quad B \rightarrow Ba|a
\]

\[
G_2 : S \rightarrow Ab|Bc \quad A \rightarrow aA|a \quad B \rightarrow aB|a
\]

It is easy to see that there is no LR(k) parser for \( G_1 \) for any constant \( k \), while \( G_2 \) is an LR(1) grammar.
LR(k) Grammars and Deterministic Bottom-up Left-to-right Parsing
(Formal Definition)

Definition (LR(k) grammar). For CFG $G = (V_T, V_N, P, S)$, let $\alpha\beta x$ and $\alpha\beta y$ be two sentential forms of $G$, for some $\alpha, \beta \in V^*$ and $x, y \in V_T^*$, which are derivable by some rightmost derivations. Grammar $G$ is an LR(k) grammar, if for a constant $k$, it satisfies the following conditions.

(i) $(k)x = (k)y$ and

(ii) if $A \rightarrow \beta$ is the last production used to derive $\alpha\beta x$ (i.e., $\alpha A x \Rightarrow \alpha\beta x$), then $A \rightarrow \beta$ must also be used to derive $\alpha\beta y$ from $\alpha Ay$ for the rightmost derivation.

Definition. An extended PDA is a PDA which can read some constant number of symbols from the top of the stack and rewrite them by other symbol.

An LR(k) parser is an extended DPDA which can look the input ahead $k$ symbols, for some constant $k$. 
Ruminating on LL(k) and LR(k) Parsing

• Since parsers are constructed based on grammars, it does not make sense to say LL(k) languages or LR(k) languages.

• Recall that a CGF G is ambiguous if there is a string $x \in L(G)$ for which two parse trees exist. Obviously, given such $x$ it it impossible for a parser to know which derivation is used to generate $x$. However, this fact does not implies that we cannot build a parser which generates a sequence of production rules that produces $x$. Consider the following simple ambiguous CFG whose language is $\{a\}$.

$$S \rightarrow A | B \quad A \rightarrow a \quad B \rightarrow a$$

It is trivial to construct an LL(0) parser (or an LR(0) parser) which simply choose either $S \rightarrow A$ or $S \rightarrow B$. However, it is impossible for the parser to decide which production rule should be used to derive $a$. (Pascal’s if-statement is such case.)
Ruminating on LL(k) and LR(k) Parsing (cont’ ed)

• We can easily prove that LL(k) parsers need no more than two states. There are LR(k) grammars for which more than two states are needed. In general, LR(k) parsers need some finite number of states depending on the grammar. The following example shows why.

\[
S \rightarrow aaAd \mid baBd \mid caCd \\
A \rightarrow aAd \mid a \quad B \rightarrow aBd \mid a \quad C \rightarrow aCd \mid a
\]

This grammar generates the language \( \{ xa^i ad^i | x \in \{a,b,c\}, i \geq 1 \} \). Notice that an LR(k) parser for this grammar should push \( xa^i a \) onto the stack before it begins to reduce the stack by applying one of the productions \( A \rightarrow a \), \( B \rightarrow a \) and \( C \rightarrow a \) depending on whether \( x \) is \( a \), \( b \), or \( c \), which is at the bottom of the stack. It can be located at arbitrarily far down at the bottom of the stack. It is too deep to look down. Reading \( x \), it must be stored in the memory. This implies that the parser needs two more states for this grammar (for example \( q_1 \), \( q_2 \) and \( q_3 \), respectively, for the cases of \( x = a \), \( x = b \), and \( x = c \)).