Lexical and Syntax Analysis

Part II
The LL Grammar Class

• Recursive-descent and other LL parsers can be used only with grammars that meet certain restrictions.

• Left recursion causes a catastrophic problem for LL parsers. Calling the recursive-descent parsing subprogram for the following rule would cause infinite recursion:

   \[ A : A + B \]

   The left recursion in the rule \( A : A + B \) is called direct left recursion, because it occurs in one rule.

• Indirect left recursion poses the same problem as direct left recursion:

   \[ A : B a A \]
   \[ B : A b \]
Functions for Left Recursive Rules (infinite recursion!)

# A : A + B
def A():
    global next_token
    global l
    A()
    if next_token.get_token().value == TokenTypes.PLUS.value:
        next_token = l.lex()
    B()

# A : B a A
def A():
    global next_token
    global l
    B()
    if next_token.get_token().value == TokenTypes.ATOKEN.value:
        next_token = l.lex()
    A()
    else:
        error("Expecting ATOKEN")

# B : A b
def B():
    global next_token
    global l
    A()
    if next_token.get_token().value == TokenTypes.BTOKEN.value:
        next_token = l.lex()
    B()
    else:
        error("Expecting BTOKEN")

Direct Left Recursion

Indirect Left Recursion
Algorithm to Eliminate Direct Left Recursion

For each non-terminal A,

(1) Group the A-rules as

\[ A : A\alpha_1 | \ldots | A\alpha_m | \beta_1 | \ldots | \beta_n \]

where none of the \( \beta \)'s begins with A.

(2) Replace the original A-rules with

\[ A : \beta_1 A' | \ldots | \beta_n A' \]

\[ A' : \alpha_1 A' | \ldots | \alpha_m A' | \varepsilon \]

The symbol \( \varepsilon \) represents the empty string. A rule that has \( \varepsilon \) as its RHS is called an erasure rule, because using it in a derivation effectively erases its LHS from the sentential form.
Eliminate Direct Left Recursion: Example

Consider the following left-recursive grammar:

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

For the E-rules, \( \alpha_1 = + T \) and \( \beta_1 = T \)

So, the new E-rules will be:

\[
\begin{align*}
E & : T E' \\
E' & : + T E' | \varepsilon
\end{align*}
\]

For the T-rules, \( \alpha_1 = * F \) and \( \beta_1 = F \)

So, the new T-rules will be:

\[
\begin{align*}
T & : F T' \\
T' & : * F T' | \varepsilon
\end{align*}
\]

The F-rules do not change since there is not left recursion.

The final transformed grammar with no left-recursion:

\[
\begin{align*}
E & : T E' \\
E' & : + T E' | \varepsilon \\
T & : F T' \\
T' & : * F T' | \varepsilon \\
F & : (E) | id
\end{align*}
\]

There exists algorithms to eliminate indirect left-recursion, but we don’t cover it in this class.
Pairwise Disjointness Test

- Left-recursion is not the only trait that makes top-down parsing problematic.
- A top-down parser also needs to choose the correct RHS based on the next token, especially when the non-terminal being expanded has multiple rules.
- The pairwise disjointness test is used to test a non-left-recursive grammar to determine whether it can be parsed in a top-down fashion. This test requires computing FIRST sets, where

  \[
  \text{FIRST}(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta\}
  \]

  The symbol \( \Rightarrow^* \) indicates a derivation of zero or more steps. If \( \alpha \Rightarrow^* \varepsilon \), then \( \varepsilon \) is also in FIRST(\( \alpha \)), where \( \varepsilon \) is the empty string.

- There are algorithms to compute the FIRST sets for any mixed string. In simple cases, FIRST can be computed by simply examining the grammar.
The pairwise disjoint test: For each nonterminal $A$ that has more than one RHS, and for each pair of rules, $A : \alpha_i$ and $A : \alpha_j$, it must be true that $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$.

Example 1: Consider the following grammar

$$
\begin{align*}
A & : a \ B \\
A & : b \ A \ b \\
A & : B \ b \\
B & : c \ B \\
B & : d
\end{align*}
$$

The FIRST sets for the RHS of $A$-rules are: $\text{FIRST}(aB) = \{ a \}$, $\text{FIRST}(bAb) = \{ b \}$, and $\text{FIRST}(Bb) = \{ c, d \}$. These are disjoint and hence PASS the pairwise disjoint test.

The FIRST sets for the RHS of $B$-rules are: $\text{FIRST}(cB) = \{ c \}$ and $\text{FIRST}(d) = \{ d \}$. These are disjoint and hence PASS the pairwise disjoint test.

So, the grammar as a whole passes the pairwise disjoint test and hence can be parsed using top-down parsers!
Example 2: Consider the following grammar

\[
\begin{align*}
A & : a \ B \\
A & : B \ A \ b \\
B & : a \ B \\
B & : b
\end{align*}
\]

The FIRST sets for the RHS of A-rules are: FIRST(aB) = \{ a \} and FIRST(BAb) = \{ a, b \}. These are not disjoint and hence FAIL the pairwise disjoint test.

The FIRST sets for the RHS of B-rules are: FIRST(aB) = \{ a \} and FIRST(b) = \{ b \}. These are disjoint and hence PASS the pairwise disjoint test.

So, the grammar as a whole fails the pairwise disjoint test and hence cannot be parsed using top-down parsers!
Left Factoring

• In many cases, a grammar that fails the pairwise disjointness test can be modified so that it will pass the test.

• The following rules do not pass the pairwise disjointness test:

\[ <\text{var}> : \text{ID} \mid \text{ID} \ [ <\text{expr}> ] \]

Using the technique of "left-factoring", we can rewrite the rules as follows:

\[ <\text{var}> : \text{ID} <\text{new}> \]
\[ <\text{new}> : \epsilon \]
\[ <\text{new}> : [ <\text{expr}> ] \]

The modified grammar passes the pairwise disjointness test!

• Algorithms do exists for left-factoring, but we do not cover them in this class.

• Left-factoring cannot solve all pairwise-disjointness problems.
Bottom-Up Parsers

• The following grammar for arithmetic expressions will be used to illustrate bottom-up parsing:

\[
E : E + T \mid T \\
T : T * F \mid F \\
F : ( E ) \mid id
\]

This grammar is left-recursive, which is acceptable to bottom-up parsers.

• A rightmost derivation for \(id + id \cdot id\) is shown below (red part highlights the RHS of rules applied at each step):

\[
E \Rightarrow E + T \\
\Rightarrow E + T \cdot F \\
\Rightarrow E + T \cdot id \\
\Rightarrow E + F \cdot id \\
\Rightarrow E + id \cdot id \\
\Rightarrow T + id \cdot id \\
\Rightarrow F + id \cdot id \\
\Rightarrow id + id \cdot id
\]

A bottom-up parser produces the reverse of a rightmost derivation by starting with the last sentential form (the input sentence) and working back to the start symbol.

At each step, the parser’s task is to find the RHS in the current sentential form that must be rewritten to get the previous sentential form.
Bottom-Up Parsers (continued)

- A right sentential form may include more than one RHS. For example, $E + T * id$ contains three RHSs: $E + T$, $T$, and $id$.
- The task of the bottom-up parser is to find the **unique handle** of a given right sentential form.
- **Definition:** $\beta$ is the **handle** of the right sentential form $\gamma = \alpha\beta w$ if and only if
  \[
  S \Rightarrow^{*_{rm}} \alpha\alpha w \Rightarrow_{rm} \alpha\beta w.
  \]
  where $\Rightarrow_{rm}$ specifies a rightmost derivation step, and $\Rightarrow^{*_{rm}}$ specifies zero or more rightmost derivation steps.

\[
\begin{align*}
E : & \quad E + T \mid T \\
T : & \quad T * F \mid F \\
F : & \quad (E) \mid id
\end{align*}
\]
Two other concepts are related to the idea of a handle.

- **Definition:** $\beta$ is a **phrase** of the right sentential form $\gamma = \alpha_1\beta\alpha_2$ if and only if
  
  \[ S \Rightarrow^* \alpha_1\Lambda\alpha_2 \Rightarrow^+ \alpha_1\beta\alpha_2 \]

  where $\Rightarrow^+$ means one or more derivation steps.

- **Definition:** $\beta$ is a **simple phrase** of the right sentential form $\gamma = \alpha_1\beta\alpha_2$ if and only if
  
  \[ S \Rightarrow^* \alpha_1\Lambda\alpha_2 \Rightarrow \alpha_1\beta\alpha_2 \]
Phrases (continued)

A phrase is a string consisting of all of the leaves of the partial parse tree that is rooted at one particular internal node of the whole parse tree.

A simple phrase is a phrase that is derived from a nonterminal in a single step.

Consider the partial parse tree shown to the right. There are 3 internal nodes: E, T, and F.
The three phrases corresponding these 3 internal nodes are: E + T * id, T * id, and id.
The only simple phrase is id.

The handle of a right sentential form is the leftmost simple phrase.
Phrases (another example)

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Partial rightmost derivation for AaaBab</th>
<th>Partial Parse Tree for AaaBab</th>
</tr>
</thead>
<tbody>
<tr>
<td>S : aAb</td>
<td>S =&gt; ABA</td>
<td>S</td>
</tr>
<tr>
<td>S : ABA</td>
<td>=&gt; ABab</td>
<td>A</td>
</tr>
<tr>
<td>A : ab</td>
<td>=&gt; AabBab</td>
<td>A B</td>
</tr>
<tr>
<td>A : BAB</td>
<td>=&gt; AaaBab</td>
<td>A B a</td>
</tr>
<tr>
<td>B : aB</td>
<td></td>
<td>B a</td>
</tr>
<tr>
<td>B : b</td>
<td></td>
<td>B b</td>
</tr>
</tbody>
</table>

The internal nodes of the partial parse tree give the following phrases:

AaaBab
aaB
aB : simple phrase; handle!
ab : simple phrase
Shift-Reduce Algorithms

- Bottom-up parsers are often called **shift-reduce** algorithms, because **shift** and **reduce** are their two fundamental actions.

- The **shift** action moves the next input token onto the parser’s stack. A **reduce** action replaces a RHS (the handle) on top of the parser’s stack by its corresponding LHS.

- Most bottom-up parsing algorithms belong to the **LR family**. LR parsers use a relatively small amount of code and a parsing table.

- The original LR algorithm was designed by **Donald Knuth**, who published it in 1965.

- **Advantages of LR parsers**: (1) They can be built for all programming languages. (2) They can detect syntax errors as soon as possible in a left-to-right scan. (2) The LR class of grammars is a proper **superset** of the class parsable by LL parsers.

- It is difficult to produce an LR parsing table **by hand**. However, there are many programs (e.g. **PLY**) available that take a grammar as input and produce the parsing table.
LR Parsers (Structure)

LR-parsers employ a **stack** to store parsing status at any point in time and use a **parsing table** that informs the next action (shift or reduce).

In general, **each grammar symbol** on the stack will be **followed** by a **state symbol** (often written as a subscripted uppercase S).

![Diagram of LR parser structure](image)
LR Parsers (Continued)

• The contents of the parse stack for an LR parser has the following form, where the Ss are state symbols and the Xs are grammar symbols:

\[ S_0 X_i S_1 X_2 S_1 S_2 \ldots X_m S_m \ (\text{top}) \]

• An LR parser configuration is a pair of strings representing the stack and the remaining input:

\[ (S_0 X_i S_1 X_2 S_1 S_2 \ldots X_m S_m, \ a_i a_{i+1} \ldots a_n \$) \]

The dollar sign is an end-of-input marker.

• The LR parsing process is based on the parsing table, which has two parts, ACTION and GOTO.

• The ACTION part has state symbols as its row labels and terminal symbols as its column labels.

• The parse table specifies what the parser should do, based on the state symbol on top of the parse stack and the next input symbol.
The two primary actions are **shift** (shift the next input symbol onto the stack) and **reduce** (replace the handle on top of the stack by the LHS of the matching rule).

Two other actions are possible: **accept** (parsing is complete) and **error** (a syntax error has been detected).

The values in the **GOTO part of the table** indicate which state symbol should be pushed onto the parse stack after a **reduction** has been completed. The row is determined by the state symbol on top of the parse stack **after** the handle and its associated state symbols have been removed. The column is determined by the **LHS of the rule** used in the reduction.
LR Parser: Parsing Algorithm

Initial state of the LR Parser:

\[(S0, a_1…a_n$)\]

Informal definitions of parser actions:

**Shift**: The next input symbol is pushed onto the stack, along with the state symbol specified in the ACTION table.

**Reduce**: First, the handle is removed from the stack. For every grammar symbol on the stack there is a state symbol, so the number of symbols removed is twice the number of symbols in the handle. Next, the LHS of the rule is pushed onto the stack. Finally, the GOTO table is used to determine which state must be pushed onto the stack.

**Accept**: The parse is complete and no errors were found.

**Error**: The parser calls an error-handling routine.
LR Parser: Parsing Algorithm in Action

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S5</td>
<td>S4</td>
</tr>
<tr>
<td>1</td>
<td>S6</td>
<td>accept</td>
</tr>
<tr>
<td>2</td>
<td>R2</td>
<td>S7</td>
</tr>
<tr>
<td>3</td>
<td>R4</td>
<td>R4</td>
</tr>
<tr>
<td>4</td>
<td>S5</td>
<td>S4</td>
</tr>
<tr>
<td>5</td>
<td>R6</td>
<td>R6</td>
</tr>
<tr>
<td>6</td>
<td>S5</td>
<td>S4</td>
</tr>
<tr>
<td>7</td>
<td>S5</td>
<td>S4</td>
</tr>
<tr>
<td>8</td>
<td>S6</td>
<td>S11</td>
</tr>
<tr>
<td>9</td>
<td>R1</td>
<td>S7</td>
</tr>
<tr>
<td>10</td>
<td>R3</td>
<td>R3</td>
</tr>
<tr>
<td>11</td>
<td>R5</td>
<td>R5</td>
</tr>
</tbody>
</table>

Trace for input: id + id * id

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>id5</td>
<td>id * id $</td>
<td>Shift 5</td>
</tr>
<tr>
<td>0F3</td>
<td>id * id $</td>
<td>Reduce 4 (use GOTO[0, T])</td>
</tr>
<tr>
<td>0T2</td>
<td>id * id $</td>
<td>Reduce 2 (use GOTO[0, E])</td>
</tr>
<tr>
<td>0E1</td>
<td>id * id $</td>
<td>Shift 6</td>
</tr>
<tr>
<td>0E1+6</td>
<td>id * id$</td>
<td>Shift 5</td>
</tr>
<tr>
<td>0E1+6id5</td>
<td>* id $</td>
<td>Reduce 6 (use GOTO[6, F])</td>
</tr>
<tr>
<td>0E1+6F3</td>
<td>id $</td>
<td>Reduce 4 (use GOTO[6, T])</td>
</tr>
<tr>
<td>0E1+6T9</td>
<td>* id $</td>
<td>Shift 7</td>
</tr>
<tr>
<td>0E1+6T9*7id5</td>
<td>id $</td>
<td>Shift 5</td>
</tr>
<tr>
<td>0E1+6T9*7F10</td>
<td>$</td>
<td>Reduce 6 (use GOTO[7, F])</td>
</tr>
<tr>
<td>0E1+6T9 $</td>
<td>Reduce 3 (use GOTO[6, F])</td>
<td></td>
</tr>
<tr>
<td>0E1 $</td>
<td>Reduce 1 (use GOTO[0, E])</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Accept</td>
<td></td>
</tr>
</tbody>
</table>

R₄ means reduce using rule 4; S₆ means shift the next input symbol onto the stack and push state 6. Empty positions in the ACTION table indicate syntax errors.
Obtaining the Rightmost Derivation

Reduce actions

Reduce 6 (use GOTO[0, F])
Reduce 4 (use GOTO[0, T])
Reduce 2 (use GOTO[0, E])
Reduce 6 (use GOTO[6, F])
Reduce 4 (use GOTO[6, T])
Reduce 6 (use GOTO[7, F])
Reduce 3 (use GOTO[6, T])
Reduce 1 (use GOTO[0, E])

Reverse Reduce actions

Reduce 1 (use GOTO[0, E]) E : E + T
Reduce 3 (use GOTO[6, T]) T : T * F
Reduce 6 (use GOTO[7, F]) F : id
Reduce 4 (use GOTO[6, T]) T : F
Reduce 6 (use GOTO[6, F]) F : id
Reduce 2 (use GOTO[0, E]) E : T
Reduce 4 (use GOTO[0, T]) T : F
Reduce 6 (use GOTO[0, F]) F : Id

Rightmost Derivation

E
=> E + T
=> E + T * F
=> E + T * id
=> E + F * id
=> T + id * id
=> F + id * id
=> id + id * id

Parse Tree may be obtained easily from rightmost derivation
**LR Parser: Parsing Algorithm in Action: Another Example**

1. $E : E + T$
2. $E : T$
3. $T : T * F$
4. $T : F$
5. $F : ( E )$
6. $F : id$

**Trace for input:** $id + (* id id)$

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S5</td>
<td>S4</td>
</tr>
<tr>
<td>1</td>
<td>S6</td>
<td>accept</td>
</tr>
<tr>
<td>2</td>
<td>R2 S7</td>
<td>R2 R2</td>
</tr>
<tr>
<td>3</td>
<td>R4 R4</td>
<td>R4 R4</td>
</tr>
<tr>
<td>4</td>
<td>S5 S4</td>
<td>8 2 3</td>
</tr>
<tr>
<td>5</td>
<td>R6 R6</td>
<td>R6 R6</td>
</tr>
<tr>
<td>6</td>
<td>S5 S4</td>
<td>9 3</td>
</tr>
<tr>
<td>7</td>
<td>S5 S4</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>S6 S11</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>R1 S7</td>
<td>R1 R1</td>
</tr>
<tr>
<td>10</td>
<td>R3 R3</td>
<td>R3 R3</td>
</tr>
<tr>
<td>11</td>
<td>R5 R5</td>
<td>R5 R5</td>
</tr>
</tbody>
</table>

**Action**

- **R4** means reduce using rule 4; **S6** means shift the next input symbol onto the stack and push state 6. Empty positions in the **ACTION** table indicate syntax errors.