Left Recursion Removal and Left Factoring
Motivating example

- In this lecture we discuss techniques (that sometimes work) to convert a grammar that is not LL(1) into an equivalent grammar that is LL(1).

- Consider the following grammar:

  \[
  \begin{align*}
  \text{exp} & \rightarrow \text{exp} \text{ addop} \text{ term} \\
  & \mid \text{term} \\
  \text{addop} & \rightarrow + \mid - \\
  \text{term} & \rightarrow \text{term} \text{ mulop} \text{ factor} \\
  & \mid \text{factor} \\
  \text{mulop} & \rightarrow \ast \\
  \text{factor} & \rightarrow (\text{exp}) \\
  & \mid \text{number}
  \end{align*}
  \]

- This grammar is not LL(1) since \text{number} is in First(\text{exp}) and in First(\text{term}).

- Thus in the entry M[exp, number] in the LL(1) parsing table we will have the entries exp \rightarrow exp addop term and exp \rightarrow term.

- The problem is the presence of the left recursive rule exp \rightarrow exp addop term \mid term.

- Thus in order to try to convert this grammar into an LL(1) grammar, we remove the left recursion from this grammar.
In this lecture we discuss techniques (that sometimes work) to convert a grammar that is not LL(1) into an equivalent grammar that is LL(1).

- Consider the following grammar:

```plaintext
exp → exp addop term | term
addop → + | −
term → term mulop factor | factor
mulop → ∗
factor → ( exp ) | number
```

- This grammar is not LL(1) since `number` is in First(`exp`) and in First(`term`).
- Thus in the entry `M[exp, number]` in the LL(1) parsing table we will have the entries `exp → exp addop term` and `exp → term`.
- The problem is the presence of the left recursive rule `exp → exp addop term | term`.
- Thus in order to try to convert this grammar into an LL(1) grammar, we remove the left recursion from this grammar.
In this lecture we discuss techniques (that sometimes work) to convert a grammar that is not LL(1) into an equivalent grammar that is LL(1).

- Consider the following grammar:
  
  \[
  \begin{align*}
  exp & \rightarrow exp \ addop \ term \mid term \\
  addop & \rightarrow + \mid - \\
  term & \rightarrow term \ mulop \ factor \mid factor \\
  mulop & \rightarrow \ast \\
  factor & \rightarrow ( \ exp ) \mid number
  \end{align*}
  \]

- This grammar is not LL(1) since \textit{number} is in \texttt{First(exp)} and \texttt{First(term)}.

- Thus in the entry \texttt{M[exp, number]} in the LL(1) parsing table we will have the entries \texttt{exp \rightarrow exp addop term} and \texttt{exp \rightarrow term}.

- The problem is the presence of the left recursive rule \texttt{exp \rightarrow exp addop term | term}.

- Thus in order to try to convert this grammar into an LL(1) grammar, we remove the left recursion from this grammar.
In this lecture we discuss techniques (that sometimes work) to convert a grammar that is not LL(1) into an equivalent grammar that is LL(1).

- Consider the following grammar:
  
  \[
  \begin{align*}
  \text{exp} & \rightarrow \text{exp addop term} \mid \text{term} \\
  \text{addop} & \rightarrow + \mid - \\
  \text{term} & \rightarrow \text{term mulop factor} \mid \text{factor} \\
  \text{mulop} & \rightarrow * \\
  \text{factor} & \rightarrow ( \text{exp} ) \mid \text{number}
  \end{align*}
  \]

- This grammar is not LL(1) since \text{number} is in First(\text{exp}) and in First(\text{term}).
In this lecture we discuss techniques (that sometimes work) to convert a grammar that is not LL(1) into an equivalent grammar that is LL(1).

- Consider the following grammar:

  \[
  \begin{align*}
  \text{exp} & \rightarrow \text{exp addop term} \mid \text{term} \\
  \text{addop} & \rightarrow + \mid - \\
  \text{term} & \rightarrow \text{term mulop factor} \mid \text{factor} \\
  \text{mulop} & \rightarrow * \\
  \text{factor} & \rightarrow ( \text{exp} ) \mid \text{number}
  \end{align*}
  \]

- This grammar is not LL(1) since \text{number} is in First(exp) and in First(term).

- Thus in the entry \( M[\text{exp}, \text{number}] \) in the \( LL(1) \) parsing table we will have the entries \( \text{exp} \rightarrow \text{exp addop term} \) and \( \text{exp} \rightarrow \text{term} \).
Motivating example

- In this lecture we discuss techniques (that sometimes work) to convert a grammar that is not LL(1) into an equivalent grammar that is LL(1).
- Consider the following grammar:
  
  \[
  \begin{align*}
  \text{exp} & \rightarrow \text{exp addop term} \mid \text{term} \\
  \text{addop} & \rightarrow + \mid - \\
  \text{term} & \rightarrow \text{term mulop factor} \mid \text{factor} \\
  \text{mulop} & \rightarrow * \\
  \text{factor} & \rightarrow ( \text{exp} ) \mid \text{number}
  \end{align*}
  \]

- This grammar is not LL(1) since \text{number} is in First(\text{exp}) and in First(\text{term}).
- Thus in the entry \( M[\text{exp, number}] \) in the \( LL(1) \) parsing table we will have the entries \( \text{exp} \rightarrow \text{exp addop term} \) and \( \text{exp} \rightarrow \text{term} \).
- The problem is the presence of the left recursive rule \( \text{exp} \rightarrow \text{exp addop term} \mid \text{term} \).
In this lecture we discuss techniques (that sometimes work) to convert a grammar that is not LL(1) into an equivalent grammar that is LL(1).

- Consider the following grammar:
  
  \[ \text{exp} \rightarrow \text{exp addop term} \mid \text{term} \]
  \[ \text{addop} \rightarrow + \mid - \]
  \[ \text{term} \rightarrow \text{term mulop factor} \mid \text{factor} \]
  \[ \text{mulop} \rightarrow * \]
  \[ \text{factor} \rightarrow ( \text{exp} ) \mid \text{number} \]

- This grammar is not LL(1) since \text{number} is in First(\text{exp}) and in First(\text{term}).
- Thus in the entry \( M[\text{exp}, \text{number}] \) in the LL(1) parsing table we will have the entries \( \text{exp} \rightarrow \text{exp addop term} \) and \( \text{exp} \rightarrow \text{term} \).
- The problem is the presence of the left recursive rule \( \text{exp} \rightarrow \text{exp addop term} \mid \text{term} \).
- Thus in order to try to convert this grammar into an LL(1) grammar, we remove the left recursion from this grammar.
Before we look at the technique of removing left recursion from a grammar, we first discuss left recursion in general.
Before we look at the technique of removing left recursion from a grammar, we first discuss left recursion in general.

Grammar rules in BNF provide for concatenation and choice but no specific operation equivalent to the \( * \) of regular expressions are provided.
Before we look at the technique of removing left recursion from a grammar, we first discuss left recursion in general.

Grammar rules in BNF provide for concatenation and choice but no specific operation equivalent to the $*$ of regular expressions are provided.

We can obtain repetition by using for example rules of the form

\[ A \rightarrow Aa | a \]

Both these grammars generate \( \{a^n | n \geq 1\} \).

We call the rule \( A \rightarrow Aa | a \) left recursive and \( A \rightarrow aA | a \) right recursive.
Before we look at the technique of removing left recursion from a grammar, we first discuss left recursion in general.

Grammar rules in BNF provide for concatenation and choice but no specific operation equivalent to the $*$ of regular expressions are provided.

We can obtain repetition by using for example rules of the form

\[ A \rightarrow Aa \mid a \]
Before we look at the technique of removing left recursion from a grammar, we first discuss left recursion in general.

Grammar rules in BNF provide for concatenation and choice but no specific operation equivalent to the $\ast$ of regular expressions are provided.

We can obtain repetition by using for example rules of the form

$A \rightarrow Aa \mid a$ or
Before we look at the technique of removing left recursion from a grammar, we first discuss left recursion in general.

Grammar rules in BNF provide for concatenation and choice but no specific operation equivalent to the $\ast$ of regular expressions are provided.

We can obtain repetition by using for example rules of the form

$A \rightarrow Aa \mid a$ or $A \rightarrow aA \mid a$
Before we look at the technique of removing left recursion from a grammar, we first discuss left recursion in general.

Grammar rules in BNF provide for concatenation and choice but no specific operation equivalent to the $*$ of regular expressions are provided.

We can obtain repetition by using for example rules of the form

$$A \rightarrow Aa \mid a$$
$$A \rightarrow aA \mid a$$

Both these grammars generate $\{a^n \mid n \geq 1\}$. 
Before we look at the technique of removing left recursion from a grammar, we first discuss left recursion in general.

Grammar rules in BNF provide for concatenation and choice but no specific operation equivalent to the * of regular expressions are provided.

We can obtain repetition by using for example rules of the form:

\[ A \to Aa \mid a \text{ or } A \to aA \mid a \]

Both these grammars generate \( \{a^n \mid n \geq 1\} \).

We call the rule \( A \to Aa \mid a \) left recursive and \( A \to aA \mid a \) right recursive.
In general, rules of the form

\[ A \rightarrow A \alpha \mid \beta \]

are called left recursive, and rules of the form

\[ A \rightarrow \alpha A \mid \beta \]

are right recursive.
In general, rules of the form
\[ A \rightarrow A\alpha \mid \beta \] are called left recursive.
In general, rules of the form
\[ A \rightarrow A\alpha \mid \beta \] are called left recursive
and rules of the form
\[ A \rightarrow \alpha A \mid \beta \] right recursive.

Grammars equivalent to the regular expression \( a^* \) are given by
\[ A \rightarrow Aa \mid \varepsilon \]
In general, rules of the form
\[ A \rightarrow A\alpha \mid \beta \] are called left recursive and rules of the form
\[ A \rightarrow \alpha A \mid \beta \] right recursive.

Grammars equivalent to the regular expression \( a^* \) are given by
pause
\[ A \rightarrow Aa \mid \varepsilon \] or
In general, rules of the form
\[ A \rightarrow A\alpha \mid \beta \]
are called left recursive and rules of the form
\[ A \rightarrow \alpha A \mid \beta \]
right recursive.

Grammars equivalent to the regular expression \( a^* \) are given by
pause
\[ A \rightarrow Aa \mid \varepsilon \] or
\[ A \rightarrow aA \mid \varepsilon \]
Notice that left recursive rules ensure that expressions associate on the left.
Notice that left recursive rules ensure that expressions associate on the left.

The parse tree for the expression $34 - 3 - 42$ in the grammar
Notice that left recursive rules ensure that expressions associate on the left.

The parse tree for the expression $34 - 3 - 42$ in the grammar

\[
\begin{align*}
exp & \rightarrow \ exp \ addop \ term \ | \ term \\
addop & \rightarrow + \ | - \\
term & \rightarrow term \ mulop \ factor \ | \ factor \\
mulop & \rightarrow * \\
factor & \rightarrow ( \ exp \ ) \ | \ number
\end{align*}
\]
Notice that left recursive rules ensure that expressions associate on the left.

The parse tree for the expression $34 - 3 - 42$ in the grammar

\[
\begin{align*}
exp & \rightarrow \text{exp addop term} \mid \text{term} \\
\text{addop} & \rightarrow + \mid - \\
\text{term} & \rightarrow \text{term mulop factor} \mid \text{factor} \\
\text{mulop} & \rightarrow * \\
\text{factor} & \rightarrow ( \text{exp} ) \mid \text{number}
\end{align*}
\]

is for example given by
Notice that left recursive rules ensure that expressions associate on the left.

The parse tree for the expression $34 - 3 - 42$ in the grammar

$$
\begin{align*}
\text{exp} & \rightarrow \text{exp} \ \text{addop} \ \text{term} \ | \ \text{term} \\
\text{addop} & \rightarrow + \ | - \\
\text{term} & \rightarrow \text{term} \ \text{mulop} \ \text{factor} \ | \ \text{factor} \\
\text{mulop} & \rightarrow * \\
\text{factor} & \rightarrow ( \ \text{exp} \ ) \ | \ \text{number}
\end{align*}
$$

is for example given by
In the rule
In the rule

\[ exp \rightarrow exp + term \mid exp - term \mid term \]

we have **immediate left recursion** and in
In the rule

\[ \text{exp} \rightarrow \text{exp} + \text{term} \mid \text{exp} - \text{term} \mid \text{term} \]

we have **immediate left recursion** and in

\[
A \rightarrow B \ a \mid A\ a \mid c \\
B \rightarrow B\ b \mid A\ b \mid d
\]

we have **indirect left recursion**.

We only consider how to remove immediate left recursion.
Consider again the rule
Consider again the rule

\[ \text{exp} \rightarrow \text{exp addop term} \mid \text{term} \]
Consider again the rule

\[ \text{exp} \rightarrow \text{exp addop term} \mid \text{term} \]

We rewrite this rule as
Consider again the rule

\[ \text{exp} \rightarrow \text{exp addop term} \mid \text{term} \]

We rewrite this rule as

\[ \text{exp} \rightarrow \text{term exp}' \]
\[ \text{exp}' \rightarrow \text{addop term exp}' \mid \varepsilon \]

to remove the left recursion.

In general if we have productions of the form
Consider again the rule

\[ \text{exp} \rightarrow \text{exp addop term} \mid \text{term} \]

We rewrite this rule as

\[ \text{exp} \rightarrow \text{term exp}' \\
\text{exp}' \rightarrow \text{addop term exp}' \mid \epsilon \]

to remove the left recursion.

In general if we have productions of the form

\[ A \rightarrow A \alpha_1 \mid \ldots \mid A \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m \]
Consider again the rule

\[ \text{exp} \rightarrow \text{exp} \text{ addop term} \mid \text{term} \]

We rewrite this rule as

\[ \text{exp} \rightarrow \text{term exp}' \]
\[ \text{exp}' \rightarrow \text{addop term exp}' \mid \varepsilon \]

to remove the left recursion.

In general if we have productions of the form

\[ A \rightarrow A \alpha_1 \mid \ldots \mid A \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m \]

we rewrite this as
Consider again the rule

\[ \text{exp} \rightarrow \text{exp addop term} \mid \text{term} \]

We rewrite this rule as

\[ \text{exp} \rightarrow \text{term exp}' \]
\[ \text{exp}' \rightarrow \text{addop term exp}' \mid \varepsilon \]

to remove the left recursion.

In general if we have productions of the form

\[ A \rightarrow A \alpha_1 \mid \ldots \mid A \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m \]

we rewrite this as

\[ A \rightarrow \beta_1 A' \mid \ldots \mid \beta_m A' \]
\[ A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon \]
in order to remove the left recursion.
If we remove the left recursion from the rule
If we remove the left recursion from the rule

\[ \text{exp} \rightarrow \text{exp} + \text{term} \mid \text{exp} - \text{term} \mid \text{term} \]

we obtain
If we remove the left recursion from the rule

\[ exp \rightarrow exp + \text{term} \mid exp - \text{term} \mid \text{term} \]

we obtain

\[ exp \rightarrow \text{term} \ exp' \]

\[ exp' \rightarrow + \text{term} \ exp' \mid - \text{term} \ exp' \mid \epsilon \]
If we remove left recursion from the grammar

\[
\begin{align*}
exp & \rightarrow \ exp \ addop \ term \mid term \\
addop & \rightarrow + \mid - \\
term & \rightarrow \ term \ mulop \ factor \mid factor \\
mulop & \rightarrow * \\
factor & \rightarrow ( \ exp ) \mid number
\end{align*}
\]
If we remove left recursion from the grammar

\[
\begin{align*}
exp & \rightarrow \exp \ addop \ term \mid term \\
addop & \rightarrow + \mid - \\
term & \rightarrow \term \ mulop \ factor \mid factor \\
mulop & \rightarrow * \\
factor & \rightarrow ( \ exp ) \mid number
\end{align*}
\]

we obtain the grammar
If we remove left recursion from the grammar

\[
\begin{align*}
exp & \rightarrow \, exp \, addop \, term \, | \, term \\
addop & \rightarrow \, + \, | \, - \\
term & \rightarrow \, term \, mulop \, factor \, | \, factor \\
mulop & \rightarrow \, * \\
factor & \rightarrow \, ( \, exp \, ) \, | \, number
\end{align*}
\]

we obtain the grammar

\[
\begin{align*}
exp & \rightarrow \, term \, exp' \\
exp' & \rightarrow \, addop \, term \, exp' \, | \, \varepsilon \\
addop & \rightarrow \, + \, | \, - \\
term & \rightarrow \, factor \, term' \\
term' & \rightarrow \, mulop \, factor \, term' \, | \, \varepsilon \\
mulop & \rightarrow \, * \\
factor & \rightarrow \, ( \, exp \, ) \, | \, number
\end{align*}
\]
Now consider the parse tree for $3 - 4 - 5$
Now consider the parse tree for $3 - 4 - 5$

This tree no longer expresses the left associativity of subtraction. Nevertheless, a parser should still construct the appropriate left associative syntax tree.
Now consider the parse tree for $3 - 4 - 5$

This tree no longer expresses the left associativity of subtraction.
Now consider the parse tree for $3 - 4 - 5$

This tree no longer expresses the left associativity of subtraction.

Nevertheless, a parser should still construct the appropriate left associative syntax tree.
We obtain the syntax tree by removing all the unnecessary information from the parse tree. A parser will usually construct a syntax tree and not a parse tree.
We obtain the syntax tree by removing all the unnecessary information from the parse tree. A parser will usually construct a syntax tree and not a parse tree.
Left factoring is required when two or more grammar rule choices share a common prefix string, as in the rule

\[ A \rightarrow \alpha\beta | \alpha\gamma \]
Left Factoring

- Left factoring is required when two or more grammar rule choices share a common prefix string, as in the rule

\[ A \rightarrow \alpha \beta | \alpha \gamma \]
Left Factoring

- Left factoring is required when two or more grammar rule choices share a common prefix string, as in the rule

\[ A \rightarrow \alpha \beta \mid \alpha \gamma \]

- Obviously, an LL(1) parser cannot distinguish between the production choices in such a situation.
Left Factoring

- Left factoring is required when two or more grammar rule choices share a common prefix string, as in the rule
  \[ A \rightarrow \alpha \beta | \alpha \gamma \]

- Obviously, an LL(1) parser cannot distinguish between the production choices in such a situation.

- In the following example we have exactly this problem:
Left Factoring

- Left factoring is required when two or more grammar rule choices share a common prefix string, as in the rule
\[ A \rightarrow \alpha \beta | \alpha \gamma \]

- Obviously, an LL(1) parser cannot distinguish between the production choices in such a situation.

- In the following example we have exactly this problem:

\[ if\text{-stmt} \rightarrow if\ (\ exp\ )\ statement\ |\ if\ (\ exp\ )\ statement\ else\ statement \]
Consider the following grammar of if-statements:

The left factored form of this grammar is:

- `if-stmt → if(exp) statement`  
- `else-part → else statement`  
- `else-part → ε`
Consider the following grammar of if-statements:

\[ if-stmt \rightarrow if \ (exp) \ statement \mid if \ (exp) \ statement \ else \ statement \]
Consider the following grammar of if-statements:

\[
if-stmt \rightarrow \text{if ( exp ) statement} \mid \text{if ( exp ) statement else statement}
\]

The left factored form of this grammar is
Consider the following grammar of if-statements:

\[ if-stmt \rightarrow if \ ( \ exp \ ) \ statement \mid if \ ( \ exp \ ) \ statement \ else \ statement \]

The left factored form of this grammar is

\[ if-stmt \rightarrow if \ ( \ exp \ ) \ statement \ else-part \\
else-part \rightarrow else \ statement \mid \epsilon \]
Here is a typical example where a programming language fails to be LL(1):
Here is a typical example where a programming language fails to be LL(1):

\[
\begin{align*}
\text{statement} & \rightarrow \text{assign-stmt} \mid \text{call-stmt} \mid \text{other} \\
\text{assign-stmt} & \rightarrow \text{identifier} \ := \ \text{exp} \\
\text{call-stmt} & \rightarrow \text{identifier} \ (\ \text{exp-list} \ )
\end{align*}
\]
Here is a typical example where a programming language fails to be LL(1):

\[
\begin{align*}
\text{statement} & \rightarrow \text{assign-stmt} \mid \text{call-stmt} \mid \text{other} \\
\text{assign-stmt} & \rightarrow \text{identifier} := \text{exp} \\
\text{call-stmt} & \rightarrow \text{identifier} ( \text{exp-list} )
\end{align*}
\]

This grammar is not in a form that can be left factored. We must first replace \textit{assign-stmt} and \textit{call-stmt} by the right-hand sides of their defining productions:
Here is a typical example where a programming language fails to be LL(1):

\[
\begin{align*}
\text{statement} & \rightarrow \text{assign-stmt} \mid \text{call-stmt} \mid \text{other} \\
\text{assign-stmt} & \rightarrow \text{identifier} \ := \ \text{exp} \\
\text{call-stmt} & \rightarrow \text{identifier} \ ( \ \text{exp-list} \ )
\end{align*}
\]

This grammar is not in a form that can be left factored. We must first replace \textit{assign-stmt} and \textit{call-stmt} by the right-hand sides of their defining productions:

\[
\begin{align*}
\text{statement} & \rightarrow \text{identifier} \ := \ \text{exp} \mid \text{identifier} \ ( \ \text{exp-list} \ ) \mid \text{other}
\end{align*}
\]

Then we left factor to obtain:
Here is a typical example where a programming language fails to be LL(1):

```plaintext
statement → assign-stmt | call-stmt | other
assign-stmt → identifier := exp
call-stmt → identifier ( exp-list )
```

This grammar is not in a form that can be left factored. We must first replace `assign-stmt` and `call-stmt` by the right-hand sides of their defining productions:

```plaintext
statement → identifier := exp | identifier ( exp-list ) | other
```

Then we left factor to obtain:

```plaintext
statement → identifier statement' | other
statement' → := exp | ( exp-list )
```
Here is a typical example where a programming language fails to be LL(1):

\[
\begin{align*}
\text{statement} & \rightarrow \text{assign-stmt} \mid \text{call-stmt} \mid \text{other} \\
\text{assign-stmt} & \rightarrow \text{identifier} \ := \ \text{exp} \\
\text{call-stmt} & \rightarrow \text{identifier} \ ( \ \text{exp-list} )
\end{align*}
\]

This grammar is not in a form that can be left factored. We must first replace \text{assign-stmt} and \text{call-stmt} by the right-hand sides of their defining productions:

\[
\begin{align*}
\text{statement} & \rightarrow \text{identifier} \ := \ \text{exp} \mid \text{identifier} \ ( \ \text{exp-list} ) \mid \text{other}
\end{align*}
\]

Then we left factor to obtain:

\[
\begin{align*}
\text{statement} & \rightarrow \text{identifier} \ \text{statement}' \mid \text{other} \\
\text{statement}' & \rightarrow \ := \ \text{exp} \mid ( \ \text{exp-list} )
\end{align*}
\]

Note how this obscures the semantics of call and assignment by separating the identifier from the actual call or assign action.