Eliminating left-recursion: three steps

Recall: A CFG is left-recursive if it includes a variable $A$ s.t.
$$A \Rightarrow A\alpha.$$  

We eliminate left-recursion in three steps.

- eliminate $\varepsilon$-productions (impossible to generate $\varepsilon$!)
- eliminate cycles ($A \Rightarrow A$)
- eliminate left-recursion

So we’ve got some constructions to learn.

Let’s try an example of eliminating $\varepsilon$-productions before we specify a construction. . .

Consider the CFG below.

$$\begin{align*}
S & \to XX \mid Y \\
X & \to aXb \mid \varepsilon \\
Y & \to aYb \mid Z \\
Z & \to bZa \mid \varepsilon
\end{align*}$$

Notice that

$$S \Rightarrow \varepsilon \quad X \Rightarrow \varepsilon \quad Y \Rightarrow \varepsilon \quad Z \Rightarrow \varepsilon$$

Hence, all the variables in this grammar are what we will call “nullable.” So in order to eliminate the $\varepsilon$-productions in this grammar, we must alter the grammar to take into account the fact that instances of these variables in a derivation may eventually be replaced by $\varepsilon$. So, for instance, we will replace

$$Z \to bZa \mid \varepsilon$$

with

$$Z \to bZa \mid ba.$$  

After elimination of $\varepsilon$-productions, we obtain

$$\begin{align*}
S & \to XX \mid X \mid Y \\
X & \to aXb \mid ab \\
Y & \to aYb \mid ab \mid Z \\
Z & \to bZa \mid ba
\end{align*}$$
Eliminating $\epsilon$-productions

Given a CFG $G = (V, \Sigma, S, P)$, a variable $A \in V$ is \textit{nullable} if

$$A \Rightarrow^* \epsilon.$$ 

The main step in the $\epsilon$-production elimination algorithm then is that the set $P$ of productions is replaced with the set $P_\epsilon$ of all productions

$$A \rightarrow \beta$$

s.t. $A \neq \beta$, $\beta \neq \epsilon$, and $P$ includes a production

$$A \rightarrow \alpha$$

s.t. $\beta$ can be obtained from $\alpha$ by deleting zero or more occurrences of nullable variables.

\textbf{Example}  Applying $\epsilon$-production elimination to the CFG

$$G = (\{S\}, \{a, b\}, S, \{ S \rightarrow aSb | \epsilon \})$$

yields the CFG

$$G_\epsilon = (\{S\}, \{a, b\}, S, \{ S \rightarrow aSb | ab \}).$$

$A$ is \textit{nullable} if $A \Rightarrow \epsilon$.

$P_\epsilon$ is the set of all productions

$$A \rightarrow \beta$$

s.t. $A \neq \beta$, $\beta \neq \epsilon$, and $P$ includes a production

$$A \rightarrow \alpha$$

s.t. $\beta$ can be obtained from $\alpha$ by deleting zero or more occurrences of nullable variables.

\textbf{Example}  Let’s apply $\epsilon$-production elimination to

$$S \rightarrow XZ$$

$$X \rightarrow aXb | \epsilon$$

$$Z \rightarrow aZ | ZX | \epsilon$$

What are the nullable variables?

What are the new productions?
Eliminating $\epsilon$-productions can greatly increase the size of a grammar.

**Example** Eliminating $\epsilon$-productions from

\[
S \to A_1 A_2 \cdots A_n \\
A_1 \to \epsilon \\
A_2 \to \epsilon \\
\vdots \\
A_n \to \epsilon
\]

increases the number of productions from $n + 1$ to $2^n - 1$.

What about ambiguity?

**Claim** If $G$ is unambiguous, so is $G_\epsilon$.

---

**Eliminating cycles**

A grammar has a *cycle* if there is a variable $A$ s.t.

\[
A \Rightarrow A.
\]

We’ll call such variables *cyclic*.

If a grammar has no $\epsilon$-productions, then all cycles can be eliminated from $G$ without affecting the language generated, using a construction we will specify in a moment.

Let’s try an example first. Consider the grammar with productions

\[
S \to X \mid Xb \mid SS
\]
\[
X \to S \mid a
\]

We have $S \Rightarrow S$ and $X \Rightarrow X$.

We can eliminate occurrences of cyclic variables as rhs’s of productions, thus eliminating cycles.

\[
S \to a \mid Xb \mid SS
\]
\[
X \to Xb \mid SS \mid a
\]
Given a CFG $G = (V, \Sigma, S, P)$, the main step in the cycle elimination algorithm is to replace the set $P$ of productions with the set $P_c$ of productions obtained from $P$ by replacing

- each production $A \rightarrow B$ where $B$ is cyclic
- with new productions $A \rightarrow \alpha$ s.t. $\alpha$ is not a cyclic variable and there is a production $C \rightarrow \alpha$ s.t. $B \Rightarrow C$.

The resulting CFG is $G_c = (V, \Sigma, S, P_c)$.

Note: You can probably convince yourself that $P_c$ has no cycles, and that $G_c$ generates the same languages as $G$.

Let’s try the construction...

\[ S \rightarrow X \mid Xb \mid Ya \]
\[ X \rightarrow Y \mid b \]
\[ Y \rightarrow X \mid a \]

Which variables are cyclic?

Which productions will be replaced?

With what?

Replace the set $P$ of productions with the set $P_c$ of productions obtained from $P$ by replacing

- each production $A \rightarrow B$ where $B$ is cyclic
- with new productions $A \rightarrow \alpha$ s.t. $\alpha$ is not a cyclic variable and there is a production $C \rightarrow \alpha$ s.t. $B \Rightarrow C$.

Let’s consider an example illustrating the importance of requiring that there be no $\epsilon$-productions.

\[ S \rightarrow a \mid SS \mid \epsilon \]

Notice that $S$ is a cyclic variable, since

\[ S \Rightarrow SS \Rightarrow S. \]

But $S$ never appears as the rhs of a production, so the construction does nothing.

(BTW What does the $\epsilon$-elimination algorithm do to this grammar?)
Eliminating “immediate” left recursion

Let’s begin with an easy example, already considered:

\[ A \rightarrow Ab \mid b \]

This grammar is left-recursive, since

\[ A \Rightarrow Ab . \]

In this case, we can eliminate left recursion as follows:

\[ A \rightarrow bA' \]
\[ A' \rightarrow bA' \mid \epsilon \]

More generally, we can eliminate “immediate” left recursion as follows. If

\[ A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \cdots \beta_n \]

represents all the \( A \)-productions of the grammar, and no \( \beta_i \) begins with \( A \), then we can replace these \( A \)-productions by

\[ A \rightarrow \beta_1A' \mid \beta_2A' \cdots \beta_nA' \]
\[ A' \rightarrow \alpha_1A' \mid \alpha_2A' \cdots \alpha_mA' \mid \epsilon \]

If

\[ A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \cdots \beta_n \]

represents all the \( A \)-productions of the grammar, and no \( \beta_i \) begins with \( A \), then we can replace these \( A \)-productions by

\[ A \rightarrow \beta_1A' \mid \beta_2A' \cdots \beta_nA' \]
\[ A' \rightarrow \alpha_1A' \mid \alpha_2A' \cdots \alpha_mA' \mid \epsilon \]

If our grammar has \( S \)-productions

\[ S \rightarrow SX \mid SSb \mid XS \mid a \]

we can replace them with

\[ S \rightarrow XSS' \mid aS' \]
\[ S' \rightarrow XS' \mid SbS' \mid \epsilon \]

Notice that this construction can fail to eliminate left-recursion if we have the production

\[ A \rightarrow A ! \]

For instance,

\[ A \rightarrow A \mid Ab \mid b \]

becomes

\[ A \rightarrow bA' \]
\[ A' \rightarrow A' \mid bA' \mid \epsilon \]
If
\[ A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \]
represents all the \( A \)-productions of the grammar, and no \( \beta_i \) begins with \( A \), then we can replace these \( A \)-productions by
\[
A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\
A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon
\]

Another interesting special case. What if there are no \( \beta_i \)'s?

For example,
\[ A \rightarrow AA \mid Ab \]

Then everything that can be derived from \( A \) has a variable in it, so \( A \) cannot appear in a derivation of a sentence.

And the construction handles this in an interesting way, yielding
\[ A' \rightarrow AA' \mid bA' \mid \epsilon \]

but no \( A \)-productions.

Notice also that this construction works only “locally”:

That is, indirect recursion is not eliminated.

For example, if we apply this construction to both variables in
\[
S \rightarrow SX \mid SSb \mid XS \mid a \\
X \rightarrow Sa \mid Xb
\]

we obtain
\[
S \rightarrow XSS' \mid aS' \\
S' \rightarrow XS' \mid SbS' \mid \epsilon \\
X \rightarrow SaX' \\
X' \rightarrow bX' \mid \epsilon
\]

and so have \( S \Rightarrow SaX'SS' \), for instance.
Here’s an algorithm that eliminates all left-recursion for any CFG without \( \epsilon \)-productions and without cycles.

Arrange the variables in some order \( A_1, A_2, \ldots, A_n \).

for \( i := 1 \) to \( n \) do begin
    for \( j := 1 \) to \( i - 1 \) do begin
        replace each production of the form \( A_i \rightarrow A_j \gamma \)
        by the productions \( A_i \rightarrow \delta_1 \gamma | \delta_2 \gamma | \cdots | \delta_k \gamma \)
        where \( A_j \rightarrow \delta_1 | \delta_2 | \cdots | \delta_k \) are all the current \( A_j \)-productions;
    end
    eliminate the immediate left recursion among the \( A_i \)-productions;
end

Consider the grammar

\[
S \rightarrow SX | SSb | XS | a \\
X \rightarrow Xb | Sa | b
\]

Let’s order the variables \( S, X \):

The first time through we simply eliminate immediate left recursion in \( S \)-productions, yielding

\[
S \rightarrow XSS' | aS' \\
S' \rightarrow XS' | SbS' | \epsilon \\
X \rightarrow Xb | Sa | b
\]

Arrange the variables in some order \( A_1, A_2, \ldots, A_n \).

for \( i := 1 \) to \( n \) do begin
    for \( j := 1 \) to \( i - 1 \) do begin
        replace each production of the form \( A_i \rightarrow A_j \gamma \)
        by the productions \( A_i \rightarrow \delta_1 \gamma | \delta_2 \gamma | \cdots | \delta_k \gamma \)
        where \( A_j \rightarrow \delta_1 | \delta_2 | \cdots | \delta_k \) are all the current \( A_j \)-productions;
    end
    eliminate the immediate left recursion among the \( A_i \)-productions;
end

So at this point we have grammar

\[
S \rightarrow XSS' | aS' \\
S' \rightarrow XS' | SbS' | \epsilon \\
X \rightarrow Xb | Sa | b
\]

and the next obligation is to replace the production

\[
X \rightarrow Sa
\]

with the productions

\[
X \rightarrow XSS'a | aS'a.
\]

We then eliminate immediate left recursion among

\[
X \rightarrow XSS'a | aS'a | Xb | b.
\]
Eliminating immediate left recursion among

\[ X \rightarrow XSS'a \mid Xb \mid b \mid aS'a \]

yields

\[ X \rightarrow bX' \mid aS'aX' \]
\[ X' \rightarrow SS'aX' \mid bX' \mid \epsilon \]

So the final result is

\[ S \rightarrow XSS' \mid aS' \]
\[ S' \rightarrow XS' \mid SbS' \mid \epsilon \]
\[ X \rightarrow bX' \mid aS'aX' \]
\[ X' \rightarrow SS'aX' \mid bX' \mid \epsilon \]

Let’s look at examples showing that this algorithm can fail if the grammar has \( \epsilon \)-productions or cycles.

In the simplest case, when there is only one variable, call it \( X \), the presence of a cycle implies that the grammar includes the production

\[ X \rightarrow X \, . \]

Moreover, the whole left recursion elimination algorithm reduces to elimination of immediate left recursion among \( X \)-productions.

And we have previously observed that our construction for immediate left recursion elimination is no good in the presence of \( X \rightarrow X \).

For example, if the grammar is

\[ X \rightarrow X \mid a \]

the construction for eliminating immediate left recursion yields

\[ X \rightarrow aX' \]
\[ X' \rightarrow X' \]

What about more complex cycles?
Try ordering $S, X$.

First step: eliminate immediate left recursion in $S$-productions.

There is none.

Next: replace production
\[ X \to S \]
with productions
\[ X \to X \mid b. \]

It remains only to eliminate immediate left recursion in the current $X$-productions, which are
\[ X \to X \mid b \mid a. \]

As before, the presence of production $X \to X$ breaks our construction — which yields
\[ X \to bX' \mid aX' \]
\[ X' \to X' \]

Here’s an example with an $\epsilon$-production and no cycles:
\[
\begin{align*}
S & \to XSa \mid b \\
X & \to \epsilon
\end{align*}
\]

Try order $S, X$.

First step: eliminate immediate left recursion in $S$-productions.

There is none.

Next: replace any $X$-productions whose rhs begins with $S$.

There are none.

Last: eliminate immediate left recursion in the current $X$-productions.

There is none.

So our left-recursion elimination algorithm leaves the grammar unchanged.

Yet $S \Rightarrow Sa$, so the grammar is left-recursive.
So now we can take any grammar and eliminate left-recursion (in three steps), making it suitable for top-down parsing (with backtracking!).

Notice that this works even for ambiguous grammars.

Next time we’ll define the main component of a top-down parser — the parsing table.

But practically speaking, we would also like to avoid backtracking.

Next time we’ll see how this can be done for top-down parsing.

We’ll define the class of LL(1) grammars, suitable for predictive parsing.

For next time

Read 4.4.