

ToC

- 1 Syntax Analysis: the problem
- 2 Theoretical Background
- 3 Syntax Analysis: solutions**
 - **Top-Down parsing**
 - Bottom-Up Parsing

Left Recursion

Left recursive grammars

A grammar \mathcal{G} is **left recursive** if it has a non terminal A such that there is a derivation $A \xRightarrow{*} A\alpha$ for some string α . **Top-down parsing strategies cannot handle left-recursive grammars**

Immediate left recursion

A grammar has an **immediate left recursion** if there is at least one production of the form $A \rightarrow A\alpha$. It is possible to transform the grammar still generating the same language and removing the left recursion. Consider the general case:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

where $n, m \geq 1$ and all β_i do not start with A . Equivalent productions are:

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A' \\ A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

Left Recursion

Left recursive grammars

A grammar \mathcal{G} is **left recursive** if it has a non terminal A such that there is a derivation $A \xRightarrow{*} A\alpha$ for some string α . **Top-down parsing strategies cannot handle left-recursive grammars**

Immediate left recursion

A grammar has an **immediate left recursion** if there is at least one production of the form $A \rightarrow A\alpha$. It is possible to transform the grammar still generating the same language and removing the left recursion. Consider the general case:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

where $n, m \geq 1$ and all β_i do not start with A . Equivalent productions are:

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A' \\ A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

Eliminating Left Recursion

The following is a general algorithm to eliminate left recursion at any level

Input: Grammar G with no cycles or ϵ – *productions*

Output: An equivalent grammar with no left recursion

Arrange the non terminals in some order A_1, A_2, \dots, A_n

for all $i \in [1 \dots n]$ **do**

for all $j \in [1 \dots i - 1]$ **do**

 replace each production of the form $A_i \rightarrow A_j \gamma$ by the
 productions $A_i \rightarrow \delta_1 \gamma | \delta_2 \gamma | \dots | \delta_k \gamma$ where $A_j \rightarrow \delta_1 | \delta_2 | \dots | \delta_k$ are all current
 A_j – *productions*

end for

 eliminate the immediate left recursion among the A_i – *productions*

end for

Left Factoring

Left Factoring

Left Factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing. When the choice between two alternative productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice

Transformation rule

In general the grammar:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

can be rewritten in:

$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 \mid \beta_2 \end{aligned}$$

In general find the longest prefix and then iterate till no two alternatives for a nonterminal have a common prefix

Left Factoring

Left Factoring

Left Factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing. When the choice between two alternative productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice

Transformation rule

In general the grammar:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

can be rewritten in:

$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 \mid \beta_2 \end{aligned}$$

In general find the longest prefix and then iterate till no two alternatives for a nonterminal have a common prefix

Top-down parsing

Top-down parsing

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string starting from the root and creating the nodes of the parse tree in pre-order (depth-first). Equivalently ... finding the left-most derivation for an input string.

Recursive descent parsing

A recursive descent (top-down) parsing consist of a set of procedures, one for each nonterminal.

function A

Choose an *A*-production, $A \rightarrow X_1 X_2 \cdots X_k$;

for all $i \in [1 \cdots k]$ **do**

if (X_i is a non terminal) **then** call procedure $X_i()$;

else if (X_i equals the current input symbol a) **then**

 advance the input to the next symbol;

else an error has occurred;

end if

end for

end function

Top-down parsing

Backtracking is expensive and not easy to manage. With grammar with no left-factoring and left-recursion we can do better:

At work

At each step of a top-down parsing the key problem is that of determining the production to be applied for a nonterminal.

Let's consider the usual sentence **id + id * id** and a suitable grammar for top-down parsing:

$$E \rightarrow TE' \quad E' \rightarrow +TE' | \epsilon \quad T \rightarrow FT' \quad T' \rightarrow *FT' | \epsilon \quad F \rightarrow (E) | \text{id}$$

FIRST and FOLLOW sets

$FIRST(\alpha)$	set of terminals that begin strings derived from α
$FOLLOW(A)$	set of terminals a that can appear immediately to the right of A in some sentential form
$nullable(X)$	it is true if it is possible to derive ϵ from X

FIRST

To compute $FIRST(X)$ for all grammar symbols X , apply the following rules until no more terminals or ϵ can be added to any $FIRST$ set

- 1 if X is a terminal, then $FIRST(X) = \{X\}$
- 2 if X is a non terminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $FIRST(X)$ if a is in $FIRST(Y_i)$, for some $i \leq k$, and ϵ is in all of $FIRST(Y_1) \cdots FIRST(Y_{i-1})$. If ϵ is in $FIRST(Y_j)$ for all $j = 1, 2, \dots, k$ then add ϵ to $FIRST(X)$. If Y_1 does not derive ϵ , then we add nothing more to $FIRST(X)$, but if $Y_1 \rightarrow^* \epsilon$, then we add $FIRST(Y_2)$, and so on.
- 3 if $X \rightarrow \epsilon$ is a production, then add ϵ to $FIRST(X)$

It is then possible to compute $FIRST$ for any string $X_1 X_2 \cdots X_k$

FIRST and FOLLOW sets

$FIRST(\alpha)$	set of terminals that begin strings derived from α
$FOLLOW(A)$	set of terminals a that can appear immediately to the right of A in some sentential form
$nullable(X)$	it is true if it is possible to derive ϵ from X

FIRST

To compute $FIRST(X)$ for all grammar symbols X , apply the following rules until no more terminals or ϵ can be added to any $FIRST$ set

- 1 if X is a terminal, then $FIRST(X) = \{X\}$
- 2 if X is a non terminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $FIRST(X)$ if a is in $FIRST(Y_i)$, for some $i \leq k$, and ϵ is in all of $FIRST(Y_1) \cdots FIRST(Y_{i-1})$. If ϵ is in $FIRST(Y_j)$ for all $j = 1, 2, \dots, k$ then add ϵ to $FIRST(X)$. If Y_1 does not derive ϵ , then we add nothing more to $FIRST(X)$, but if $Y_1 \rightarrow^* \epsilon$, then we add $FIRST(Y_2)$, and so on.
- 3 if $X \rightarrow \epsilon$ is a production, then add ϵ to $FIRST(X)$

It is then possible to compute $FIRST$ for any string $X_1 X_2 \cdots X_k$