Introduction to Syntax Analysis

Syntax Analysis

The *syntactic* or the *structural* correctness of a program is checked in the syntax analysis phase of compilation. The structural properties of language constructs can be specified in different ways. Different styles of specification are useful for different purposes.

Different Formalisms

- Syntax diagram (SD),
- Backus-Naur form (BNF), and
- Context-free grammar (CFG).

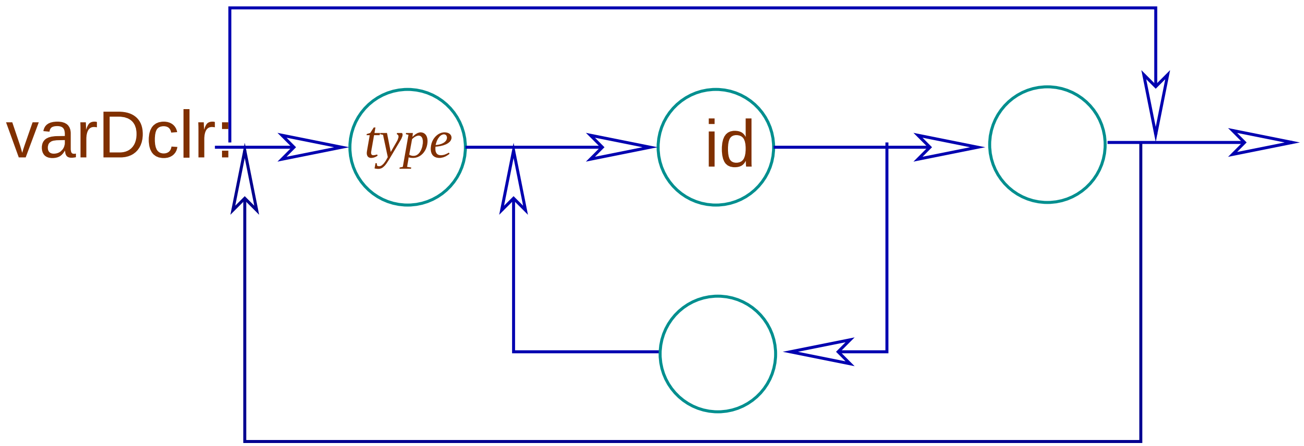
Example

We take an example of simple variable declaration in C language[a].

```
int a, b, c; float x, y;
```

[a] This part of syntax is actually a *regular expression*.

Syntax Diagram



Context-Free Grammar

$$
\begin{aligned}
<\mathsf{VDP}> \;&\rightarrow\; \varepsilon \mid <\mathsf{VD}><\mathsf{VD\_OPT}> \\
<\mathsf{VD}> \;&\rightarrow\; <\mathsf{TYPE}>\mathbf{id}<\mathsf{ID\_OPT}> \\
<\mathsf{ID\_OPT}> \;&\rightarrow\; \varepsilon \mid , \mathbf{id} <\mathsf{ID\_OPT}> \\
<\mathsf{VD\_OPT}> \;&\rightarrow\; ; \mid ; <\mathsf{VD}><\mathsf{VD\_OPT}> \\
<\mathsf{TYPE}> \;&\rightarrow\; \mathbf{int} \mid \mathbf{float} \mid \cdots
\end{aligned}
$$

Backus-Naur Form

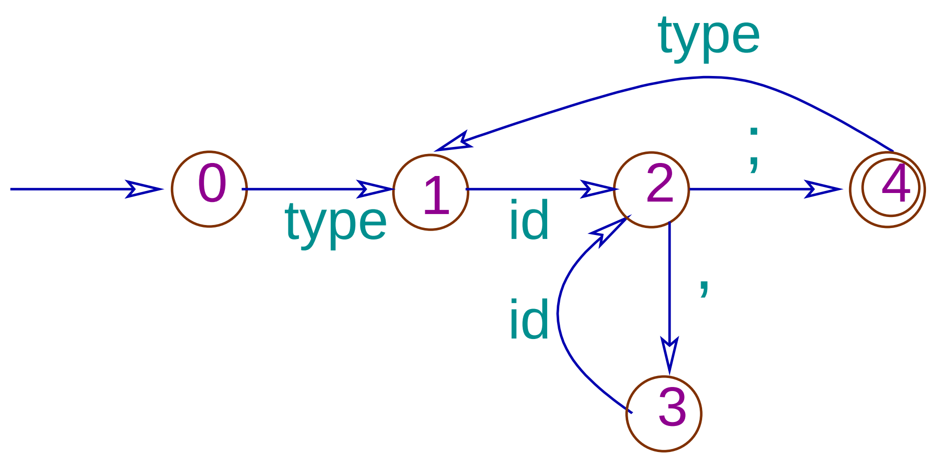
$$< \mathsf{VDP} > \;::=\; \varepsilon \mid\; < \mathsf{VD} >; \{ < \mathsf{VD} > ;\; \}$$

$$< \mathsf{VD} > \;::=\; < \mathsf{TYPE} > id \, \{ \, , \; id \, \}$$

This formalism is a beautiful mixture of CFG and regular expression.

Note

Our variable declaration is actually a regular language with the following state transition diagram:



Note

Different styles of specification have different purpose. SD is good for human understanding and visualization. The BNF is very compact. It is used for theoretical analysis and also in automatic parser generating softwares. But for most of our discussion we shall consider structural specification in the form of a *context-free grammar (CFG)*.

Note

There are non-context-free structural features of a programming language that are handled outside the formalism of grammar.

- Variable declaration and use: `... int sum ... sum = ...`, this is of the form $xwywz$ and is not context-free.
- Matching of actual and formal parameters of a function, matching of print format and the corresponding expressions etc.

Specification to Recognizer

The *syntactic specification* of a programming language, written as a *context-free grammar* can be be used to construct its *parser* by synthesizing a *push-down automaton (PDA)*[a].

[a]This is similar to the synthesis of a *scanner* from the regular expressions of the *token classes*.

Context-Free Grammar

A *context-free grammar (CFG)* G is defined by a *4-tuple* of data (Σ, N, P, S), where Σ is a finite set of *terminals*, N is a finite set of *non-terminals*. P is a finite subset of $N \times (\Sigma \cup N)^*$. Elements of P are called production or rewriting rules. The forth element S is a distinguished member of N, called the start symbol or the axiom of the grammar.

Derivation and Reduction

If $p = (A, \alpha) \in P$, we write it as $A \rightarrow \alpha$ ("A produces α" or "A can be replaced by α"). If $x = uAv \in (\Sigma \cup N)^*$, then we can *rewrite* x as $y = u\alpha v$ using the rule $p \in P$. Similarly, $y = u\alpha v$ can be *reduced* to $x = uAv$.

The first process is called derivation and the second process is called reduction.

Language of a Grammar

The language of a grammar G is denoted by $L(G)$. The language is a subset of Σ^*. An $x \in \Sigma^*$ is an element of $L(G)$, if starting from the *start symbol* S we can produce x by a finite sequence of rewriting[a]. The sequence of derivation of x may be written as $S \rightarrow x$[b].

[a] In other word x can be *reduced* to the start symbol S.

[b] In fact it is the *reflexive-transitive closure* of the single step derivation. We abuse the same notation.

Sentence and Sentential Form

Any $\alpha \in (N \cup \Sigma)^*$ derivable from the start symbol S is called a *sentential form* of the grammar. If $\alpha \in \Sigma^*$, i.e. $\alpha \in L(G)$, then α is called a *sentence* of the grammar.

Parse Tree

Given a grammar $G = (\Sigma, N, P, S)$, the parse tree of a sentential form x of the grammar is a rooted ordered tree with the following properties:

- The *root* of the tree is labeled by the *start symbol* S.
- The leaf nodes from left two right are labeled by the symbols of x.

Parse Tree

- Internal nodes are labeled by non-terminals so that if an internal node is labeled by $A \in N$ and its children from left to right are $A_1 A_2 \cdots A_n$, then $A \rightarrow A_1 A_2 \cdots A_n \in P$.
- A leaf node may be labeled by ε is there is a $A \rightarrow \varepsilon \in P$ and the parent of the leaf node has label A.

Example

Consider the following grammar for arithmetic expressions:

$G = (\{\texttt{id},\ \texttt{ic}, (,), +, -, *, /\}, \{E, T, F\}, P, E)$.

The set of *production rules*, P, are,

$$
\begin{aligned}
E &\rightarrow E + T \mid E - T \mid T \\
T &\rightarrow T * F \mid T/F \mid F \\
F &\rightarrow \texttt{id} \mid \texttt{ic} \mid (E)
\end{aligned}
$$

Example

Two derivations of the sentence $\texttt{id} + \texttt{ic} * \texttt{id}$ are,

d_1: $E \rightarrow E + T \rightarrow E + T * F \rightarrow E + F * F \rightarrow T + F * F \rightarrow F + F * F \rightarrow F + \texttt{ic} * F \rightarrow \texttt{id} + \texttt{ic} * F \rightarrow \texttt{id} + \texttt{ic} * \texttt{id}$

d_2:

$E \rightarrow E + T \rightarrow T + T \rightarrow F + T \rightarrow \texttt{id} + T \rightarrow \texttt{id} + T * F \rightarrow \texttt{id} + F * F \rightarrow \texttt{id} + \texttt{ic} * F \rightarrow \texttt{id} + \texttt{ic} * \texttt{id}$

It is clear that the derivations for a sentential form need not be unique.

Leftmost and Rightmost Derivations

A derivation is said to be *leftmost* if the leftmost nonterminal of a sentential form is rewritten to get the next sentential form. The *rightmost* derivation is similarly defined.

Due to the context-free nature of the production rules, any string that can be derived by unrestricted derivation can also be derived by leftmost(rightmost) derivation.

Ambiguous Grammar

A grammar G is said to be *ambiguous* if there is a sentence $x \in L(G)$ that has two distinct parse trees.

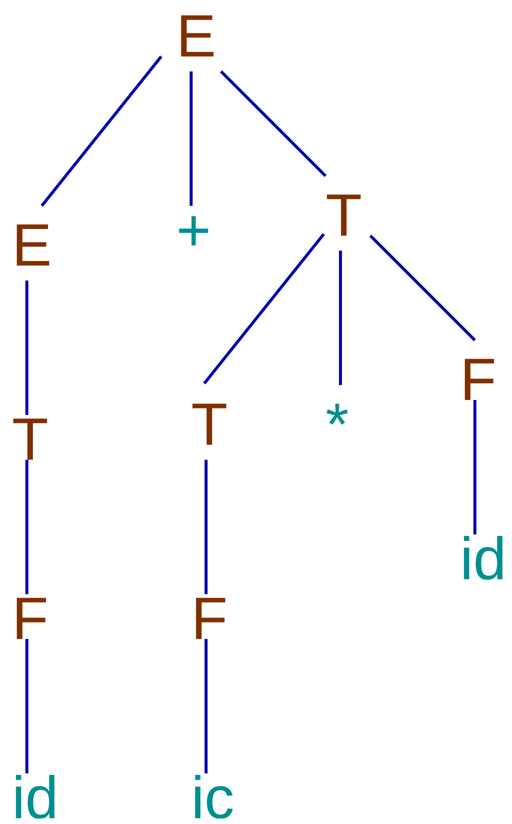
Example

Our previous grammar of arithmetic expressions is unambiguous. Following is an ambiguous grammar for the same language: $G' = (\{\mathtt{id},\ \mathtt{ic}, (,), +, -, *, /\}, \{E\}, P, E)$. The *production rules* are,

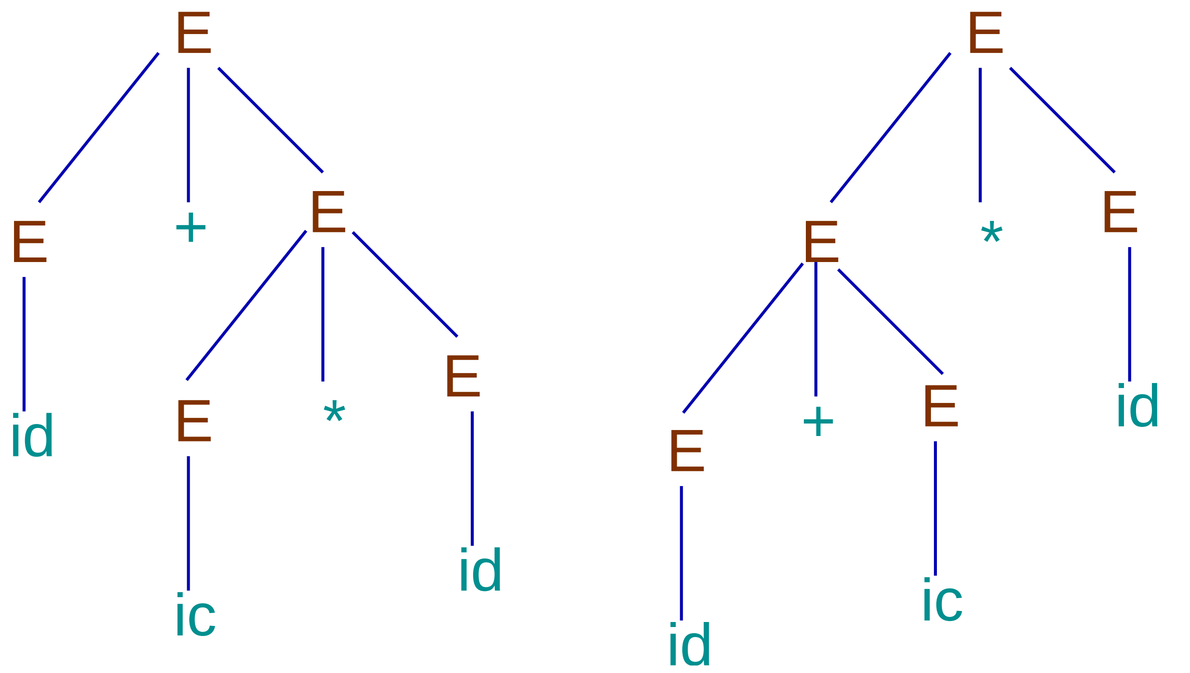
$$\begin{aligned} E \;&\rightarrow\; E + E \mid E - E \mid E * E \mid E/E \mid \\ &\quad\;\; \mathtt{id} \mid \mathtt{ic} \mid (E) \end{aligned}$$

Number of non-terminals may be less in an ambiguous grammar.

Unique Parse Tree



Non-Unique Parse Tree



Note

Leftmost(rightmost) derivation is unique for an unambiguous grammar but not in case of a ambiguous grammar.

d_3: $E \rightarrow E + E \rightarrow \texttt{id} + E \rightarrow \texttt{id} + E * E \rightarrow \texttt{id} + \texttt{ic} * E \rightarrow \texttt{id} + \texttt{ic} * \texttt{id}$

d_4: $E \rightarrow E * E \rightarrow E + E * E \rightarrow \texttt{id} + E * E \rightarrow \texttt{id} + \texttt{ic} * E \rightarrow \texttt{id} + \texttt{ic} * \texttt{id}$

The length of derivation of string with an ambiguous grammar may be shorter.

`if-else` Ambiguity

Consider the following production rules:

$$S \rightarrow \texttt{if}(E)S \mid \texttt{if}(E)\ S\ \texttt{else}\ S \mid \cdots$$

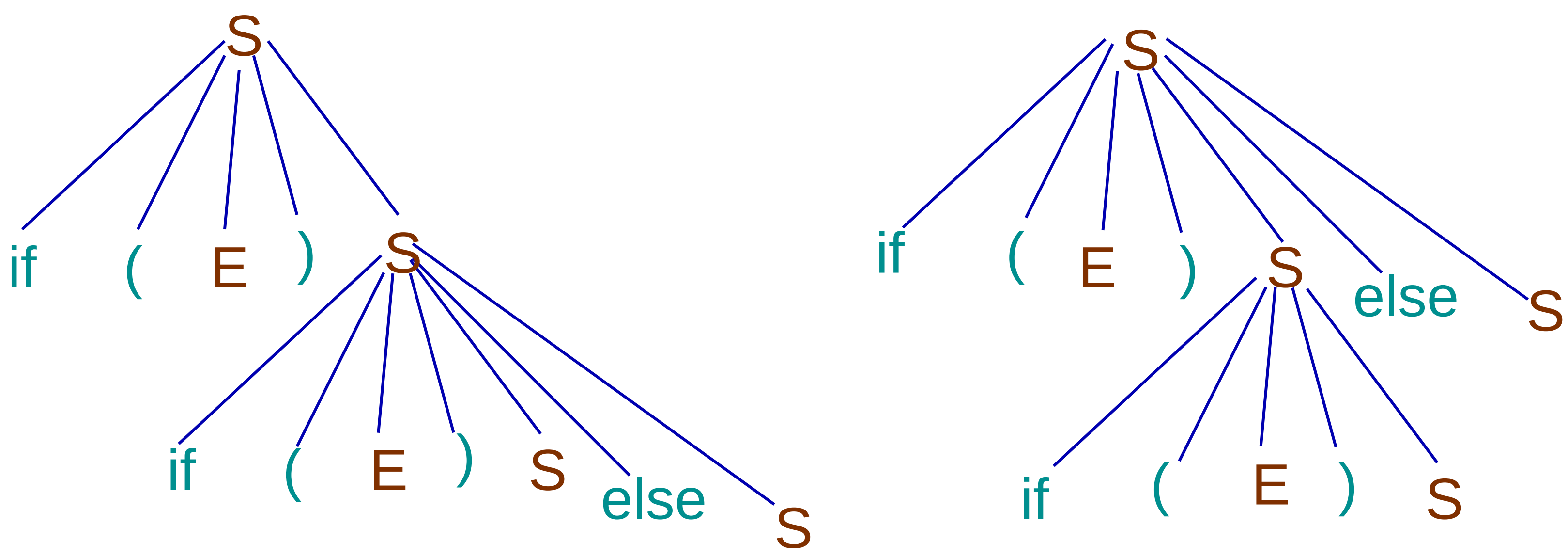
A statement of the form

`if(E1) if(E2) S2 else S3`

can be parsed in two different ways. Normally we associate the `else` to the nearest `if`[a].

[a] C compiler gives you a warning to disambiguate using curly braces.

`if-else` Ambiguity



if-else Modified

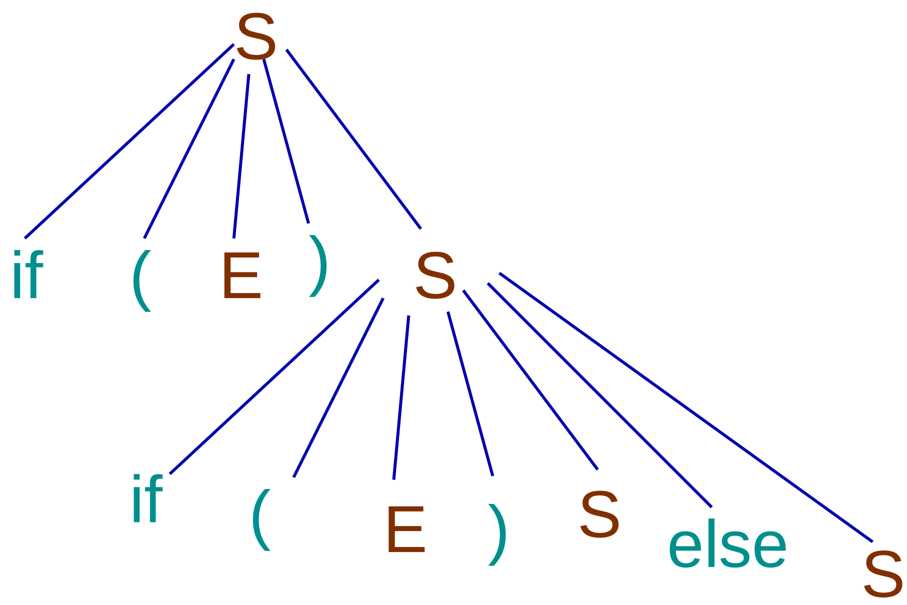
Consider the following production rules:

$$\begin{aligned} S &\rightarrow \texttt{if}(E)S \mid \texttt{if}(E)\ ES\ \texttt{else}\ S \mid \cdots \\ ES &\rightarrow \texttt{if}(E)\ ES\ \texttt{else}\ ES \mid \cdots \end{aligned}$$

We restrict the statement that can appeare in then-part. Now following statement has unique parse tree.

```
if(E1) if(E2) S2 else S3
```

if-else Unambiguous



Note

Consider the following grammar G_1 for arithmetic expressions:

$$
\begin{array}{rcl}
E & \rightarrow & T + E \mid T - E \mid T \\
T & \rightarrow & F * T \mid F/T \mid F \\
F & \rightarrow & \texttt{id} \mid \texttt{ic} \mid (E)
\end{array}
$$

Is $L(G) = L(G_1)$? Is there anything wrong with this grammar?

Problem

Consider another version of the grammar G_2:

$$
\begin{aligned}
E &\rightarrow E * T \mid E/T \mid T \\
T &\rightarrow T + F \mid T - F \mid F \\
F &\rightarrow \mathtt{id} \mid \mathtt{ic} \mid (E)
\end{aligned}
$$

What is different in this grammar? Is $L(G) = L(G_2)$.

Problem

Construct parse trees corresponding to the input `25-2-10` for G and G_1. What are the *postorder sequences* in these two cases (replace the non-terminals by ε)?
Similarly, construct parse trees corresponding to the input `5+2*10` for G and G_2. Find out the *postorder sequences* in these two cases?
Why postorder sequence?

Postorder Sequences

- G: 25 2 - 10 -

  G_1: 25 2 10 - -

- G: 5 2 10 * +

  G_2: 5 2 + 10 *

A Few Important Transformations

Useless Symbols

A grammar may have *useless symbols* that can be removed to produce a simpler grammar. A symbol is useless if it does not appear in any sentential form producing a sentence.

Useless Symbols

We first remove all non-terminals that does not produce any terminal string; then we remove all the symbols (terminal or non-terminal) that does not appear in any sentential form. These two steps are to be followed in the given order[a].

[a] As an example (HU), all useless symbols will not be removed if done in the reverse order on the grammar $S \to AB \mid a$ and $A \to a$.

ε-Production

If the language of the grammar does not have any ε, then we can free the grammar from ε-production rules. If ε is in the language, we can have only the start symbol with ε-production rule and the remaining grammar free of it.

Example

$$
\begin{aligned}
S &\rightarrow 0A0 \mid 1B1 \mid BB \\
A &\rightarrow C \\
B &\rightarrow S \mid A \\
C &\rightarrow S \mid \varepsilon
\end{aligned}
$$

All non-terminals are nullable.

Example

After removal of ε-productions.

$$
\begin{aligned}
S &\rightarrow 0A0 \mid 1B1 \mid BB \mid 00 \mid 11 \mid B \mid \varepsilon \\
A &\rightarrow C \\
B &\rightarrow S \mid A \\
C &\rightarrow S
\end{aligned}
$$

Unit Production

A production of the form $A \rightarrow B$ may be removed but not very important for compilation.

Normal Forms

A context-free grammar can be converted into different normal forms e.g. Chomsky normal form etc. These are useful for some decision procedure e.g. CKY algorithm. But are not of much importance for compilation.

Left and Right Recursion

A CFG is called left-recursive if there is a non-terminal A such that $A \rightarrow A\alpha$ after a finite number of steps. Left-recursion from a grammar is to be eliminated for a *top-down* parser[a].

[a]The right recursion can be similarly defined. It does not have so much problem as we do not read input from right to left, but in a *bottom-up* parser the stack size may be large due to right-recursion.

Immediate Left-Recursion

A left-recursion is *immediate* if a production rule of the form $A \rightarrow A\alpha$ is present in the grammar. It is easy to eliminate an *immediate left-recursion*. We certainly have production rules of the form

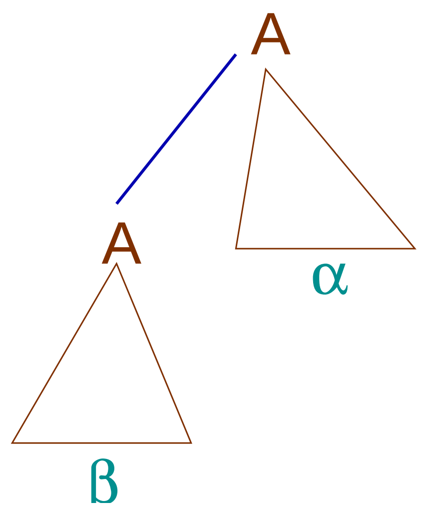
$$A \rightarrow A\alpha_1 \mid \beta$$

where the first symbol of β does not produce A[a].

[a] Otherwise A will be a useless symbol.

Parse Tree

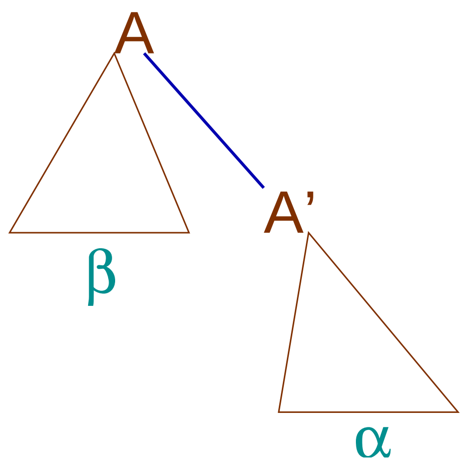
The parse tree with this pair of production rules looks as follows:



The yield is $\beta\alpha$.

Rotation

We can rotate the parse tree to get the same yield, but without the left-recursion.



The new rules are $A \to \beta A'$ and $A' \to \alpha A' \mid \varepsilon$.

Removal of Immediate Left-Recursion

The original grammar is

$$
\begin{aligned}
A &\rightarrow A\alpha_1 \mid A\alpha_k \mid \cdots \mid A\alpha_k \\
A &\rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_l
\end{aligned}
$$

The transformed grammar is

$$
\begin{aligned}
A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_l A' \\
A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_k A' \mid \varepsilon
\end{aligned}
$$

Example

Original grammar:

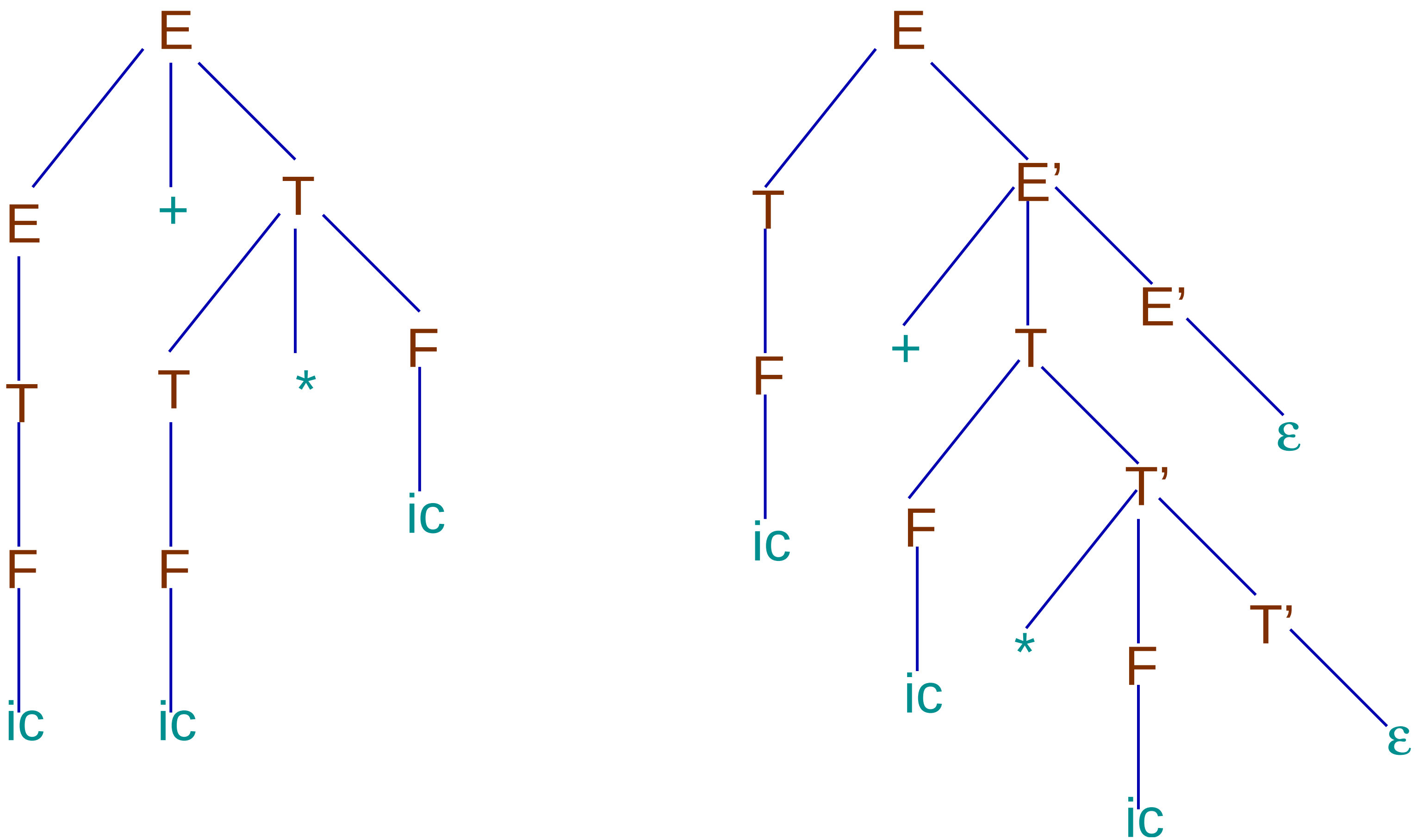
$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid ic
\end{aligned}
$$

The transformed grammar is

$$
\begin{array}{lclclcl}
E & \rightarrow & TE' & \quad & E' & \rightarrow & +TE' \mid \varepsilon \\
T & \rightarrow & FT' & \quad & T' & \rightarrow & *FT' \mid \varepsilon \\
F & \rightarrow & (E) \mid ic & & & &
\end{array}
$$

Change in the Parse Tree

Consider the input `ic+ic*ic`:



Removal of Indirect Left-Recursion

Consider the following grammar:

$$
\begin{aligned}
A &\rightarrow Aab \mid Ba \mid Cb \mid b \\
B &\rightarrow Aa \mid Db \\
C &\rightarrow Ab \mid Da \\
D &\rightarrow Bb \mid Ca
\end{aligned}
$$

The grammar has indirect left-recursion: $A \rightarrow Ba \rightarrow Aaa$ etc.

Removal of Indirect Left-Recursion

Following algorithm eliminates left-recursion.
First we order the non-terminals:
$A_1 < A_2 < \cdots < A_n$

Algorithm

for $i = 1$ **to** n

  for $j = 1$ **to** $i - 1$

    replace rule of the form $A_i \rightarrow A_j\gamma$
    by $A_i \rightarrow \delta_1\gamma \mid \cdots \mid \delta_k\gamma$, where
    $A_j \rightarrow \delta_1 \mid \cdots \mid \delta_k$ are the current
    A_j productions

  remove immediate left-recursion of
  A_i-productions.

Note

There is no left-recursion within the variables $A_1, A_2, \cdots, A_{i-1}$ and we are removing left-recursion from A_i.

Example

Let $A < B < C < D$. In the *first-pass* $(i = 1)$ of the outer loop, the immediate recursion of A is removed.

$$\begin{array}{rcl} A & \rightarrow & BaA' \mid CbA' \mid bA' \\ A' & \rightarrow & abA' \mid \varepsilon \\ B & \rightarrow & Aa \mid Db \\ \cdots & \cdots & \cdots \end{array}$$

Example

In the *second-pass* ($i = 2$) of the outer loop, $B \rightarrow Aa$ are replaced and immediate left-recursions on B are removed.

$$
\begin{array}{rcl}
A & \rightarrow & BaA' \mid CbA' \mid bA' \\
A' & \rightarrow & abA' \mid \varepsilon \\
B & \rightarrow & BaA'a \mid CbA'a \mid bA'a \mid Db \\
\cdots & \cdots & \cdots
\end{array}
$$

Example

$$
\begin{aligned}
A &\rightarrow BaA' \mid CbA' \mid bA' \\
A' &\rightarrow abA' \mid \varepsilon \\
B &\rightarrow DbB' \mid bA'aB' \mid CbA'aB' \\
B' &\rightarrow aA'aB' \mid \varepsilon \\
C &\rightarrow Ab \mid Da \\
\cdots & \quad \cdots \quad \cdots
\end{aligned}
$$

Example

In the *third-pass* ($i = 3$) of the outer loop,

$$
\begin{array}{rcl}
A & \rightarrow & BaA' \mid CbA' \mid bA' \\
A' & \rightarrow & abA' \mid \varepsilon \\
B & \rightarrow & DbB' \mid bA'aB' \mid CbA'aB' \\
B' & \rightarrow & aA'aB' \mid \varepsilon \\
C & \rightarrow & BaA'b \mid CbA'b \mid bA'b \mid Da \\
\cdots & \cdots & \cdots
\end{array}
$$

Example

$$
\begin{array}{rcl}
A & \rightarrow & BaA' \mid CbA' \mid bA' \\
A' & \rightarrow & abA' \mid \varepsilon \\
B & \rightarrow & DbB' \mid bA'aB' \mid CbA'aB' \\
B' & \rightarrow & aA'aB' \mid \varepsilon \\
C & \rightarrow & DbB'aA'b \mid bA'aB'aA'b \mid CbA'aB'aA'b \\
 & & CbA'b \mid bA'b \mid Da \\
\cdots & \cdots & \cdots
\end{array}
$$

Left Factoring

There may be more than one grammar rules for a non-terminal so that the right hand side of them have the same prefix. This creates a problem of rule selection for the non-terminal in some top-down parser. Such a grammar is transformed by left factoring to change the rules so that terminal prefixes of the right-hand sides of the productions of a non-terminal are unique.

Example

If we have production rules of the form $A \to xB\alpha$, $A \to xC\beta$, $A \to xD\gamma$, we transform them to $A \to xE$ and $E \to B\alpha \mid C\beta \mid D\gamma$, where $x \in \Sigma^*$.

Parsing

Using the grammar as a specification, a *parser* tries to construct the derivation sequence (reduction sequence or the parse tree) of a given input (a program to compile). This construction may be top-down or bottom-up.

Top-Down Parsing

A top-down parser starts from the start symbol (S) to generate the input string of tokens (x). Given a sentential form α the parser tries to determine a non-terminal A in α and one of its production rules $A \to \beta$, so that next sentential form γ can be derived satisfying

$S \to \alpha \xrightarrow{A \to \beta} \gamma \to x$.

Bottom-Up Parsing

A bottom-up parser starts from the input (x) and tries to reduce it to the start symbol (S). Given a sentential form α the parser tries to determine β, a substring of α, that matches with the right-hand side of a production $A \rightarrow \beta$, so that when β is replaced by A, the previous sentential form γ is obtained, satisfying $S \rightarrow \gamma \xrightarrow{A \rightarrow \beta} \alpha \rightarrow x$.

Note

We always read (consume) the input from left-to-right. In a top-down parser on the input x, the snapshot is as follows:
A part of the input u has already been generated/consumed i.e. $x = uv$ and the parser has the sentential form $uA\alpha$.

Note

Looking at the initial part of the remaining input v it is necessary for the parser to decide the correct production to get the next sentential form. If it always expands the left-most non-terminal, it is going by the leftmost derivation. But the choice of production rule may lead to dead-end or backtracking.

Example

Consider the following grammar:

$$S \;\rightarrow\; aSa \mid bSb \mid a \mid b$$

Given a *sentential form* aabaSabaa and the remaining portion of the input `ab`$\cdots$ it is impossible to decide by seeing one or two or any finite number of input symbols, whether to use the first or the third production rule to generate '`a`' of the input.

Example

Consider the following grammar:

$$S \rightarrow aSa \mid bSb \mid c$$

Given a *sentential form* aabaSabaa and the remaining portion of the input `abc`$\cdots$, it is clear from the first element of the input string that the first production rule is to be applied to get the next sentential form.

Note

In case of a bottom-up parser on the input x, the snapshot is as follows: The current sentential form is αv where the remaining portion of the input is v i.e. $x = uv$ and $\alpha \rightarrow u$. At this point the parser is to choose an appropriate portion of αv as the right-hand side β of some production $A \rightarrow \beta$ to reduce the current sentential form to the previous sentential form.

Note

There may be more than one such choices possible, and some of them may be incorrect. If β is always a *suffix* of α, then we are following a sequence of right-most derivation in reverse order (reductions).

Example

Consider the grammar:

$$E \rightarrow E + E \mid E * E \mid ic$$

Given the input `ic+ic*ic`$\cdots$, many reductions are possible and in this case all of them will finally lead to the start symbol. The previous sentential form can be any one of the following three, and there are many more: `E+ic*ic`$\cdots$, `ic+E*ic`$\cdots$, `ic+ic*E`$\cdots$ etc. The first one is the right sentential form.