Errors type

We know that programs can contain errors at many different levels. For example, errors can be

1. Lexical errors include misspellings of identifiers, keywords, or operators -e.g., missing quotes around text intended as a string.
2. Syntactic errors include misplaced semicolons or extra or missing braces; that is, '((" or ")."
3. Semantic errors include type mismatches between operators and operands.
4. logical, such as an infinitely recursive call

Syntax Analysis

Parser:

The parser has two functions:

1) It checks that the tokens appearing in its input, which is the output of the lexical analyzer, occur in patterns that are permitted by the specification for the source language.
2) It also imposes on the tokens a tree-like structure that is used by the subsequent phases of the compiler.

Example: if a Pascal program contains the following expression:

A + /B

Then after **lexical analysis** this expression might appear to the syntax analyzer as the token sequence

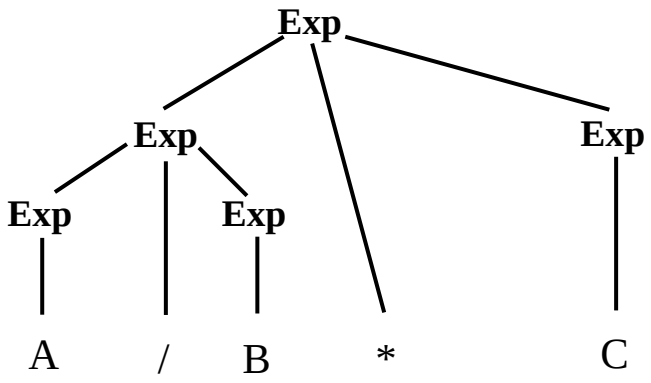
id1 + / id2

On seeing the /, the syntax analyzer should detect an **error** situation, because the presence of these two adjacent operators violates the formation rules of a Pascal expression.

Example: identifying which parts of the token stream should be grouped together:

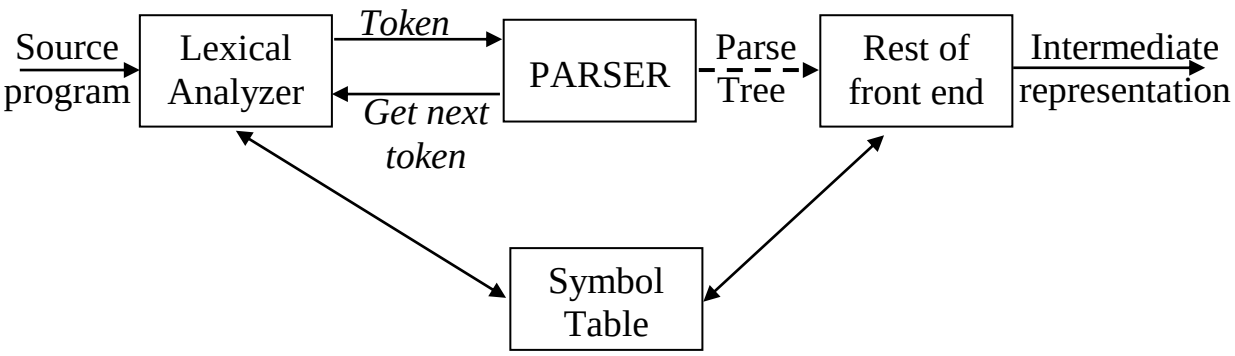
A/B*C

Parse Tree:



The Role of the Parser

In the compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in figure below, and verifies that the string can be generated by the grammar for the source language. The parser must be reported syntax errors clearly fashion.



Position of Parser in Compiler Model

By design, every programming language has precise rules that prescribe the syntactic structure of well-formed programs. In C, for example, a program is made up of functions, a function out of declarations and statements, a statement out of expressions, an expression out of tokens and so on. The syntax of programming language constructs can be specified by context-free grammars or

BNF (Backus-Naur Form) notation . Grammars offer significant benefits for both language designers and compiler writers

Context-Free Grammars (CFG)

Many programming language constructs have an inherently recursive structure that can be defined by context-free grammars. For example, we might have a conditional statement defined by a rule such as

If $\mathbf{S_1}$ and $\boldsymbol{S_2}$ are statements and E is an expression, then

"If E **then** S_1 **else** S_2" is a statement.

This form of conditional statement cannot be specified using the notation regular expressions.

Could also express as: ***stmt*** ⟶ **if *expr* then *stmt* else *stmt***

Such as a role is called syntactic variables, ***stmt*** to denote the class of statements and ***expr*** the class of expressions.

Components of Context-Free Grammars (CFG)

A context free grammar (CFG for short) consists of terminals, nonterminals, a start symbol, and productions.

1) **Terminals** are the basic symbols from which strings are formed. The word "***token***" is a synonym for "**terminal**" when we are talking about grammars for programming languages.
2) **Nonterminals** are syntactic variables that denote sets of strings.
3) One nonterminal is distinguished as the **Start Symbol**.
4) The set of **Productions** where each production consists of a nonterminal, called the left side followed by an arrow, followed

by a string of nonterminals and/or terminals called the right side.

Example: The grammar with the following productions defines simple arithmetic expressions.

expr ⟶ ***expr op expr***
expr ⟶ **(*expr*)**
expr ⟶ **– *expr***
expr ⟶ **id**
op ⟶ **+**
op ⟶ **–**
op ⟶ *****
op ⟶ **/**
op ⟶ **↑**

In this grammar, the terminal symbols are

id + - * / ↑ ( )

The nonterminal symbols are ***expr*** and ***op,*** and ***expr*** is the start symbol.

The above grammar can be rewriting by using **shorthands** as:

E ⟶ ***EAE*** | **(*E*)** | **-*E*** | **id**
A ⟶ **+** | **-** | ***** | **/** | **↑**

where ***E*** and ***A*** are nonterminals, with ***E*** the start symbol. The remaining symbols are terminals.

Derivations and Parse Trees

How does a context-free grammar define a language? The central idea is that productions may be applied repeatedly to expand the nonterminals in a string of nonterminals and terminals. For example, consider the following grammar for arithmetic expressions:

E ⟶ ***E* + *E* \ *E* * *E*** | **(*E*)** \ **-*E*** \ **id** …… (**1.1**).

The nonterminal ***E*** is an abbreviation for expression. The production ***E*** ⟶ **-*E*** signifies that an expression preceded by minus sign is also an expression. In the simplest case can replace single ***E*** by **-*E***. We can describe this action by writing

$E \longrightarrow -E$ which is read as "***E*** derives ***-E***"

We can take a single ***E*** and repeatedly apply productions in any order to obtain a sequence of replacements. For example,

$E \longrightarrow -E \longrightarrow -(E) \longrightarrow -(\mathbf{id})$

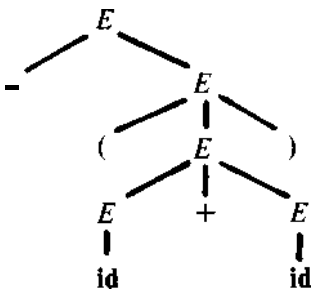
We call such a sequence of replacements a **derivation** of **- (id)** from ***E***. This derivation provides a proof that one particular instance of an expression is the string **- (id)**.

Example: The string **-(id + id)** is a sentence of grammar (1.1) because there is the derivation

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E)$$
$$\Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

Parse Tree may be viewed as a graphical representation for derivations that filters out the choice regarding replacement order. Each **interior node** of the parse tree is labeled by some **nonterminal *A***, and the children of the node are labeled, from left to right, by the symbols in the right side of the production by which this ***A*** was replaced in the derivation.

The **leaves** of the parse tree are labeled by **nonterminals** or **terminals** and, read from left to right. For example, the parse tree for **-(id+id)** that implied by the derivation of previous example.

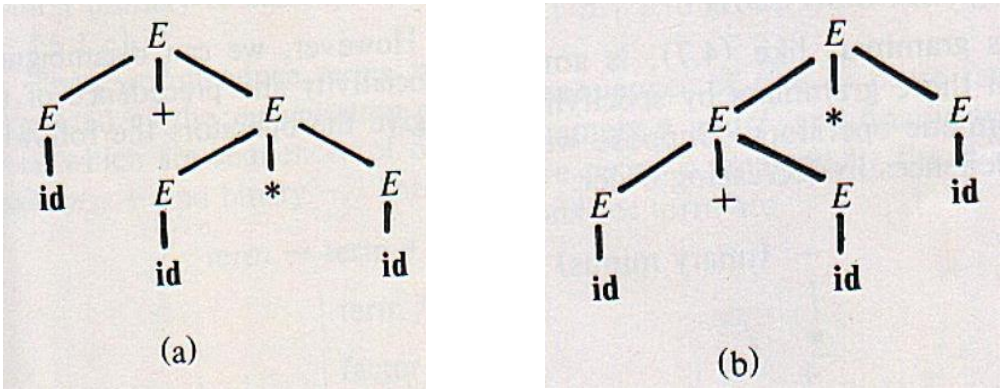


Example: Let us again consider the arithmetic expression grammar (1.1), with which we have been dealing. The sentence **id + id * id** has the two distinct leftmost derivations:

$$E \longrightarrow E + E \longrightarrow \mathbf{id} + E \longrightarrow \mathbf{id} + E * E \longrightarrow \mathbf{id} + \mathbf{id} * E \longrightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$$

$$E \longrightarrow E * E \longrightarrow \mathbf{E} + E * E \longrightarrow \mathbf{id} + E * E \longrightarrow \mathbf{id} + \mathbf{id} * E \longrightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$$

With the two corresponding parse tree shown in figure below:



Two parse trees for **id + id * id**

Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be ambiguous. Put another way, an ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for some sentence. In this type, we cannot uniquely determine which parse tree to select for a sentence.

Example: Consider the following grammar for arithmetic expressions involving **+, -, *, /,** and↑ (exponentiation)

$E \longrightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\uparrow E \mid (E) \mid -E \mid \mathbf{id}$

This grammar is ambiguous.

Eliminate Ambiguity

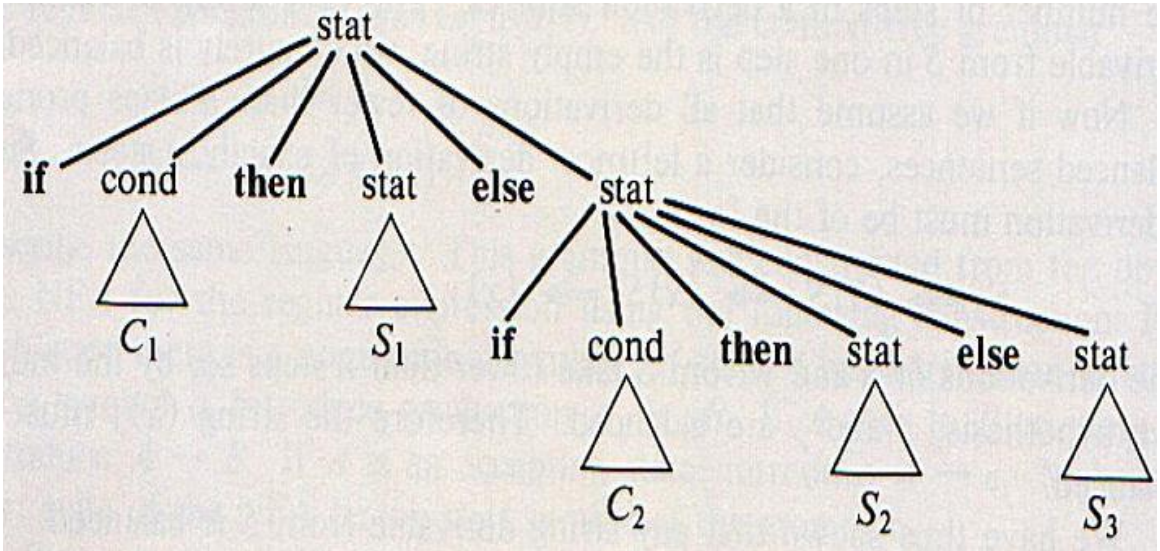
Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity and transform the ambiguous grammar to unambiguous grammar. As an example, we shall eliminate the ambiguity from the following "**dangling-else**" grammar:

Stat ⟶ **if *cond* then *stat***
| **if *cond* then *stat* else *stat***
| ***other-stat***

Here "***other-stat***" stands for any other statement. According to this grammar, the compound conditional statement

if C_1 then S_1 else if C_2 then S_2 else S_3

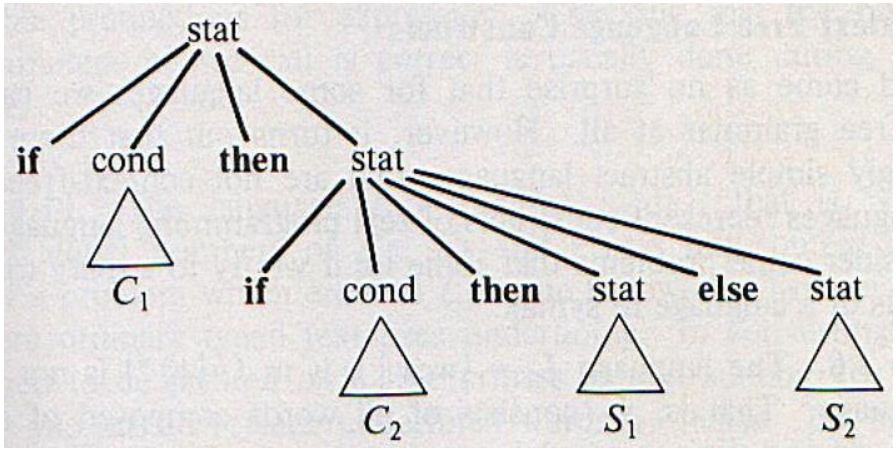
Has the parse tree like the following figure:

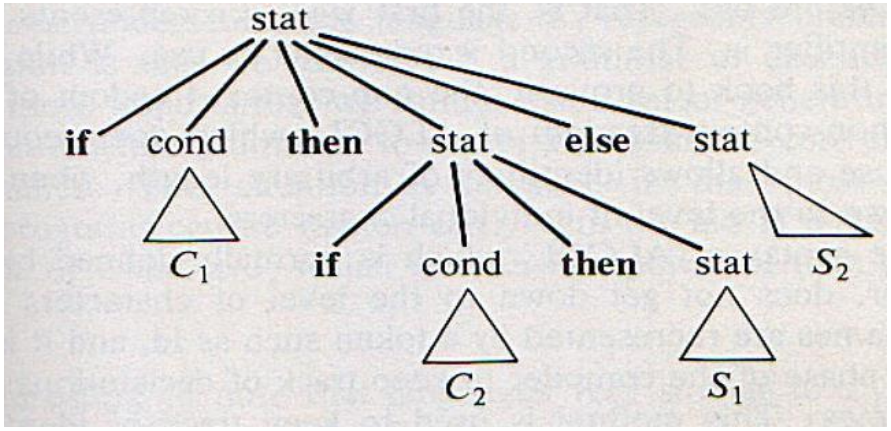


This grammar is ambiguous since the string

if C_1 then if C_2 then S_1 else S_2

Has the two parse tree shown in the figure below:





The above ambiguous grammar transform to the unambiguous grammar:

stat ⟶ ***matched-stat***

| ***unmatched-stat***

matched-stat ⟶ **if *cond* then *matched-stat* else *matched-stat***

| ***other-stat***

unmatched-stat ⟶ **if *cond* then *stat***

| **if *cond* then *matched-stat* else *unmatched-stat***

This grammar generates the same set of string for ambiguous grammar, but it allows only one parsing for above string.

Regular Expression vs. Context-Free Grammar

Every language that can be described by a regular expression can also be described by Context-Free Grammar.

Example: Design CFG that accept the RE = $a\ (a|b)^*b$

$S \longrightarrow aAb$

$A \longrightarrow aA \mid bA \mid \varepsilon$

Example: Design CFG that accept the RE = $(a|b)^*abb$

$S \longrightarrow aS \mid bS \mid aX$

$X \longrightarrow bY$

$Y \longrightarrow bZ$

$Z \longrightarrow \varepsilon$

Example: Design CFG that accept the RE =a^nb^n where $n \geq 1$.

$S \longrightarrow aXb$

$X \longrightarrow aXb \mid \varepsilon$

Example: Design CFG ***Singed Integer*** number.

$S \longrightarrow XD$

$X \longrightarrow + \mid -$

$D \longrightarrow 0D \mid 1D \mid 2D \mid 3D \mid 4D \mid 5D \mid 6D \mid 7D \mid 8D \mid 9D \mid \varepsilon$

Elimination of Left Recursion

A grammar is ***Left Recursive*** if it has a nonterminal A such that there is a derivation $A \xrightarrow{+} A\alpha$ for some string α. **Top-Down**

Parsing methods cannot handle left recursive grammars, so a transformation that eliminates left recursion is needed.

In the following example, we show how the left recursion pair of productions $A \longrightarrow A\alpha \mid \beta$ could be replaced by the non-left-recursive productions:

$$A \longrightarrow A\alpha \mid \beta$$

$$A \longrightarrow \beta A'$$

$$A' \longrightarrow \alpha A' \mid \varepsilon$$

without changing the set of strings derivable from A. This rule by itself suffices in many grammars.

Example: Consider the following grammar for arithmetic expressions.

$$E \longrightarrow E+T \mid T$$

$$T \longrightarrow T*F \mid F$$

$$F \longrightarrow (E) \mid \text{id}$$

Eliminating the immediate ***left recursion*** (productions of the form $A \longrightarrow A\alpha$) to the productions for E and then for T, we obtain

$$E \longrightarrow TE'$$

$$E' \longrightarrow +TE' \mid \varepsilon$$

$$T \longrightarrow FT'$$

$$T' \longrightarrow *FT' \mid \varepsilon$$

$$F \longrightarrow (E) \mid \text{id}$$

Note: No matter how many A-productions there are, we can eliminate immediate left recursion from them by the following technique. First, we group the A-productions as

$$A \longrightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots\ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots\ldots \mid \beta_n$$

where no β, begins with an A. Then, we replace the A-productions by:

$$A \longrightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots\ldots \mid \beta_n A'$$
$$A' \longrightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots\ldots \mid \alpha_m A' \mid \varepsilon$$

The nonterminal A generates the same strings as before but is no longer left recursive.

Note: This procedure eliminates all immediate left recursion from the A and A' productions, but it does not eliminate left recursion involving derivations of **two or more** steps.

For **Example**, consider the grammar:

$S \longrightarrow Aa \mid b$

$A \longrightarrow Ac \mid Sd \mid \varepsilon$

The nonterminal S is left- recursive because $S \longrightarrow Aa \longrightarrow Sda$, but is not immediately left recursive. **Algorithm** in below will systematically eliminate left recursion from grammar.

Algorithm: Eliminating left recursion.

Input: Grammar G with no cycles or ε -productions.

Output: An equivalent grammar with no left recursion.

Method: Apply the algorithm below to G. Note that the resulting non left-recursive grammar may have ε -productions.

Arrange the nonterminals in some order $A_1, A_2, \ldots\ldots A_n$.

for i: = 1 **to** n **do**

  for j : = 1 **to** i-1 **do begin**

    Replace each production of the form $A_i \longrightarrow A_i\gamma$ by the productions $A_i \longrightarrow \delta_1\gamma \mid \delta_2\gamma \mid \delta_3\gamma \mid \ldots\ldots \mid \delta_k\gamma$,

Where $A_j \longrightarrow \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots\ldots \mid \delta_k$ are all the current A_j-productions;
Eliminate the immediate left recursion among A_j-productions
End

Let us apply this procedure to previous grammar. We substitute the ***S***-productions in $A \longrightarrow Sd$ to obtain the following ***A***-productions.

$A \longrightarrow Ac \mid Aad \mid bd \mid \varepsilon$

Eliminating the immediate left recursion among the ***A***-productions yields the following grammar.

$S \longrightarrow Aa \mid b$

$A \longrightarrow bdA' \mid A'$

$A' \longrightarrow c\,A' \mid adA' \mid \varepsilon$

Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for **predictive parsing**. The basic idea is that when it is not clear which of **two** alternative productions to use to expand a nonterminal ***A***, we may be able to rewrite the, ***A***-productions to defer the decision until we have seen enough of the input to make the right choice. For example, if we have the two productions

if $A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two ***A***-productions, and the input begins with a nonempty string derived from α, we do not know whether to expand ***A*** to $\alpha\beta_1$or to $\alpha\beta_2$. However, we may defer the decision by expanding ***A*** to $\alpha A'$. Then, after seeing the input derived from α, we expand A' to β_1 or to β_2. That is left-factored.

$$A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2 \quad \Longrightarrow \quad \begin{array}{l} A \longrightarrow \alpha A' \\ A' \longrightarrow \beta_1 \mid \beta_2 \end{array}$$

Algorithm : Left factoring a grammar.

Input: Grammar ***G***.

Output: An equivalent left-factored grammar.

Method: For each nonterminal ***A*** find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \varepsilon$, i.e., there is a nontrivial common prefix, replace all the ***A*** productions $A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots\ldots \mid \alpha\beta_2 \mid \gamma$ where γ represents all alternatives that do not begin with α by

$$A \longrightarrow \alpha A' \mid \gamma$$

$$A' \longrightarrow \beta_1 \mid \beta_2 \mid \ldots\ldots\ldots \mid \beta_n$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

Example:

$$S \longrightarrow iEtS \mid iEtSeS \mid a$$
$$E \longrightarrow b$$

Left-factored, this grammar becomes:

$$S \longrightarrow iEtSS' \mid a$$
$$S' \longrightarrow eS \mid \varepsilon$$
$$E \longrightarrow b$$

Example:

$$A \longrightarrow aA \mid bB \mid ab \mid a \mid bA$$

Solution:

$$A \longrightarrow aA' \mid bB'$$
$$A' \longrightarrow A \mid b \mid \varepsilon$$
$$B' \longrightarrow B \mid A$$