

Syntax Analyzer (Parser)

We show that the lexical structure of tokens could be specified by regular expressions and that from a regular expressions we could automatically construct a lexical analyzer to recognize the tokens denoted by the expression. We shall use a notation called a **context-free grammar** or grammar only, for the syntactic specification of a programming language which is also called a BNF (Backus-Naur Form) description.

The parser obtains a string of tokens from the lexical analyzer, and verifies that the string can be generated by the grammar for the source language. The parser reports any syntax errors and gives an indication of the type of these errors. Fig.1 shows the position of parser.

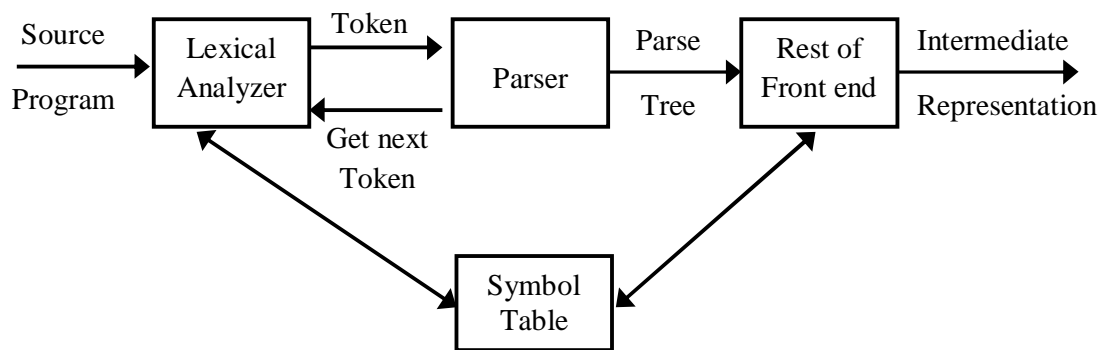


Fig.1: position of parser in compiler model

Context Free Grammars (CFG's)

It is natural to define certain programming-language construct recursively. For example, we might state:-

If S_1 and S_2 are statements and E is an expression, then:-

"If E then S_1 else S_2 " is a statement ----- (1)

Or: If S_1, S_2, \dots, S_n are statements then

"begin $S_1; S_2; \dots; S_n$ end" is a statement ----- (2)

As a third example:

If E_1 and E_2 are expressions, then " $E_1 + E_2$ " is an expression ----- (3)

If we use the syntactic category "statement" to denote the class of the statements and "expression" to denote the class of expressions, then (1) can be expressed by this production:

Statement \longrightarrow if expression then statement else statement ----- (4)

And (3) can be written as:-

Expression \longrightarrow expression + expression ----- (5)

If we want to write (2) in the same way we may have:

Statement \longrightarrow begin statement; statement; ... ; statement end

But the use of ellipses (...) would create problems when we attempt to define translations based on this description. For this reason, we require that each rewriting rule (production) have a known number of symbols, with no ellipses permitted.

To express (2) by rewriting rules, we can introduce a new syntactic category "statement-list" denoting any sequence of statements separated by semicolons. Then we can write:-

Statement \longrightarrow begin statement-list end

Statement-list \longrightarrow statement | statement; statement-list ----- (6)

A set of rules such that (6) is an example of a grammar. In general, a grammar involves four quantities: - start symbol, terminals, non-terminals, productions (rewriting rules).

EX: consider the following grammar for simple arithmetic expressions. The non-terminal symbols are **expressions** and **operator**, with **expression** the start symbol. The terminal symbols are: **id**, **+**, **-**, *****, **/**, **(**, **)**, **↑**

The productions are

Expression \longrightarrow Expression operator Expression

Expression \longrightarrow (Expression)

Expression \longrightarrow - Expression

Expression \longrightarrow id

Operator \longrightarrow +

Operator \longrightarrow -

Operator \longrightarrow *

Operator \longrightarrow /

Operator \longrightarrow ↑

We can write these productions in this form:-

$$E \longrightarrow EAE \mid (E) \mid -E \mid id$$

$$A \longrightarrow + \mid - \mid / \mid \uparrow \mid *$$

Derivation and Parse Trees

The central idea of how context free grammar defines a language is that the productions may be applied repeatedly to expend the non-terminals in a string of non-terminals and terminals. For example, consider the following grammar for arithmetic expressions,

$$E \longrightarrow E+E \mid E * E \mid (E) \mid -E \mid id \text{ ----- (7)}$$

We say that $\alpha AB \Rightarrow \alpha y \beta$ if $A \rightarrow y$ is a production and α and β are arbitrary strings of grammars symbols. If $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$, we say α_1 derives α_n . The symbol \Rightarrow means "derives in one step". Often we wish to say "derives in zero or more steps". For this purpose we can use the symbol \Rightarrow^* , and the symbol \Rightarrow^+ for "derives in one or more steps". Given a CFG G with a start symbol S , we can use the \Rightarrow^+ relation to define $L(G)$, the language generated by G . Strings in $L(G)$ may contain only terminal symbols of G . We say a string of terminals W is in $L(G)$ if and only if $S \Rightarrow^+ W$. The string W is called a sentence of G . if $S \Rightarrow^* \alpha$, where α may contain non-terminals, then we say α is sentential form of G .

EX: The string $-(id+id)$ is a sentence of grammar (7) because:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$$

There is often a degree of arbitrariness in the choice of replacement mode in a derivation. Of course, we may choose any alternate for non-terminal being replaced. But in addition, at any step of the derivation we may choose which non-terminal we wish to replace. For example, the derivation of the previous example could contain from $-(E+E)$ as follows

$$-(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id) \text{ ----- (8)}$$

To understand how certain parsers work we need to consider derivations in which only the left most non-terminal in a sentential form is replaced at each step. Such derivations are called **left most**. The derivation in the previous example is left most derivation. Analogously, **right most**

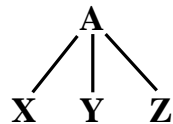
derivations are those in which the right most non-terminal is replaced at each step.

Parse Trees

We can create a 'graphical representation' for derivations that filter out the choice regarding replacement order. This representation is called the parse tree, and it has the important purpose of making explicit the hierarchical syntactic structure of sentences that is implied by the grammar.

Each interior node of the parse tree is labeled by some non-terminal A , and the children of 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.

For example if $A \longrightarrow XYZ$ is a production used at some step of a derivation, then the parse tree for that derivation will have the sub tree:-



The leaves of the parse tree are labeled by non-terminals or terminals and read from left to right; they constitute a sentential form called the yield or frontier of the tree. For example the parse tree for $-(id+id)$ implied by the derivation of the previous example is shown in Fig.2.

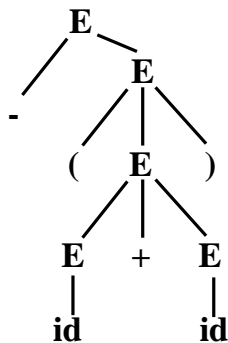


Fig.2: Parse tree

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$$

EX: let us again consider the arithmetic expression grammar (7), with which we have been dealing. The sentence $\text{id}+\text{id}*\text{id}$ has the two distinct left most derivations:-

$E \Rightarrow E+E$	$E \Rightarrow E * E$
$\Rightarrow \text{id}+E$	$\Rightarrow E+E * E$
$\Rightarrow \text{id}+E * E$	$\Rightarrow \text{id}+E * E$
$\Rightarrow \text{id}+\text{id} * E$	$\Rightarrow \text{id}+\text{id} * E$
$\Rightarrow \text{id}+\text{id} * \text{id}$	$\Rightarrow \text{id}+\text{id} * \text{id}$

(a)

The two parse tree are shown in Fig.3

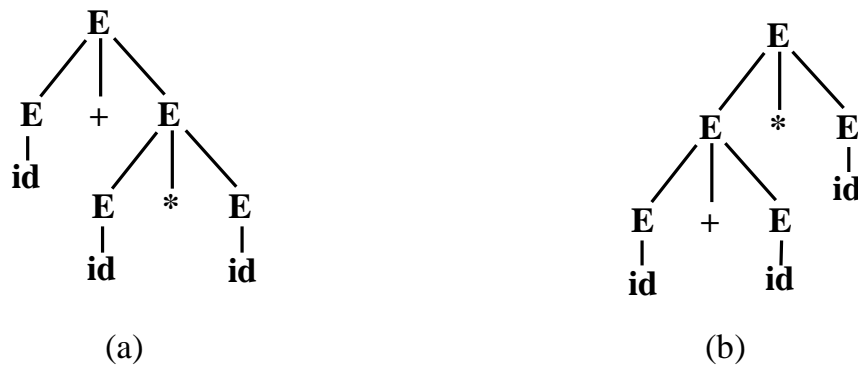


Fig.3: Two parse trees for $\text{id}+\text{id}*\text{id}$

Note that, in same sense, the parse tree of Fig.3 (a) is "**correct**" in that it reflects the commonly assumed precedence of $+$ and $*$, while the tree of Fig.3(b) does not. It is customary to treat operator $*$ as having higher precedence than $+$, so Fig.3 (a), which groups the operands of $*$ before those of $+$, corresponds to the structure we would normally attribute to an expression like $a+b*c$.

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 produced more than **one left most** or more than **one right most derivation** for some sentence. For certain types of parsers, it is desirable that the grammar be made unambiguous, for if it is not, we cannot uniquely determine which parse tree to select for a sentence.

Example Consider the following grammar for arithmetic expression involving +, -, *, / and \uparrow (**exponentiation**)

$$E \longrightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid -E \mid \text{id} \quad \text{----- (9)}$$

This grammar like (7) is ambiguous. However, we can disambiguate both these grammars by specifying the **associativity** and **precedence** of the arithmetic operators.

Suppose we wish to give the operators the following precedence **in decreasing order:-**

- (unary minus)

\uparrow

* /

+ -

Suppose further we wish \uparrow to be right-associative [e.g. $a \uparrow b \uparrow c$ is to mean $a \uparrow (b \uparrow c)$] and the other binary operators to be left associative [e.g. $a - b - c$ is to mean $(a - b) - c$].

We can also rewrite a grammar to incorporate the associativity and precedence rules into the grammar itself. Let us transform (9) into an equivalent unambiguous grammar that obeys the associativity and precedence rules given above.

We begin by introducing **one non-terminal for each precedence level**. An '**element**' is either a single identifier or a parenthesized expression. We therefore have the productions:-

$$\text{element} \longrightarrow (\text{expression}) \mid \text{id}$$

Next we introduce the category of '**Primaries**', which are elements with zero or more of the operator of highest precedence, the unary minus. The rule for primary is:-

$$\text{Primary} \longrightarrow - \text{Primary} \mid \text{element}$$

Then we construct '**factors**' as a sequence of one or more primaries connected by exponentiation signs. That is:-

$$\text{factor} \longrightarrow \text{primary} \uparrow \text{factor} \mid \text{primary}$$

Note that the choice of the right side '**primary** \uparrow **factor**' rather than '**factor** \uparrow **primary**' forces expressions like $a \uparrow b \uparrow c$ to group from the right as $a \uparrow (b \uparrow c)$.

Then we introduce '**term**', which are sequences of one or more **factors** connected by multiplicative operators namely '*' and '/', and finally **expressions**, which are sequences of one or more **terms** connected by the additive operators, '+' and binary '-'. The production for term is:-

$$\text{term} \longrightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor}.$$

These productions cause **terms** to be grouped from the left [e.g. $a*b*c$ means $(a*b)*c$]. Then the final, **unambiguous grammar** is:-

$$\begin{aligned} \text{Expression} \longrightarrow & \text{expressions} + \text{term} \\ & \mid \text{expression} - \text{term} \\ & \mid \text{term} \end{aligned}$$

$$\begin{aligned} \text{term} \longrightarrow & \text{term} * \text{factor} \\ & \mid \text{term} / \text{factor} \\ & \mid \text{factor} \end{aligned}$$

$$\text{factor} \longrightarrow \text{primary} \uparrow \text{factor} \mid \text{primary}$$

$$\text{primary} \longrightarrow - \text{primary} \mid \text{element}$$

$$\text{element} \longrightarrow (\text{expression}) \mid \text{id}$$

Basic Parsing Techniques

We show previously that the CFG can be used to define the syntax of a programming language, but now we show how to check whether an input string is a sentence of a given grammar and how to construct a parse tree for this string. The input to the parser is typically a sentence of tokens. The output of the parser can be of many different forms, and for simplicity we assume that the parser is some representation of the parse tree.

The most common forms of parsers are 'operator precedence' and 'recursive descent'. Operator precedence is especially suitable for parsing expressions, since it can use information about the precedence and associativity of operators to guide the parse. 'Recursive descent' uses a collection of mutually recursive routines to perform the syntax analysis. A common situation is for operator precedence to be used for expressions and recursive descent for the test of the language.

There are two newer methods gaining popularity that are both more general than the older methods and more firmly grounded in grammar theory. The first of these methods, LL parsing and the second is LR parsing.

Parsers

A parser for a grammar G is a program that takes as input a string W and produces as output either a parse tree for W , if W is a sentence of G , or an error message indicating that W is not sentence of G .

We discuss the operations of two types of parsers for CFG's:- bottom-up and top-down parsers build, parse tree from the bottom (leaves) to the top (root), while top-down parsers start with the root and work down to the leaves. In both cases the input to the parser is being scanned from the left to right, one symbol at a time.

The bottom-up parsing method is called "Shift-reduce" parsing, because it appears on top of the stack. One type of shift-reduce is "operator precedence parser". The top-down parsing method is called "recursive descent" parsing.

Representation of a Parse Tree

There are two basic types of representations: - **implicit** and **explicit**. The sequence of productions used in some derivation is an example of an implicit representation. A linked list structure for the parse tree is an

explicit representation. Recall that a derivation in which the left most non-terminal is replaced at every step is said to be left most. If $\alpha \Rightarrow \beta$ by a step in which the left-most non-terminal is α is replaced, we write $\alpha \xRightarrow{\text{lm}} \beta$. Every left most step, using our notational conventions, has the form $wAy \xRightarrow{\text{lm}} w?y$ in which w consists of terminals only. If α derives β

by a left most derivation we write $\alpha \xRightarrow[\text{lm}]{*} \beta$.

If $S \xRightarrow[\text{lm}]{*} \alpha$, then we say α is a left-sentential form of the grammar. Right most derivations are sometimes called "canonical derivations".

Every sentence of a language has both a left most and right most derivation. To find one left most derivation for sentence W , we can take any derivation for W and construct from it the corresponding parse tree T , from T we can then construct a left most derivation by traversing the tree top-down. We begin with the start symbol S , which corresponds to the root of T . we then construct the left most derivation:-

$$S=\alpha_1 \xRightarrow[\text{lm}]{} \alpha_2 \xRightarrow[\text{lm}]{} \dots \xRightarrow[\text{lm}]{} \alpha_n = W.$$

This derivation is constructed corresponding to T , one step at a time, using the following procedures:-

If the root labeled S has children labeled A , B and C , we create the first step of the left most derivation by replacing S by the labels of its children; i.e.

$S \xRightarrow[\text{lm}]{} ABC$. Here S is α_1 and ABC is α_2 .

If the node for A has children labeled XYZ in T , we create the next step of the derivation by replacing A by the labels of its children, i.e.

$ABC \xRightarrow[\text{lm}]{} XYZBC$. Here $XYZBC$ is α_3 . We continue in this fashion by finding the node corresponding to the left most non-terminal D of α_i and replacing D by its children in T to obtain α_{i+1} , for each $i=1, 2, \dots n-1$.

If we want to construct a parse tree in preorder, then the order in which the nodes are created corresponds to the order in which the productions are applied in a left derivation.

EX:-Consider the grammar:-

- (1) $S \longrightarrow i C t S$
- (2) $S \longrightarrow i C t S e S$ ----- (1)
- (3) $S \longrightarrow a$
- (4) $C \longrightarrow b$

Here i, t and e stand for "if", "then" and "else" respectively and C for "conditional", S for "statement".

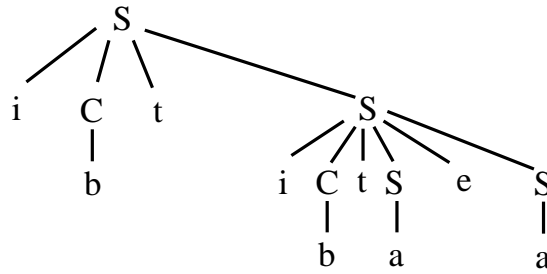


Fig.4: Parse Tree T

The left-most derivation corresponding to this parse tree is given by:-

$$\begin{aligned}
 S &\xRightarrow{lm} i C t S \\
 &\xRightarrow{lm} i b t S \\
 &\xRightarrow{lm} i b t i C t S e S \\
 &\xRightarrow{lm} i b t i b t S e S \\
 &\xRightarrow{lm} i b t i b t a e S \\
 &\xRightarrow{lm} i b t i b t a e a
 \end{aligned}$$

A right most derivation can be constructed from a parse tree analogously. At each step we replace the right-most non-terminal by the tables of its children. For example, the first two steps of a right most derivation constructed from Fig.4 would be

$$S \Rightarrow i C t S \Rightarrow i C t i C t S e S$$