**Syntax Analyzer (Parser)**

**Syntax Analysis**

In our compiler model, 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 program. We expect the parser to report any syntax errors in an intelligible fashion. It should also recover from commonly occurring errors so that it can continue processing the remainder of its input.



Fig(8) Position of parser in Compiler model

The methods commonly used in compilers are classified as being either Top-down or bottom -up. As indicated by their names, Top- down parsers build parse trees from the top (root) to the bottom (leaves) and work up to the root.

In both cases, the input to the parser is scanned from left to right, one

symbol at time. We assume the output of the parser is some representation of the parse tree for the stream of tokens produced by the lexical analyzer. In practice there are a number of tasks that might be conducted during parsing, such as collecting information about various tokens into the symbol table, performing type checking and other kinds of semantic analysis, and generating intermediate code.

**Grammars and productions**

For the syntactic specification of a programming language , we shall use a notation called a context free grammar which systematically describe the syntax of programming language constructs like expressions and statements. Using a syntactic variable *stmt* to denote statements and variable *expr* to denote expressions, the production

*stmt*  → **if** ( *expr* ) *stmt* **else** *stmt*

specifies the structure of this form of conditional statement. Other productions then define precisely what an *expr* is and what else a *stmt* can be.

**Defintion:**

A grammar *G* is a quadruple { *N, T, S, P* } with the four components

(a) *N* - a finite set of **non-terminal** symbols,

(b) *T* - a finite set of **terminal** symbols,

(c) *S* - a special **goal** or **start** or **distinguished** symbol,

(d) *P* - a finite set of **production rules** or, simply, **productions**

the productions have the form

leftside→ definition

Here "→ " can be interpreted as "is defined as" or "produces" (in some texts the symbol ::= is used in preference to "→"). In such productions, both leftside and definition consist of a string concatenated from one or more terminals and non-terminals

Example :

expr→expr op expr

expr→(expr)

expr→id

op →+│-│\*│/│↑

**Notationl convention**

The notational shorthand can be summarized

1-These symbols are usually nonterminals:

a-lower-case names such as expression , statement .

b-Italic capital letters near the beginning of the alphabet .

c-The letter S, where it appears, is usually the start symbol .

2-These symbols are usually terminals

a-single lower-case letters a , b ,c,….

b-operator symbols like +,-,etc .

c-Punctuation symbols such as parentheses , comma ,etc.

d-The digits 0,1,… ,9

e-Boldface string such as **id** and **if** .

3-Cabital symbols near the end of the alphabet such as X,Y,Z represents

a grammar symbols , that is either terminal or nonterminals .

4-Small letters near the end of the alphabet like u ,v,…,z represent strings of terminals .

5-Lower case Greek letters such as μ, γ , β , δ for example represents

string of grammar symbols .

Example

E→EAE│(E) │-E│id

A→+│-│\*│/│↑

**Derivation and parse tree**

We can create a graphical representation for derivations that filters out the choice regarding replacement order . This representation is called the parse tree . The two specific orders of derivation, which are important from the point of view of parsing, are:

1. Left-most order of derivation

2. Right-most order of derivation

**Example**

E→E+E│E\*E│(E) │-E│id

The string id+id\*id is a statement of the grammar because

E→E+E→E+E\*E→id+E\*E→id+id\*E→id+id\*id

This derivation is called leftmost derivation , we can produce another derivation , which is called rightmost derivation as follows

E→E\*E→E+E\*E→E+E\*id→E+id\*id→id+id\*id

E

E

E

+

E

E

E

\*

id

id

id

Leftmost derivation

E

E

E

\*

E

E

E

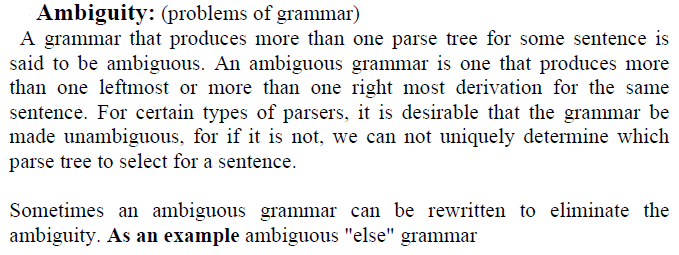
+

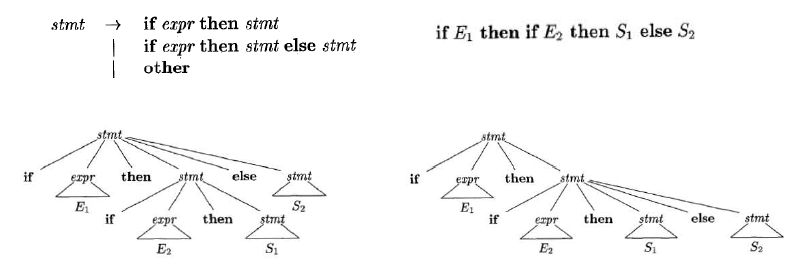
id

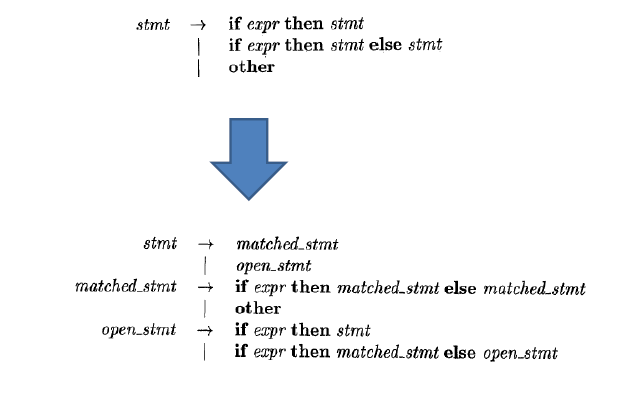
id

id

Rightmost derivation







Example:

E→E+E│E-E

│E\*E│E/E

│E↑E│(E)

│-E│id

This grammar is ambiguous . However, we can disambiguate this grammar by specifying the associativity and the precedence of the arithmetic operators. The order of the operator precedence is

-(unary)

↑ (right associative)

\* / (left associative)

+ - (left associative)

E→E+ term

│E- term

│term

term→term\* factor

│term/ factor

│factor

factor→primary↑ factor

│ primary

primary→-element │ element

element→(expr)

│id

Where the level represent the precedence , left associative represent the left recursive and the right associative represent the right recursive.

Example:

bexp → bexp or bexp bexp→ bexp or F / F

**/** bexp and bexp F → F and G / G

/ not bexp G→ not G / true / false

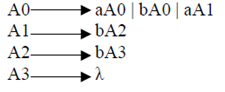
/true

/false

**Context-Free Grammars Versus Regular Expressions**

we can establish that grammars are a more powerful notation than regular expressions. Every construct that can be described by a regular expression can be described by a grammar, but not vice-versa. Alternatively, every regular language is a context-free language, but not vice-versa.

For example, the regular expression *(alb)\*abb* and the grammar



describe the same language, the set of strings of a's and b's ending in abb.

We can construct mechanically a grammar to recognize the same language as a nondeterministic finite automaton (NFA). The grammar above was constructed from the NFA in the figure below using the following construction:

1. For each state i of the NFA, create a nonterminal Ai***.***

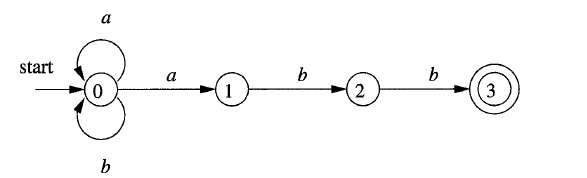
**2.** If state i has a transition to state j on input a, add the production

Ai → aAj.

If state igoes to state j on input ***ϵ,*** add the production Ai → Aj.

**3.** If iis an accepting state, add Ai→ ***ϵ***.

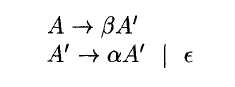
4. If i is the start state, make Ai be the start symbol of the grammar.

****

**Elimination of Left Recursion**

*A* grammar is *left recursive* if it has a nonterminal A such that there is a derivation A→ Aα for some string a. Top-down parsing methods cannot

handle left-recursive grammars, so a transformation is needed to eliminate left recursion. We showed how the left-recursive pair of productions A → Aα | β could be replaced by the non-left-recursive productions:



for many grammars.

**Example** : The non-left-recursive expression grammar , repeated here,

*E* → T *E' E* → *E* + *T* | T

*E'* → + *T E'* | ***ϵ*** *T* → *T* \* *F* | F

*T* → F T*' F* → *( E )* | ***id***

*T'* → \* *FT'* | ***ϵ***

*F* → *( E )* | ***id***

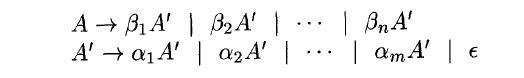
***Non-left recursive grammar left recursive grammar***

is obtained by eliminating immediate left recursion from the expression grammar. The left-recursive pair of productions *E* → *E* + *T* | T are replaced by *E* → T *E'* and *E'* → + *T E'* | ***ϵ .*** The new productions for T and *T'* are obtained similarly by eliminating immediate left recursion.

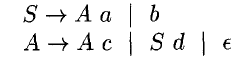
Immediate left recursion can be eliminated by the following technique, which works for any number of A-productions.



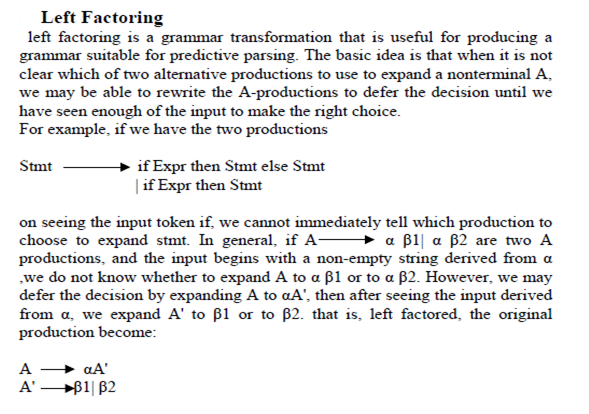
First, group the productions as where no β***i*** begins with an A. Then, replace the A-productions by

****

The nonterminal A generates the same strings as before but is no longer left recursive. This procedure eliminates all left recursion from the A and A' productions (provided no αi is ϵ), but it does not eliminate left recursion involving derivations of two or more steps. For example, consider the grammar



The nonterminal S is left recursive because S → Aa → Sda, but it is not immediately left recursive.



Stmt→  **if** Expr **then** StmtA*'*

A*'* → elseStmt  *|* ***ϵ***