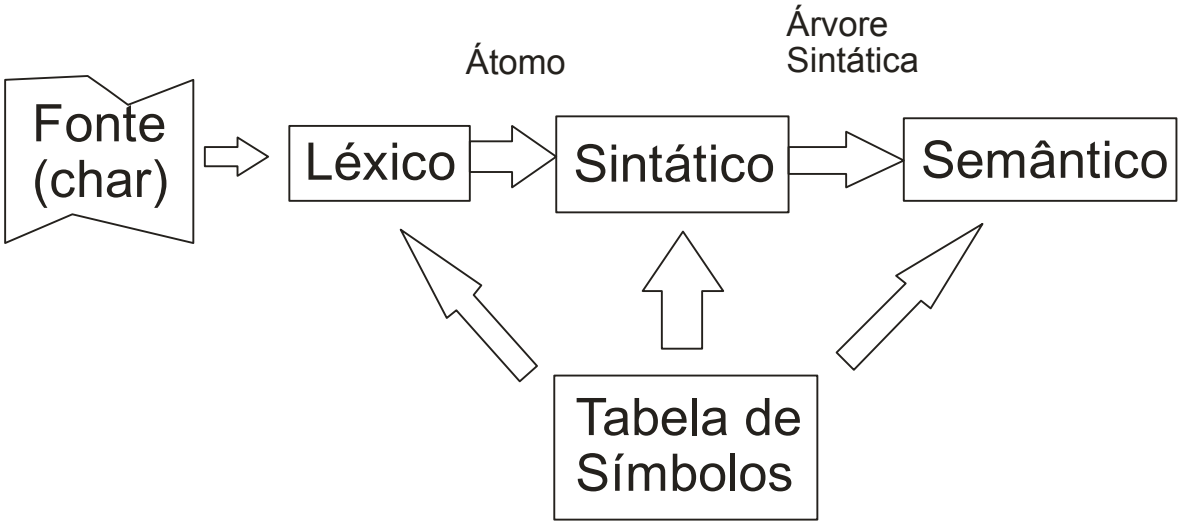
Análise Sintática.



A análise sintática de uma linguagem de programação de programação trata dos aspectos livres de contexto da LP. Estes aspectos são representados por uma gramática que é baseada nos átomos (*tokens*) da LP, que são fornecidos pelo analisador léxico.
Os aspectos dependentes de contexto são tratados por meio de rotinas durante a análise semântica (formam a chamada semântica estática).

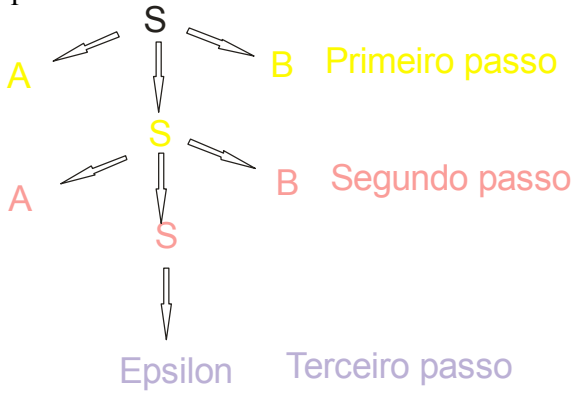
Linguagens Livres de Contexto

São linguagens que podem apresentar *aninhamento sintático* e são geradas por gramáticas livres de contexto.
Uma gramática livre de contexto apresenta regras de produção cujo lado esquerdo possui *apenas 1 não terminal*. Desta maneira não há contexto para a aplicação das regras. Exemplo:

$G = (\{S\},\{a,b\}, \{s \rightarrow aSb \mid \varepsilon\},s)$
$S \Rightarrow \varepsilon$
$S \Rightarrow aSb \Rightarrow a\varepsilon b = ab$
$S \Rightarrow aSb \Rightarrow a\,(aSb)b \Rightarrow aa\varepsilon bb = a^2b^2$
$F(G) = a^nb^n\ ,\ n > 0$

Árvore de Derivação.

Representa uma derivação específica na qual o não terminal inicial é a raiz da árvore e as folhas são terminais Exemplo: a^2b^2



Gramática Ambígua

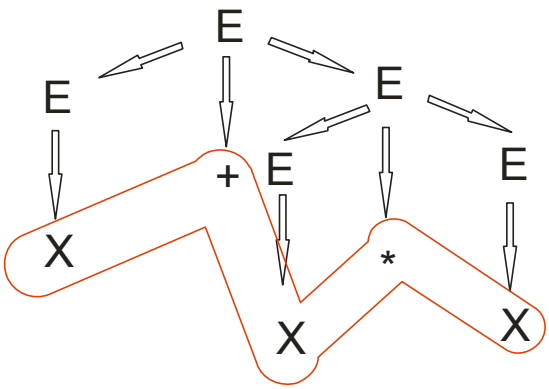
É uma gramática que possui mais de uma árvore de derivação para a mesma sentença (cadeia).
O mesmo ocorre com as derivações mais à esquerda (ou direita).
Exemplo:

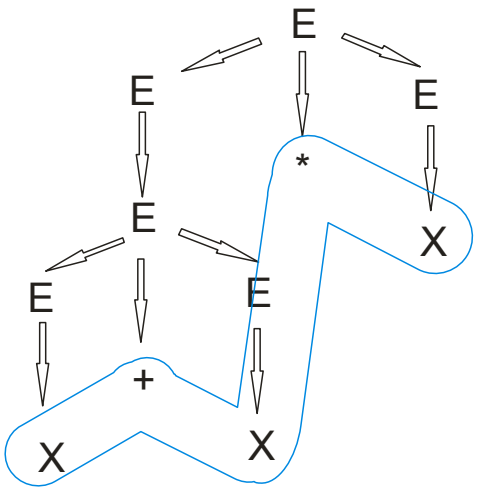
G = ({E}, {+, *, X}, {E →E + E | X, E →E * E}, E)

X + X * X

$E \Rightarrow E + E \Rightarrow X + E \Rightarrow X + E * E \Rightarrow X + X * E \Rightarrow X + X * X$

$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow X + E * E \Rightarrow X + X * E \Rightarrow X + X * X$





Problemas:

1. **Ambigüidade**

Para as LPs a ambigüidade deve ser evitada. Nos casos aqui estudados é possível encontrar uma GLC não ambígua.

2. **Cadeia Vazia**

Regras de produção do tipo X → ε trazem dificuldade
Deve-se eliminar essas regras

G = ({S,A},{a,b},{S→aA | ε, A→Sb | ε},S)
V = {S,A}

A → aAb | Sb | b
S → aSb | aA | a
S' → S | ε

3. Recursividade à esquerda

A → Aa | a , o não terminal da regra é o primeiro do lado direito da regra

A → Ab1 | Ab2 | ... | Abn | a1 | a2 | ... | am onde $a_i \cup b_i \subseteq V^+$
A → a1X |a2X| ... | amX | a1 | ... | am
X → b1X |b2X| ... | bnX | ε b1 | ... | bn

Recursividade Cruzada

1. Renomear não terminais se $A_r \rightarrow A_s b$ e r > s
2. Substituir A_s por suas regras
3. Resolver a recursividade explícita

Exemplo:

S → Aa | ab
A → Sb | b

A1 → A2a | ab
A2 → A1b | b
A2 → A2ab | abb | b

A2 → A2ab | abb | b
A2 → abbX | bX
X → abX | ε

4. Gramática Não Fatorada

A → aB | aC onde $a \subseteq V^+$

a = 'bcbdb'

Fatoração: Eliminação dos prefixos comuns:

A → aX
X → B | C

Representações de LP's

BNF (Backus-Naur Form)

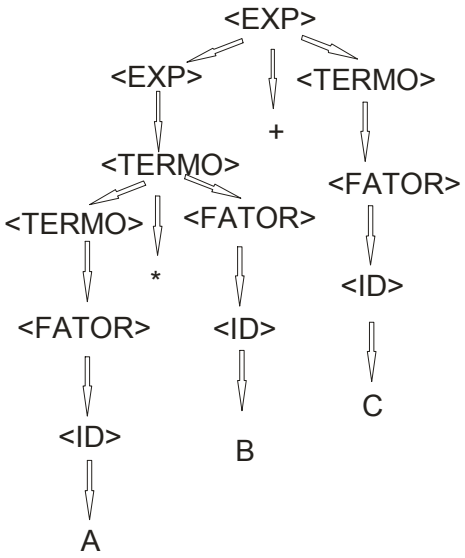
<X> → indica não terminal X
X → indica terminal X
::= → "é definido como"
| → alternativa

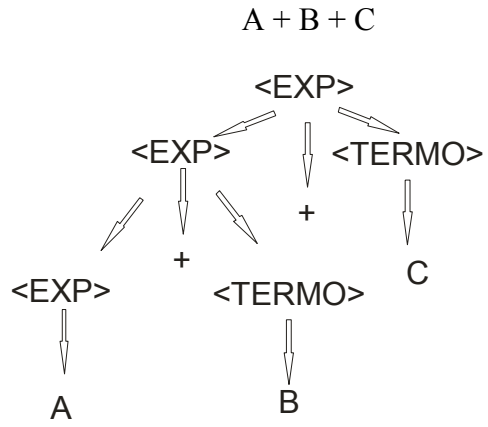
Exemplo :

G = ({E,T,F,I}{A,B,C,+,*},(},P,E)
P = {E → E + T | T, T → T * F | F , F →(E) | I , I → A | B | C }
<EXP> ::= <EXP> + <TERMO>
| <TERMO>
<TERMO> ::= <TERMO> * <FATOR>
| <FATOR>
<FATOR> ::= (<EXP>)
| <ID>
<ID> ::= A | B | C

Exemplo: A * B + C

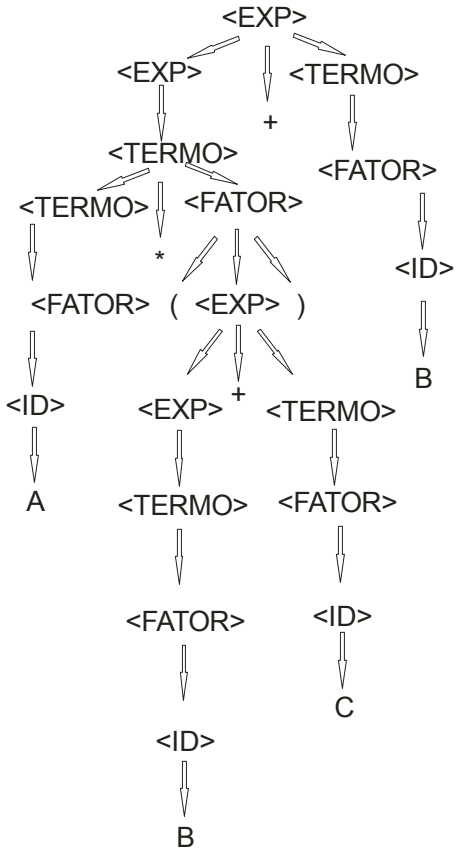
<EXP> ⇒ <EXP> + <TERMO>
⇒ <TERMO> + <TERMO>
⇒ <TERMO> * <FATOR> + <TERMO>
⇒ <FATOR> * <FATOR> + <TERMO>
⇒ <ID> * <FATOR> + <TERMO>
⇒ A * <FATOR > + <TERMO>
⇒ A * <ID> + <TERMO>
⇒ A * B + <TERMO>
⇒ A * B + <FATOR>
⇒ A * B + <ID>
⇒ A * B + C





Exemplo:

A * (B + C) + B



2) Extended BNF (EBNF)
(Notação de Wirth)

X → não terminal X
"X" → Terminal X
= → é definido como
| → Alternativa
(X) → o mesmo que X
[X] → X ou ε
{X} → X*
. → terminador da regra

Exemplo:

EXP = EXP "+" TERMO | TERMO.
TERMO = TERMO "*" FATOR | FATOR .
FATOR = "(EXP)" | ID.
ID = "A"| "B" | "C".

TERMO = FATOR {"*" FATOR}.
EXP = TERMO{"+"TERMO}.
FATOR = "("EXP ")" | "A" | "B" | "C"
TERMO = ("("EXP ")" | "A" | "B" | "C") { "*" ("(" EXP ")" | ...) }.
EXP = ...

Análise Sintática

Pode-se realizada das formas:

1 Descendente: gera a árvore a partir do não terminal inicial para as folhas;
2 Ascendente: gera uma árvore a partir das folhas para o não terminal inicial;

Análise Descendente:

Pode ser recursiva (não-determinística ou determinística) ou a partir de tabelas de regras de transição (autômato de pilha)

Análise descendente Recursiva : Criação do reconhecedor da gramática em notação de Wirth

***Pressuposto:** Léxico é uma Rotina que captura o lexema e armazena em um *token*.

```
Função FATOR( ) : Boolean
Início
        Se token é "(" então
                Léxico;
                EXP( );
                Se token é ")" então
                        Léxico;
                        Retorna true
                Senão  Retorna false
        Senão
                Se token é "A" ou "B" ou "C"
                        Léxico;
                        Retorna true
                Senão  Retorna false
Fim Fator
```

```
Função TERMO( ) : Boolean
Início
        Se FATOR( ) então
                Enquanto token é "*"
                        Léxico;
                        Se não FATOR( ) então
                                Retorna false
                Fim enquanto
                Retorna true
        Senão
                Retorna false
Fim Termo
```

```
Função EXP( ) : Boolean
Início
        Se TERMO( ) então
                Enquanto token é "+"
                    Léxico;
                    Se não TERMO( ) então
                                Retorna false
                Fim enquanto
                Retorna true
        Senão
                Retorna false
Fim Exp
```

```
Função Sintático( ) : Boolean
Início
        Léxico;
        Se EXP( ) então
                Retorna true
        Senão
                Retorna false
Fim Sintático
```

A função "Sintático" é a encarregada de iniciar a análise sintática. Ao efetuar uma chamada ao Léxico o primeiro átomo da cadeia de entrada está disponível na variável *token* para ser processada. A partir deste ponto a análise descendente é processada pela função EXP( ).