E ::= a | b | +EE | \*EE

A::= y1 |y2 | y3 | y4

1-Toda produção é da forma A::= yα, onde y é um terminal.

2-Se A::= y1α1| y2α2 |....| ynαn são todas as alternativas para o não-terminal A, então os terminais yi são todos diferentes entre si.

Cálculo do FIRST para uma gramática

* 1. Se α for um terminal, então FIRST(α) é {α}.
  2. Se α::=λ for uma produção, adiciona λ ao FIRST(α).
  3. Se α for um não-terminal e α::= y1 y2 ...yk uma produção, colocar em FIRST(α) tudo o que estiver em FIRST(y1). Se y1 derivar λ, então é adicionado FIRST(α) o FIRST(y2), e assim por diante, só é adicionado ao FIRST(α) o λ se forem adicionados sucessivamente os FIRST(y1), FIRST(y2) até FIRST(yk-1) e yk derivar λ.

S ::= AS | BA S::= FIRST(A) α1| FIRST(B) α2 => S::= {a,c}α1|{b,d}α2

A ::= aB | C A::= FIRST(a) α1| FIRST(C) α2

B ::= bA | d B::= FIRST(b) α1| FIRST(d) α2

C ::= c

T={a, b, d, c }

N={S,A,B,C}

FIRST(C) = { c }

FIRST(B) = {b,d}

FIRST(A) = {a,c}

FIRST(S) = {a,c,b,d}

Se A::= y1α1| y2α2 |....| ynαn são todas as alternativas para o não-terminal A, então os conjuntos de **FIRST(yi)** são disjuntos dois a dois.

Assim **yi**pode ser um **terminal** ou **não-terminal**.

Cálculo do FIRST para uma gramática

1. Se α for um terminal, então FIRST(α) é {α}.
2. Se α::=λ for uma produção, adiciona λ ao FIRST(α).
3. Se α for um não-terminal e α::= y1 y2 ...yk uma produção, colocar em FIRST(α) tudo o que estiver em FIRST(y1). Se y1 derivar λ, então é adicionado FIRST(α) o FIRST(y2), e assim por diante, só é adicionado ao FIRST(α) o λ se forem adicionados sucessivamente os FIRST(y1), FIRST(y2) até FIRST(yk-1) e yk derivar λ.

Como ficaria o FIRST da gramática abaixo:

S ::= ABCDd

A ::= aA | λ

B ::= bC | λ

C ::= cD

C ::= λ Se α::= λ for uma produção, adiciona λ ao FIRST(α).

D ::= e

FIRST(D) = {e}

FIRST(C) = {c, λ}

FIRST(B) = {b, λ}

FIRST(A) = {a, λ}

FIRST(S) = {a,b,c,e}

Como ficaria o FIRST da gramática abaixo:

S ::= ABCDd

A ::= aA | λ

B ::= CD | λ

C ::= cD | λ Se α::= λ for uma produção, adiciona λ ao FIRST(α).

D ::= e

FIRST(D) = {e}

FIRST(C) = {c, λ}

FIRST(B) = {FIRST(C),FIRST(D) }={c,FIRST(D)}= {c,e}

FIRST(A) = {a, λ}

FIRST(S) = {a,b,c,e}

**ASDR – eliminação de recursividade à esquerda**

E ::= E + E | num

onde num = [0-9], ou seja, 0 | 1 | 2 | 3 | .. | 9

palavras geradas pela gramática: 1, 2, 3, 1+2, 3+1,1+2+3, 1+2+3+...

É uma gramática ambígua ?

Sim

Para construir um Analisador Sintático Descendente Recursivo é necessário que a gramática **não seja** **ambígua**.

Para eliminar a ambiguidade precisamos reescrever a gramática

E::= E + F | F

F::= num

Dá pra fazer ASDR para gramática?

E::= E + F

void E(){

E();

if( \*palavra == ´+´)

palavra++;

else

erro()

F();

}

Recursiva à esquerda, não dá pra implementar o ASDR.

Note que a gramática produz palavras com o padrão:

E::= E + F | F

F::= num

E => E + F => E+F+F => E+F+F+F => E.....+F+F+F=>F.....+F+F+F

E=> F

void E(){

F();

while( \*palavra == ‘+’){

palavra++; // consome o terminal +

F();

}

}

Regra para eliminar a recursão à esquerda

E::= E + F | F

F::= num

Assim a construção **{ α }** numa produção denotará a repetição da **cadeia** α zero ou mais vezes (α\*).

De maneira geral temos:

A::= y1 | y2 | ... | yn | Aα

E::= F | E + F

Pode ser reescrito por:

A::= (y1 | y2 | ... | yn) { α }

E::= F{+F}

A gramática completa reescrita fica assim:

E::= F{+F}

F::= num

onde num = [0-9], ou seja, 0 | 1 | 2 | 3 | .. | 9

Regra para eliminar a recursão à esquerda da gramática adicionando a produção T::= T\*F | F

E ::= E+T | T 🡺 E::=T{+T}

T::= T\*F | F 🡺 T::=F{\*F}

F::= num

Agora acrescentando parênteses na regra (produção) da gramática, lembrando que os parênteses mudam a prioridade dos operadores, ou seja, os operadores entre os parênteses devem ser resolvidos primeiro, na árvore de derivação devem estar mais próximo as folhas.

E::=T{+T}

T::=F{\*F}

F::= (E) |num