Análise Sintática Descendente

- Uma tentativa de construir uma árvore de derivação da esquerda para a direita
- Cria a raiz e, a seguir, cria as subárvores filhas.
- Produz uma derivação mais à esquerda da sentença em análise.
- Constrói a árvore de derivação para a cadeia de entrada de cima para baixo (top-down).

Análise Descendente (Top-down)

- Constrói da raiz para as folhas
- Há duas formas de analisadores sintático descendentes:
  - **Analisadores com retrocesso**
  - **Analisadores preditivos**

Análise Sintática Descendente

- **Analisador sintático preditivo**
  - Tenta prever a construção seguinte na cadeia de entrada com base em uma ou mais marcas de verificação à frente.
- **Analisador sintático com retrocesso**
  - Testa diferentes possibilidades de análise sintática de entrada, retrocedendo se alguma possibilidade falhar.
  - São mais poderosos.
  - São mais lentos.

Análise Sintática Descendente

- A sequência de árvores de derivação para a entrada **id+id*id** representa uma análise sintática descendente de acordo com a gramática:

E → T E’

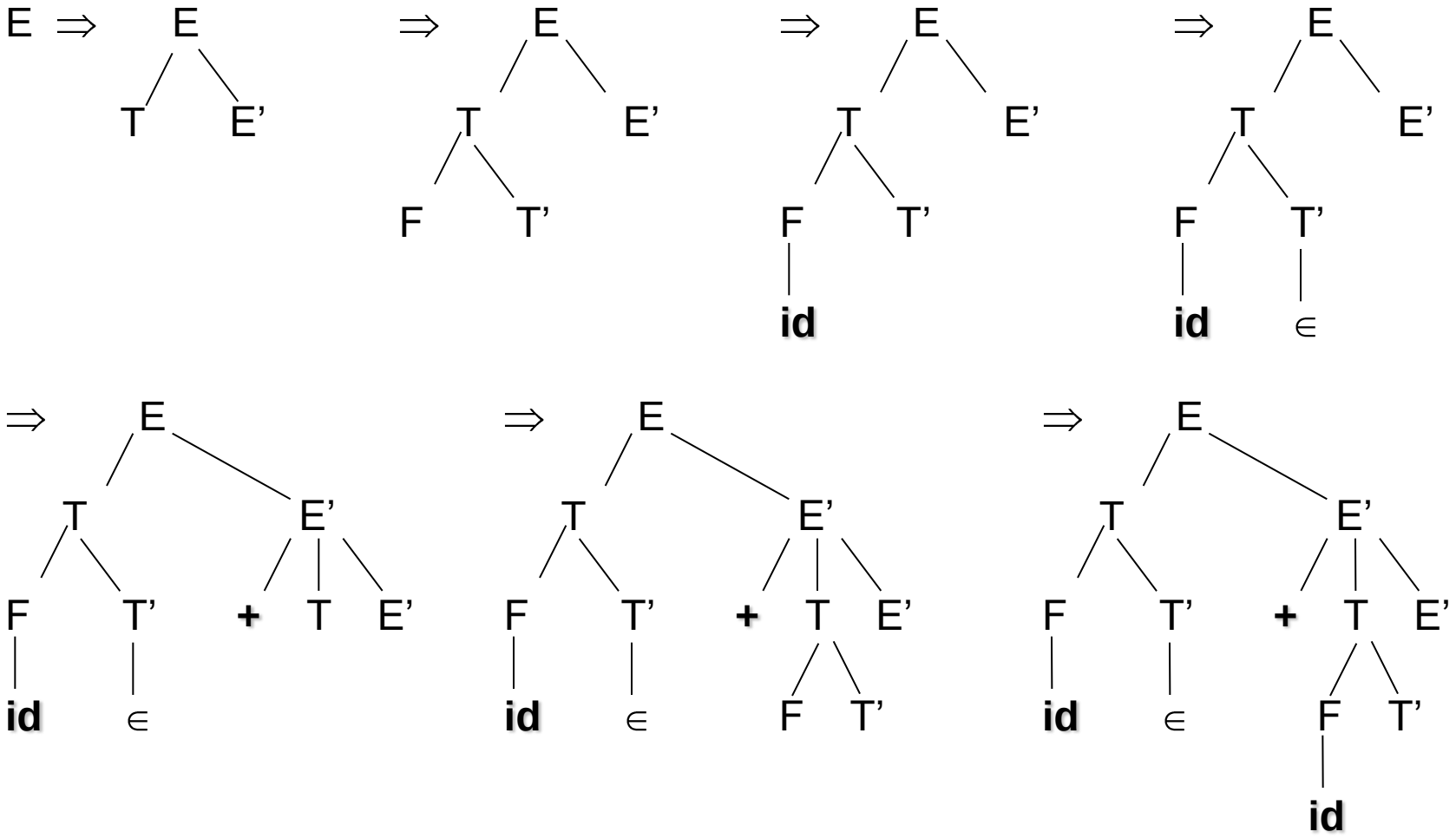
E’ → + T E’ | ∈

T → F T’

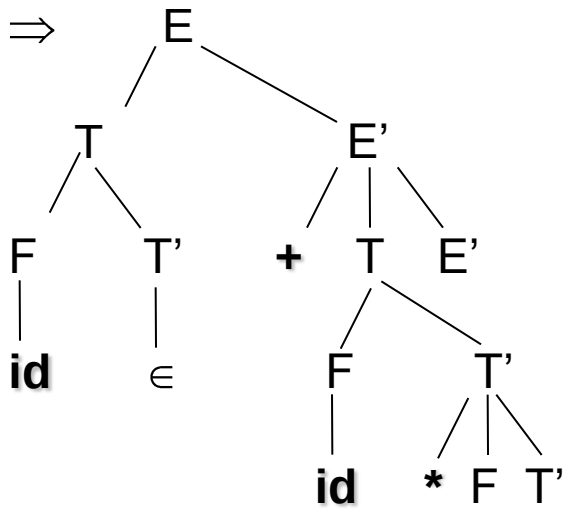
T’ → * F T’ | ∈

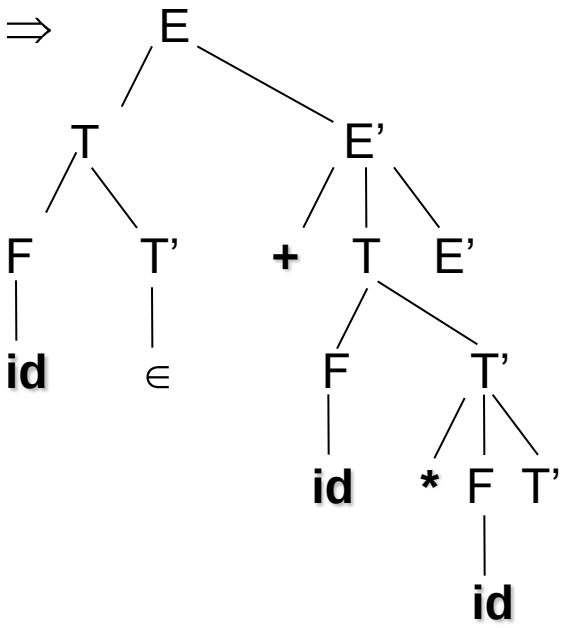
F → (E) | **id**

Análise Sintática Descendente para **id+id*id**

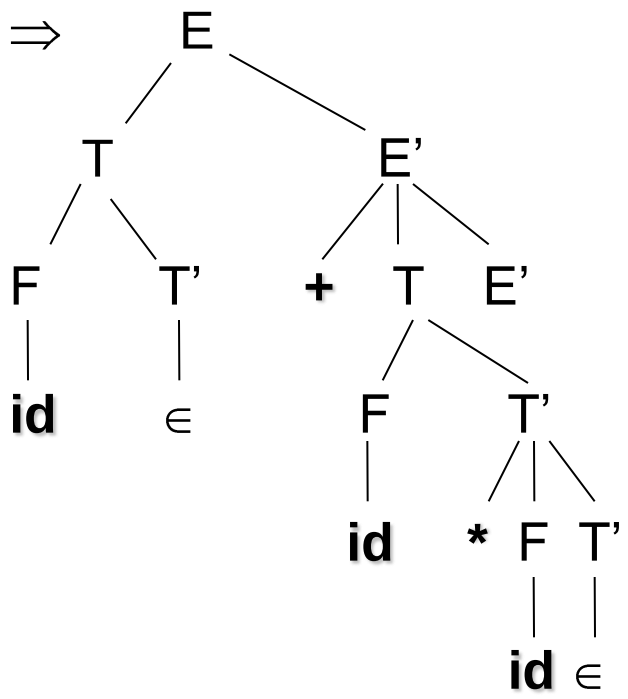


Análise Sintática Descendente para **id+id*id**

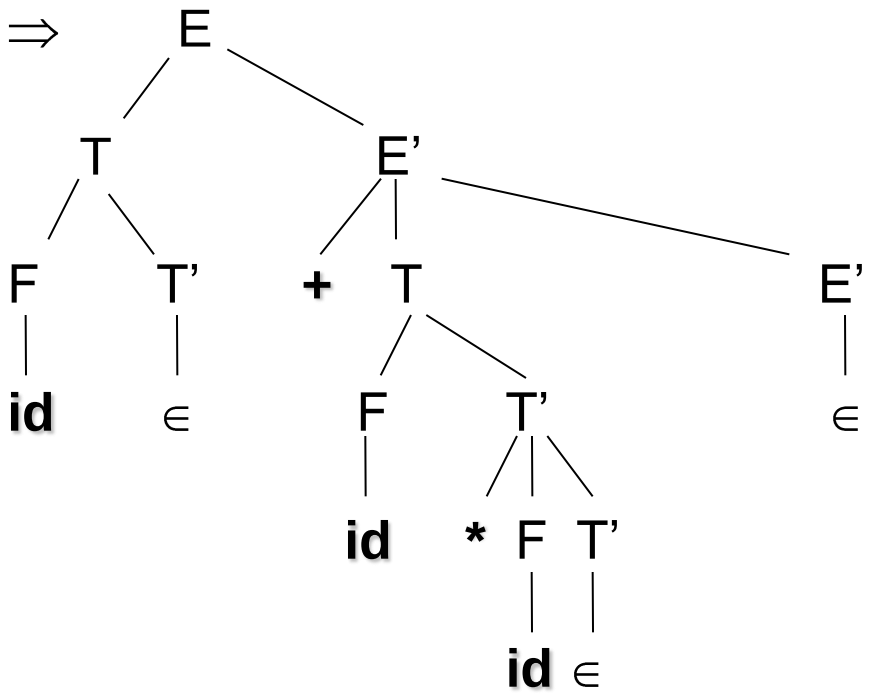




Análise Sintática Descendente para **id+id*id**



Análise Sintática Descendente para **id+id*id**



Análise Sintática Descendente para **id+id*id**

- Corresponde a uma derivação mais à esquerda da entrada.
- A análise sintática decrescente apresentada introduziu o método de reconhecimento sintático preditivo (análise de descida recursiva).
- A análise sintática descrescente apresentada constrói uma árvore com dois nós rotulados com E’.

Análise Sintática Descendente para **id+id*id**

- No primeiro nó E', a produção E' → +TE' é a escolhida.
- No segundo no E', a produção E' → $\in$ é escolhida.
- Um analisador preditivo pode escolher entre as produções examinando o próximo símbolo da entrada.
- A classe de gramáticas para as quais podemos construir analisadores preditivos examinando *k* símbolos adiante na entrada é chamada *LL(k)*.

Análise Sintática LL(k)

- Na prática não é utilizada com tanta frequência.
- É útil como estudo de um esquema com uma pilha explícita.
- Pode servir como introdução para os algoritmos ascendentes (mais poderosos e complexos)
- É útil para formalizar alguns problemas que aparecem na análise descendente recursiva.
- O primeiro L se refere ao fato do processamento ocorrer da esquerda para a direita (left)
- O segundo L se refere ao fato de acompanhar uma derivação à esquerda para a cadeia de entrada.
- Podemos ver o número 1 ou a letra K entre parênteses que significa a verificação de quantos símbolos à frente (é mais comum verificar apenas um símbolo à frente).

Análise Sintática Descendente Recursiva

- Consiste em um conjunto de procedimentos, um para cada não-terminal da gramática.
- A execução começa com a ativação do procedimento referente ao símbolo inicial da gramática, que pára e anuncia sucesso se o seu corpo conseguir escandir toda a cadeia da entrada.
- Pode exigir retrocesso, voltar atrás no reconhecimento, fazendo repetidas leituras sobre a entrada.
  - Raramente presentes nas linguagens de programação.
  - Não é muito eficiente.
  - Os preferidos são os baseados em tabelas, como o algoritmo de programação dinâmica.

Análise Sintática Descendente Recursiva

- Pseudocódigo típico para um não-terminal

```
void A() {
    Escolha uma produção-A, A → x1 x2 ... xk
    for (i = 1 até k) {
        if (xi é um não-terminal)
            ativa procedimento xi();
        else if (xi igual ao símbolo de entrada a)
            avance na entrada para o próximo símbolo terminal;
        else /* ocorreu um erro */;
    }
}
```

- Esse pseudocódigo é não determinista, pois escolhe a produção-A a ser aplicada de uma maneira arbitrária

Análise Sintática Descendente Recursiva

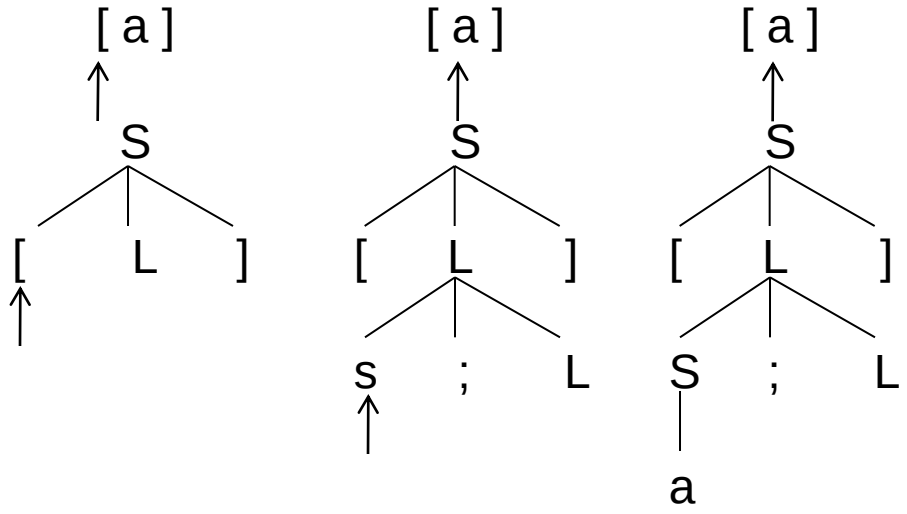
- Análise recursiva com retrocesso.
  - Considere a sentença [a] derivada a partir da gramática abaixo:

    S → a | [L]

    L → S ; L | S

Análise Sintática Descendente Recursiva

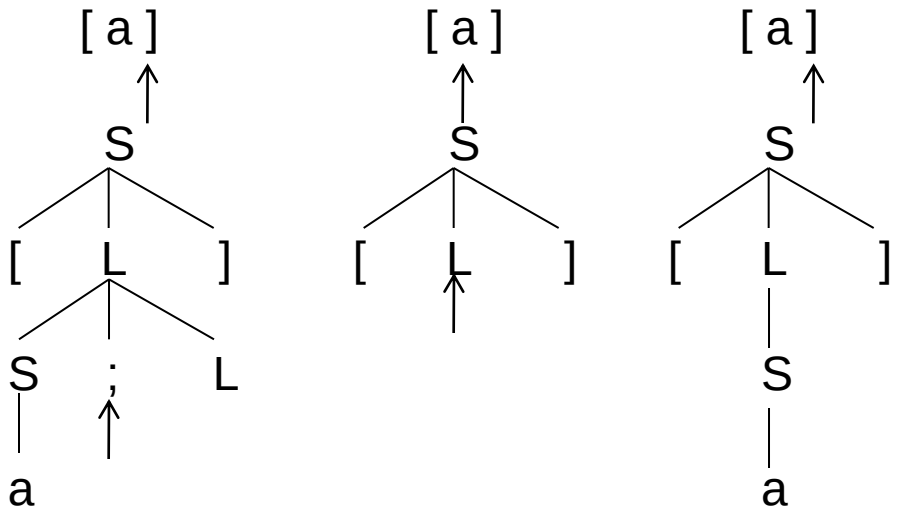
- Reconhecimento da sentença [a]



• O reconhecimento de **[** é bem sucedido.
• A derivação de **L** é efetuada usando **S ; L**.
• **S** é expandido novamente, obtendo-se sucesso.

Análise Sintática Descendente Recursiva

- Reconhecimento da sentença [a]



- A comparação seguinte (] com ;) falha.
- O analisador deve retroceder para o ponto em que estava por ocasião da opção pela primeira alternativa de **L**.
- É aplicada a segunda alternativa, **L → S**.
- A derivação final é obtida aplicando-se a produção **S → a**.

Análise Sintática Descendente Recursiva

- Analisador recursivo com retrocesso.
- Programa principal:

```
begin
    token := LETOKEN;
    if S
    then
      if token = '$' then write ('SUCESSO') else write ('ERRO')
    else write ('ERRO')
end
```

Análise Sintática Descendente Recursiva

- Analisador recursivo com retrocesso.

```
function S;
    if token = 'a'
    then {token := LETOKEN; return true}
    else
      if token = '['
      then {token := LETOKEN;
                  if L
                  then if token = ']'
                          then {token := LETOKEN; return true}
                          else return false
                  else return false
      else return false}
    else return false
```

Análise Sintática Descendente Recursiva

- Analisador recursivo com retrocesso

```
function L;
    MARCA_PONTO;
    if S
    then if token = ';'
            then {token := LETOKEN;
                  if L
                  then return true
                  else return false}
            else {RETROCEDE;
                  if S
                  then return true
                  else return false}
    else return false
```

Análise Sintática Descendente Recursiva

- Analisador recursivo com retrocesso
  - LETOKEN retorna um *token* lido a partir da sentença de entrada.
  - MARCA_PONTO marca, na sentença de entrada, um ponto de possível reinício da análise.
  - RETROCEDE volta o ponteiro de leitura para o último ponto marcado.

Análise Sintática Descendente Recursiva

- Considere a gramática de expressões a seguir:

  E → E +|- T | T

  T → T * F | F

  F → (E) | id

- Considere a regra gramatical para um *fator* F.
  - OBS.: E = exp; T = termo; F = fator

Análise Sintática Descendente Recursiva

- Pseudocódigo de um procedimento descendente recursivo para reconhecer um *fator.*

```
procedure fator;
begin
    case marca of
    ( : casamento( ( );
        exp;
        casamento( ) );
    id :
        casamento(id);
    else erro;
    end case;
end fator;
```

- Nesse pseudocódigo existe uma variável *marca* para registrar a marca seguinte da entrada, e um procedimento *casamento* que casa a marca seguinte com seu parâmetro.

Análise Sintática Descendente Recursiva

- Procedimento *casamento*

```
procedure casamento (marcaEsperada);
begin
        if marca = marcaEsperada then
                capturaMarca;
        else
                erro;
        end if;
end;
```

Relembrando

- BNF ou forma de Backus-Naur, as gramáticas livres de contextos.
- EBNF ou BNF estendida, as construções repetitivas e opcionais

Análise Sintática Descendente Recursiva

- Repetição e escolha: EBNF
  - Exemplo, a regra gramatical (simplificada) para uma declaração if:

If-decl → **if** (*exp*) *declaração*

| **if** (*exp*) *declaração* **else** *declaração*

Análise Sintática Descendente Recursiva

Tradução para o procedimento

```
procedure declIf;
begin
    casamento (if);
    casamento ((();
    exp;
    casamento ());
    declaração;
    if marca = else then
        casamento (else);
        declaração;
    end if;
end declIf;
```

- não dá para distinguir de imediato as duas escolhas à direita da regra gramatical.
- podemos adiar a decisão de reconhecer a parte opcional *else* até encontrar a marca **else** na entrada.
- o código para declaração if em EBNF

if-decl → **if** (exp) declaração [**else** declaração]

Análise Sintática Descendente Recursiva

- Considere o caso de E na gramática para expressões aritméticas simples em BNF:

$$E \rightarrow E + | - T \mid T$$

- Se ativar o E em um procedimento E recursivo, levaria a um laço recursivo infinito.
- Um teste para efetuar uma escolha entre ***E → E +|- T*** e ***E → T***, é problemático, pois tanto E como T podem começar com parênteses à esquerda.

Análise Sintática Descendente Recursiva

- A solução é o uso da regra EBNF:

$$E \rightarrow T \{+ \mid - T\}$$

- As chaves expressam repetição.

Análise Sintática Descendente Recursiva

- Tradução em código para um laço:

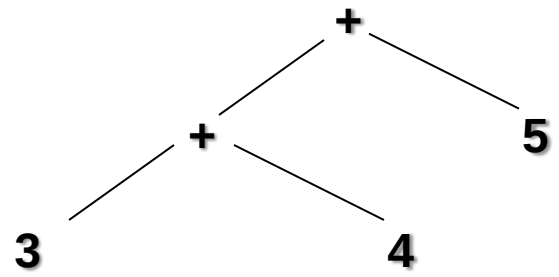
```
procedure exp;
begin
    termo;
    while marca = + or marca = - do
        casamento (marca);
        termo;
    end while;
end exp;
```

De maneira similar, gere um código para a regra EBNF a seguir:

T → F {* F}

Análise Sintática Descendente Recursiva

- Considere a expressão **3 + 4 + 5**, cuja árvore sintática é:



O nó que representa a soma de 3 e 4 de ser processado antes do nó-raiz.

- A árvore sintática gera o pseudocódigo a seguir:

```
function exp : árvoreSintática;
var temp, novatemp : árvoreSintática;
begin
    temp := termo;
    while marca = + or marca = - do
        novatemp := criaNóOp(marca);
        casamento (marca);
        filhoEsq(novatemp) := temp;
        filhoDir(novatemp) := termo;
        temp := novatemp;
    end while;
    return temp;
end exp;
```

• a função cirNóOp recebe uma marca de operador como parâmetro e retorna um novo nó de árvore sintática.
• o procedimento *exp* constrói a árvore sintática em vez da árvore de análise sintática.

Construa um pseudocódigo de uma árvore sintática para uma declaração if *de forma estritamente descendente, com o uso de um analisador sintático descendente recursivo*

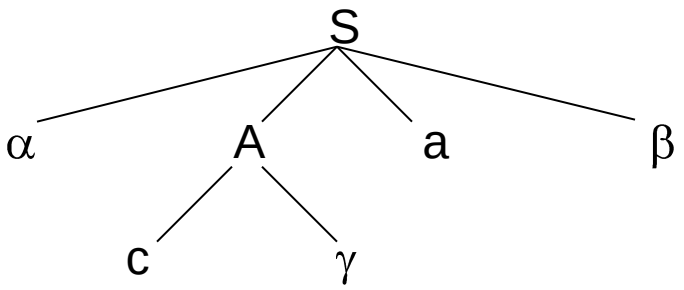
Análise Sintática Descendente Recursiva

- **First e Follow**
  - Funções associativas a uma gramática G.
  - Permitem escolher qual produção aplicar, com base no próximo símbolo da entrada.
  - Durante a recuperação de erro conjuntos de tokes produzidos por FOLLOW podem ser usados como tokens de sincronismo.

Análise Sintática Recursiva Recursiva

- FIRST(α), onde α é qualquer cadeia de símbolos da gramática, como sendo o conjunto de símbolos terminais que iniciam as cadeias derivadas de α.
- As regras abaixo definem esse conjunto:
  1) Se $\alpha \Rightarrow^* \varepsilon$ então ε é um elemento de FIRST(α);
  2) Se $\alpha \Rightarrow^* a\delta$ então **a** é um elemento de FIRST(α), sendo **a** um símbolo terminal e δ uma forma sentencial qualquer, podemos ser vazia.

Exemplo: $A \Rightarrow^* c\gamma$, porque **c** está em FIRST(A).



O símbolo não terminal **c** está em FIRST(A) e o símbolo terminal **a** está em FOLLOW(A).

Dado um símbolo não-terminal A definido por várias alternativas:

$$A \rightarrow \beta_1 \mid \beta_2 \mid ... \mid \beta_n$$

a implementação de um reconhecedor recursivo preditivo para A exige que os conjuntos FIRST de β_1, β_2, ..., β_n sejam disjuntos dois a dois.

Considere a gramática sem recursão à esquerda:

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \in \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \in \\
F &\rightarrow (E) \mid id
\end{aligned}
$$

1. FIRST(*F*) = FIRST(*T*) = FIRST(*E*) = {(,id}
2. FIRST(*E'*) = {+,∈}
3. FIRST(*T'*) = {*,∈}
4. FOLLOW(*E*) = FOLLOW(*E'*) = {),$}
5. FOLLOW(*T*) = FOLLOW(*T'*) = {+,),$}
6. FOLLOW(*F*) = {+,*,),$}

Para entender:

• As duas produções para F possuem corpos iniciando com os símbolos terminais id e o parênteses esquerdo.

• T possui somente uma produção, e seu corpo começa com F.

• Como F não deriva $\in$, o FIRST(T) deve ser igual ao FIRST(F). O mesmo se aplica ao FIRST(E).

• Uma das duas produções para E' tem um corpo que começa com o terminal +, e o corpo da outra é $\in$. Sempre que um não-terminal deriva $\in$, incluímos $\in$ em FIRST.

• E é o símbolo inicial da gramática, FOLLOW(E) deve incluir $.

• FOLLOW(E') deve ser o mesmo que FOLLOW(E).

• T aparece do lado direito das produções-E apenas seguido por E'. Assim, tudo exceto $\in$ que está em FIRST(E') deve ser incluído em FOLLOW(T).

  ✓ Isso explica o símbolo +.

• Tudo em FOLLOW(E) também deve ser incluído em FOLLOW(T).

  ✓ FIRST(E') contém $\in$, e E' é a cadeia inteira seguindo T nos corpos das produções-E

  ✓ Isso explica os símbolos $ e o parêntese direito.

  ✓ T' por aparecer apenas nas extremidades das produções-T, é necessário fazer o FOLLOW(T') = FOLLOW(T)

Considere as produções abaixo, que definem os comandos **if-then**, **while-do**, **repeat-until** e **atribuição**:

COMANDO → **if** EXPR **then** COMANDO |
while EXPR **do** COMANDO |
repeat LISTA **until** EXPR |
id := EXPR

Para as produções que definem COMANDO, tem-se os seguintes conjuntos:

FIRST(CONDICIONAL) = { if }
FIRST(ITERATIVO) = { while, repeat }
FIRST(ATRIBUIÇÃO) = { id }

Analisador recursivo preditivo

- A função correspondente ao não-terminal COMANDO:

```
function COMANDO;
        if token = 'if'                          /* token ∈ first(CONDICIONAL) */
        then if CONDICIONAL
                then return true
                else return false
        else if token = 'while'  or token = 'repeat'  /* token ∈ first(ITERATIVO) */
                then if ITERATIVO
                        then return true
                        else return false
                else if token = 'id'                  /* token ∈ first(ATRIBUIÇÃO) */
                        then if ATRIBUIÇÃO
                                then return true
                                else return false
                        else return false
```

Analisador recursivo preditivo com subrotinas

• Os não-terminais são reconhecidos por procedimentos tipo subrotina

• Analisador recursivo preditivo para uma gramática que gera declarações de variáveis

DECL	→	LISTA_ID : TIPO
LISTA_ID	→	id | LISTA_ID, id
TIPO	→	SIMPLES | AGREGADO
SIMPLES	→	int | real
AGREGADO	→	mat DIMENSÃO SIMPLES | conj SIMPLES
DIMENSÃO	→	[num]

Analisador recursivo preditivo com subrotina

• Eliminando-se a recursividade à esquerda das produções acima:

DECL	$\rightarrow$	LISTA_ID : TIPO
LISTA_ID	$\rightarrow$	id L_ID
L_ID	$\rightarrow$	, id L_ID | $\in$
TIPO	$\rightarrow$	SIMPLES | AGREGADO
SIMPLES	$\rightarrow$	int | real
AGREGADO	$\rightarrow$	mat DIMENSÃO SIMPLES | conj SIMPLES
DIMENSÃO	$\rightarrow$	[num]

Programa principal

```
begin
      LETOKEN
      DECL
end
```

```
Procedure LISTA_ID;
  if token = 'id'
  then {LETOKEN;
        L_ID}
  else ERRO(2)
```

```
Procedure DECL;
  LISTA_ID;
  if token = ':'
  then {LETOKEN;
        TIPO}
  else ERRO(1)
```

```
Procedure L_ID;
  if token = ','
  then {LETOKEN;
   if token = 'id'
   then {LETOKEN;
         L_ID}
   else ERRO(3)
  else return;
```

```
Procedure TIPO;
  if token = 'int' or token = 'real'
  then SIMPLES
  else if token = 'mat' or token = 'conj'
       then AGREGADO
       else ERRO(4)

Procedure SIMPLES;
  if token = 'int'
  then LETOKEN;
  else if token='real'
       then LETOKEN
       else ERRO(5)

procedure AGREGADO;
  if token = 'mat'
  then {LETOKEN;
        DIMENSÃO;
        SIMPLES}
  else {LETOKEN;
        SIMPLES}
```

• As produções que derivam a palavra vazia, não é escrito código.

• Na subrotina que implementa o símbolo L_ID:

  ✓ Se o restante da sentença a ser analisada inicia por ‘**,**’, o analisador tenta reconhecer **id** L_ID;

  ✓ Senão, sai da subrotina L_ID sem chamar a rotina de ERRO, significando o reconhecimento de L_ID $\rightarrow \in$

Se α é uma forma sentencial (sequência de símbolos da gramática), então **FIRST(α)** é o conjunto de terminais que iniciam formas sentenciais derivadas a partir de α. Se $\alpha \Rightarrow^* \in$, então a palavra vazia também faz parte do conjunto.

A função **FOLLOW** é definida para símbolos não-terminais. Sendo **A** um não-terminal, **FOLLOW(A)** é o conjunto de terminais **a** que podem aparecer imediatamente à direita de **A** em alguma forma sentencial. O conjunto de terminais **a**, tal que existe uma derivação da forma $\mathbf{S} \Rightarrow^* \alpha\mathbf{A}\mathbf{a}\beta$ quaisquer.

Algoritmo para calcular FIRST(X)

1) Se a é terminal, então FIRST(a) = {a}.
2) Se $X \rightarrow \in$ é uma produção, então adicione $\in$ a FIRST(X).
3) Se $X \rightarrow Y_1 Y_2 ... Y_k$ é uma produção e, para algum i, todos Y_1, Y_2, ..., Y_{i-1} derivam $\in$, então FIRST(Y_i) está em FIRST(X), juntamente com todos os símbolos não-$\in$ de FIRST(Y_1), FIRST(Y_2), ..., FIRST(Y_{i-1}). O símbolo $\in$ será adicionado a FIRST(X) apenas se todo Y_j(j = 1, 2, ..., k) derivar $\in$.

Algoritmo para calcular FOLLOW(X)

1) Se S é o símbolo inicial da gramática e $ é o marcador de fim da sentença, então $ está em FOLLOW(S).
2) Se existe produção do tipo $A \rightarrow \alpha X \beta$, então todos os símbolos de FIRST(β), exceto $\in$, fazem parte de FOLLOW(X).
3) Se existe produção do tipo $A \rightarrow \alpha X$, ou $A \rightarrow aX\beta$, sendo que $\beta \Rightarrow^* \in$, então todos os símbolos que estiverem em FOLLOW(A) fazem parte de FOLLOW(X).

Observe

- A palavra vazia jamais fará parte de algum conjunto de FOLLOW.
- Os conjuntos de FOLLOW são formados apenas por símbolos terminais
- ε não é símbolo terminal.

Funções FIRST e FOLLOW

- Considere a gramática não-ambígua abaixo que gera expressões lógicas:

$$E \rightarrow E \vee T \mid T$$
$$T \rightarrow T \,\&\, F \mid F$$
$$F \rightarrow \neg F \mid id$$

Funções FIRST e FOLLOW

- Ao eliminar a recursividade à esquerda das produções E e T, obtém-se

$$E \rightarrow T\ E'$$

$$E' \rightarrow \vee\ T\ E' \mid \varepsilon$$

$$T \rightarrow F\ T'$$

$$T' \rightarrow \&\ F\ T' \mid \varepsilon$$

$$F \rightarrow \neg\ F \mid id$$

Funções FIRST e FOLLOW

- A tabela de análise preditiva:

	id	$\vee$	&	$\neg$	$
E	$E \rightarrow T\ E'$			$E \rightarrow T\ E'$	
E'		$E' \rightarrow \vee\ T\ E'$			$E' \rightarrow \varepsilon$
T	$T \rightarrow F\ T'$			$T \rightarrow F\ T'$	
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow \&\ F\ T'$		
F	$F \rightarrow id$			$F \rightarrow \neg\ F$	$T' \rightarrow \varepsilon$

Algoritmo do Analisador Preditivo Tabular:

Entrada: uma sentença **S** e a tabela de análise **M** para a gramática **G.**

Resultado: uma derivação mais à esquerda de **s,** se **s** está em **L(G)**, ou uma indicação de erro, caso contrário.

Método: inicialmente, o analisador está numa configuração na qual a pilha **$S** (com **S** no topo), e a fita de entrada contém **s$.** O programa utiliza a tabela de análise preditiva **M** e comporta-se do seguinte modo:

Movimentos de um analisador tabular preditivo

Pilha'	Entrada	Ação
$E	id ∨ id & id $	E → T E'
$E' T	id ∨ id & id $	E → F T'
$E' T' F	id ∨ id & id $	F → id
$E' T' id	id ∨ id & id $	desempilha e lê símbolo
$E' T'	∨ id & id $	T' → ε
$E'	∨ id & id $	E' → ∨ T E'
$E' T ∨	∨ id & id $	desempilha e lê símbolo
$E' T	id & id $	T → F T'
$E' T' F	id & id $	F → id
$E' T' id	id & id $	desempilha e lê símbolo
$E' T'	& id $	T' → & F T'
$E' T' F&	& id $	desempilha e lê símbolo
$E' T' F	id $	F → id
$E' T' id	id $	desempilha e lê símbolo
$E' T'	$	T' → ε
$E'	$	E' → ε
$	$	aceita a sentença

Posicione o cabeçote sobre o primeiro símbolo de **s$**;
Seja **X** o símbolo de topo da pilha e **a** o símbolo sob o cabeçote.
Repete

se X é um terminal
então se X = a
então desempilha X e avança o cabeçote
senão ERRO
senão
se M[X,a] = $X \rightarrow Y_1\ Y_2 \ldots Y_K$
então { desempilha X;
empilha $Y_K\ Y_{K-1} \ldots Y_1$ com Y_1 no topo;
imprime a produção $X \rightarrow Y_1\ Y_2 \ldots Y_K$
}
senão ERRO

Até que X = $

Funções FIRST e FOLLOW

- Determinação das funções FIRST e FOLLOW
- Conjuntos FIRST
  - Convém iniciar pelos não-terminais mais simples.

$$FIRST(F) = \{ \neg, id \}$$

$$FIRST(T') = \{ \&, \varepsilon \}$$

$$FIRST(E') = \{ \vee, \varepsilon \}$$

FIRST(T) = FIRST(F) ou seja, $FIRST(T) = \{ \neg, id \}$

Funções FIRST e FOLLOW

- Conjuntos FOLLOW

FOLLOW(E) = { $ }

FOLLOW(E’) = FOLLOW(E) ou seja,

FOLLOW(E’) = { $ }

FOLLOW(T) = { $\vee$, $ }

FOLLOW(T’) = FOLLOW(T) ou seja,

FOLLOW(T’) = { $\vee$, $ }

FOLLOW(F) é a união de FIRST(T’), FOLLOW(T) e FOLLOW(T’), ou seja,

FOLLOW(F) = { $\vee$, &, $ }

Funções FIRST e FOLLOW

- Algoritmo para construir uma tabela de análise preditiva:

Entrada: gramática G

Resultado: Tabela de Análise M

Método:

1. Para cada produção **A** $\rightarrow$ α de G, execute os passos 2 e 3.
2. Para cada terminal a de **FIRST(α)**, adicione a produção **A** $\rightarrow$ α a **M[A, a]**.
3. Se **FIRST(α)** inclui a palavra vazia, então **A** $\rightarrow$ α a **M[A, b]** para cada **b** em **FOLLOW(A)**.

Funções FIRST e FOLLOW

Para	Tem-se	
$E \rightarrow T\ E'$	FIRST(T E') = {$\neg$, id}	M[E, $\neg$] = M[E, id] = $E \rightarrow T\ E'$
$E' \rightarrow \vee\ T\ E'$	FIRST ($\vee$ T E') = { $\vee$ }	M[E', $\vee$] = $E \rightarrow \vee\ T\ E'$
$E' \rightarrow \varepsilon$	FOLLOW (E') = { $ }	M[E', $] = $E' \rightarrow \varepsilon$
$T \rightarrow F\ T'$	FIRST (F T') = { $\neg$, id }	M[T, $\neg$] = M[T, id] = $M \rightarrow F\ T'$
$T' \rightarrow \&\ F\ T'$	FIRST (& F T) = { & }	M[T', &] = $T' \rightarrow \&\ F\ T'$
$T' \rightarrow \varepsilon$	FOLLOW (T') = { $\vee$, $ }	M[T', $\vee$] = M[T', $] = $T' \rightarrow \varepsilon$
$F \rightarrow \neg\ F$	FIRST ($\neg$ F) = { $\neg$ }	M[F, $\neg$] = $F \rightarrow \neg\ F$
$F \rightarrow$ id	FIRST (id) = { id }	M[F, id] = $F \rightarrow$ id

Análise Sintática LL(1)

- Utiliza uma pilha explícita, em vez de ativações recursivas.
- Implementa um autômato de pilha controlado por uma tabela de análise.
- É rica o suficiente para reconhecer a maioria das construções presentes nas linguagens de programação.
- Nenhuma gramática com recursão à esquerda ou ambígua pode ser LL(1).

Análise Sintática LL(1)

- Uma gramática G é LL(1) se e somente se, sempre que $A \rightarrow \alpha \mid \beta$ forem duas produções distintas de G, as seguintes condições forem verdadeiras:
  1. Para um terminal ***a***, tanto α quanto β não derivam cadeias começando com ***a***.
  2. No máximo um dos dois, α ou β, pode derivar a cadeia vazia.
  3. Se $\beta \Rightarrow^* \in$, então α não deriva nenhuma cadeia começando com um terminal em FOLLOW(A). De modo semelhante, se $\alpha \Rightarrow^* \in$, então β não deriva qualquer cadeia começando com um terminal em FOLLOW(A).

Análise Sintática LL(1)

- Gramática simples que gera cadeias de parênteses balanceados:

$$S \rightarrow (S) S \mid \in$$

- Ações de análise sintática de um analisador descendente.

	Pilha de análise sintática	Entrada	Ação
1	$ S	( ) $	$S \rightarrow (S) S$
2	$ S) S (	( ) $	casamento
3	$ S) S	) $	$S \rightarrow \in$
4	$ S)	) $	casamento
5	$ S	$	$S \rightarrow \in$
6	$	$	aceita

Análise Sintática LL(1)

- A primeira coluna enumera os passos.
- A segunda mostra o conteúdo da pilha de análise sintática, com o final da pilha à esquerda e o topo da pilha à direita.
- O final da pilha é indicado com um cifrão.
- A terceira mostra a entrada (da esquerda para a direita).
- Um cifrão marca o final da entrada (marca de final de arquivo – EOF).
- A quarta coluna apresenta uma descrição resumida da ação do analisador sintática.

Análise Sintática LL(1)

- Começa pela colocação do símbolo de início (**S**)na pilha.
- Ele aceita uma cadeia de entrada (**( )**) se, após uma série de ações, a pilha e a entrada ficarem vazias.
- As ações básicas de um analisador descendente são:
  a) Substituir um não-terminal A no topo da pilha por uma cadeia α com base na escolha da regra gramatical $A \rightarrow \alpha$;
  b) Casar uma marca no topo da pilha com a marca de entrada seguinte.

Análise Sintática LL(1)

- Tabela sintática para uma gramática ambígua:

Seja G a gramática abaixo:

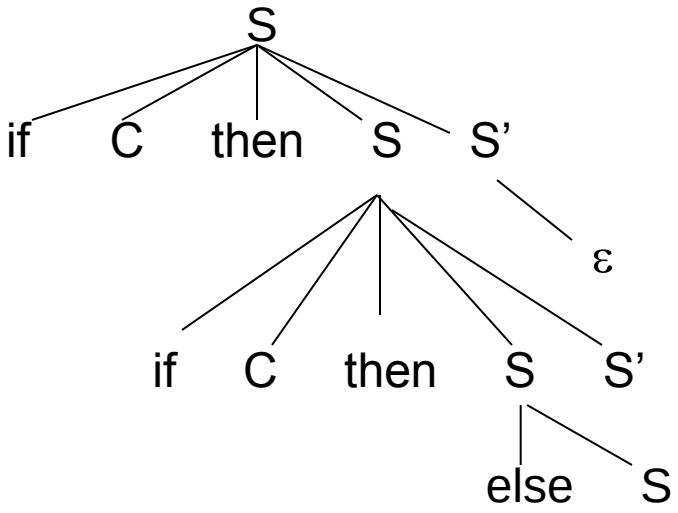
$S \rightarrow$ if C then S S' | a

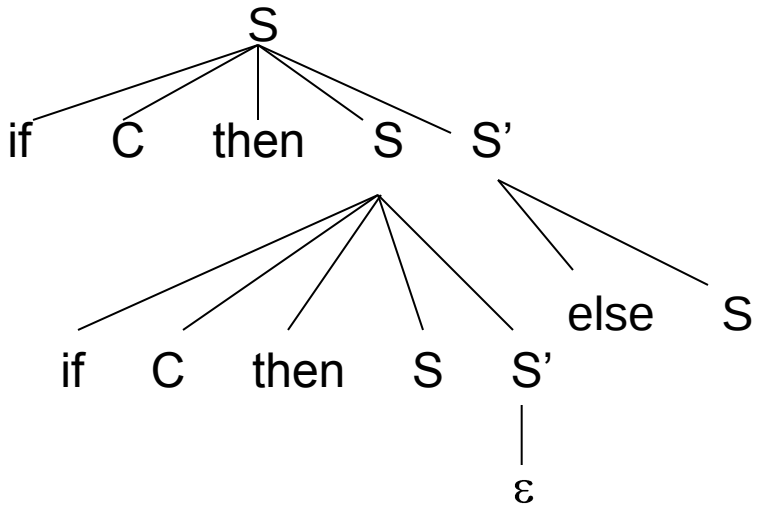
$S` \rightarrow$ else S | ε

$C \rightarrow b$

Análise Sintática LL(1)

- Árvore de derivação





A gramática G é ambigua, permite duas árvores de derivação distintas para a sentença

if b then if b then a else a.

Análise Sintática LL(1)

- Tabela para a gramática G:

	a	b	else	if	then	$
S	$S \rightarrow a$			$S \rightarrow$ if C then S S'		
S'			$S' \rightarrow \varepsilon$ $S' \rightarrow$ else S			$S' \rightarrow \varepsilon$
C		$C \rightarrow b$				

Quando **S'** estiver no topo da pilha e **else** sob o cabeçote de leitura, o analisador terá duas opções:
1)Apenas desempilhar **S'** (reconhecimento do comando **if-then**)
2)Desempilhar **S'** e empilhar **else S** (reconhecimento de **if-then-else**)

Análise Sintática LL(1)

PILHA	ENTRADA	DERIVAÇÃO
$ S	if b then if b then a else a$	S → if C then S S’
$ S’ S then C if	if b then if b then a else a$	desempilha
$ S’ S then C	b then if b then a else a$	C → b
$ S’ S then b	b then if b then a else a$	
$ S’ S then	then if b then a else a$	desempilha
$ S’ S	if b then a else a$	S → if C then S S’
$ S’ S’ S then C if	if b then a else a$	desempilha
$ S’ S’ S then C	b then a else a$	C → b
$ S’ S’ S then b	b then a else a$	desempilha
$ S’ S’ S then	then a else a$	
$ S’ S’ S	a else a$	S → a
$ S’ S’ a	a else a$	desempilha
$ S’ S’	else a$	S’ → else S
$ S’ S’ S else	else a$	desempilha
$ S’ S’ S	a$	S → a
$ S’ a	a$	desempilha
$ S’	$	S’ → ε
$	$	Aceita!

Análise Sintática LL(1)

- Para a gramática da expressão

E	$\rightarrow$	**T E'**
E'	$\rightarrow$	**+ T E' | $\in$**
T	$\rightarrow$	**F T'**
T'	$\rightarrow$	*** F T' | $\in$**
F	$\rightarrow$	**(E) | id**

- o algoritmo abaixo produz a tabela de análise

ENTRADA: Gramática G.

SAÍDA: Tabela de análise M.

MÉTODO: Para a produção $A \rightarrow \alpha$ da gramática, faça:

1. Para cada terminal *a* em FIRST(A), inclua $A \rightarrow \alpha$ em M[A,a].
2. Se ε pertence a FIRST(α), inclua $A \rightarrow \alpha$ em M[A,b] para cada terminal b em FOLLOW(A). Se ε pertence a FIRST(α) e $ pertence a FOLLOW(A), acrescente também $A \rightarrow \alpha$ em M[A,$].

Análise Sintática LL(1)

- Considere a produção E $\rightarrow$ T E', visto que:

  FIRST(TE') = FIRST(T) = {(, id)}

  essa produção é incluída em M[E,(] e M[E,id].

  A produção E' $\rightarrow$ + T E' é incluída em M[E',+], desde que FIRST(+ T E') = {+}.

  Visto que o FOLLOW(E') = {),$}, a produção E' $\rightarrow$ ε é incluída em M[E',)] e em M[E',$].

Análise Sintática LL(1)

NÃO TERMINAL	Símbolo de Entrada					
	id	+	*	(	)	$
E	$E \rightarrow T\ E'$			$E \rightarrow T\ E'$		
E'		$E' \rightarrow +\ T\ E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F\ T'$			$T \rightarrow F\ T'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *\ F\ T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow$ id			$F \rightarrow (E)$		

- O algoritmo apresentado pode ser aplicado a qualquer gramática G para produzir a tabela M de análise correspondente a G.
- Para toda gramática LL(1), cada entrada na tabela de análise identifica no máximo uma produção ou sinaliza um erro.

Análise Sintática LL(1)

- Algoritmo de análise sintática LL(1) baseado em tabela.

while topo da pilha for $\neq$ *$* **and** próxima marca for $\neq$ $ **do**
  if topo da pilha for o terminal *a* **and** próxima marca de entrada for = *a*
  then (casamento)
    retira da pilha;
    avança entrada;
  else if topo da pilha for um não-terminal *A*
    and próxima marca de entrada for terminal *a*
    and célula da tabela *M[A,a]* contiver a produção
      $A \rightarrow X_1 X_2 ... X_n$
  then (gera)
    retira da pilha;
    for i:= n **downto** 1 **do** coloca X_i na pilha;
  else erro;
if topo da pilha for = *$* **and** marca seguinte na entrada for = *$*
then aceita;
else erro;

Remoção de Recursão à Esquerda

É utilizada mais comumente para operações associativas à esquerda, como na gramática de expressões simples, na qual

$$E \rightarrow E\ op\ T \mid T$$

torna as expressões representadas por *op* associativas à esquerda.

Remoção de Recursão à Esquerda

- **Caso 1:** recursão imediata à esquerda simples
  - Considere a regra recursiva à esquerda para a gramática de expressão simples

$$E \rightarrow E\ op\ T \mid T$$

  - Ela tem a forma $A \rightarrow A\alpha|\beta$, em que A = *E*, α = *op* e β = *T*.
  - A remoção da recursão à esquerda produz:

$$\mathbf{E \rightarrow T\ E'}$$

$$\mathbf{E' \rightarrow op\ T\ E' \mid \varepsilon}$$

Remoção de Recursão à Esquerda

- **Caso 2:** recursão imediata à esquerda geral
  - Considere a regra gramatical

$$E \rightarrow E + T \mid E - T \mid T$$

  - Removemos a recursão à esquerda assim:

$$\mathbf{E \rightarrow T\ E'}$$

$$\mathbf{E' \rightarrow + T\ E' \mid - T\ E' \mid \varepsilon}$$

Remoção de Recursão à Esquerda

- **Caso 3:** recursão à esquerda geral
  - Considere a gramática artificial a seguir:

$$A \rightarrow B\ a \mid A\ a \mid c$$

$$B \rightarrow B\ b \mid A\ b \mid d$$

  - A gramática resultante é:

$$\boldsymbol{A \rightarrow B\ a\ A' \mid c\ A'}$$

$$\boldsymbol{A' \rightarrow a\ A' \mid \varepsilon}$$

$$\boldsymbol{B \rightarrow B\ b \mid A\ b \mid d}$$

• Eliminamos a regra B $\rightarrow$ A pela substituição de A por suas escolhas na primeira regra.

- Assim, obtemos a gramática

A $\rightarrow$ B a A' | c A'
A' $\rightarrow$ a A' | ε
B $\rightarrow$ B b | B a A' b | c A' b | d

• Removemos a recursão imediata à esquerda de B para obter:

A $\rightarrow$ B a A' | c A'
A' $\rightarrow$ a A' | ε
B $\rightarrow$ c A' b B' | d B'
B' $\rightarrow$ b B' | a A' b B' | ε

– Essa gramática não tem recursões à esquerda.

Recuperação de Erros

- É um fator crítico em um compilador.
- Precisa determinar se um programa está ou não sintaticamente correto (reconhecedor).
  - Reconhece as cadeias da linguagem livre de contexto geradas pela gramática da linguagem de programação.
- Além de ter o comportamento de um reconhecedor, pode apresentar diversos níveis de respostas a erros.
  - Tentar dar uma resposta significativa de erro para o primeiro erro encontrado.
  - Tentar determinar o local onde ocorreu o erro.
  - Tentar alguma forma de correção de erros (reparo de erros).

Recuperação de Erros

- Considerações importantes:
  1. A ocorrência de um erro deve ser determinada tão logo quanto possível.
  2. O analisador deve sempre tentar analisar o máximo possível de código, para que o máximo de erros possa ser identificado.
  3. Deve evitar o problema de cascata de erros, onde um erro gera uma sequência de mensagens espúrias.
  4. Evitar laços infinitos de erros.

Recuperação de Erros na Análise LL

- Na tabela LL, as lacunas representam situações de erro e devem ser usadas para chamar rotinas de erros de recuperação.
- Um erro é detectado durante o reconhecimento preditivo quando o terminal no topo da pilha não casa com o próximo símbolo de entrada,
- ou quando o não-terminal A está no topo da pilha, a é o próximo símbolo da entrada, e M[A,a] é uma entrada de erro (a entrada na tabela de análise está vazia).

Recuperação de Erros na Análise LL

- Pode-se alterar a tabela de análise para recuperar erros segundo dois modos distintos:
  - Modo pânico: na ocorrência de um erro, o analisador despreza símbolos da entrada até encontrar um token de sincronização.
  - Recuperação local: o analisador tenta recuperar o erro, fazendo alterações sobre um símbolo apenas:
    - desprezando o token da entrada, ou
    - substituindo-o por outro, ou
    - inserindo um novo token, ou ainda,
    - removendo um símbolo da pilha.

Modo Pânico

- Baseia-se na idéia de ignorar símbolos da entrada até encontrar um token no conjunto de tokens de sincronismo.
- Sua eficácia depende da escolha do conjunto de sincronismo.
- Os conjuntos devem ser escolhidos de modo que o analisador se recupere rapidamente dos erros que ocorrem na prática.

Modo Pânico

- Quando bem implementado pode ser um método muito bom para recuperação de erros.
- Tem a vantagem adicional de virtualmente garantir que o analisador sintático não entre em laço infinito durante a recuperação de erros.
- Associa a cada procedimento recursivo um parâmetro adicional composto por um conjunto de marcas de sincronização.

Modo Pânico

- As marcas que podem ser de sincronização são acrescentadas a esse conjunto cada vez que ocorre uma ativação.
- Ao encontrar um erro, o analisador varre à frente, descartando as marcas até que um dos conjuntos sincronizados seja encontrado na entrada, encerrando a análise.

Modo Pânico

1. Inclua todos os símbolos do FOLLOW(A) no conjunto de sincronização para o não-terminal A.
2. Acrescentar ao conjunto de sincronização de uma construção de nível inferior os símbolos que iniciam construções de nível superior.
3. Incluir os símbolos em FIRST(A) no conjunto de sincronização do não-terminal A (pode retornar a análise de acordo com A se um símbolo em FIRST(A) aparecer na entrada).
4. Se um não-terminal gerar a cadeia vazia, pode adiar a detecção de erro, mas não faz com que o erro se perca.
5. Se um terminal no topo da pilha não casar com o terminal da entrada, desempilha o terminal.

Modo Pânico

- Pseudocódigo para recuperação de erros:

procedure *varrepara*(*conjsincr*);
begin
  while not (*marca* **in** *conjsincr* ∪ {$}) **do**
    capturaMarca;
end *varredura*;

procedure *verificaentrada*(*conjprimeiro*, *conjsequencia*);
begin
  if not (*marca* **in** *conjprimeiro*) *then*
    error;
    varrepara (*conjprimeiro* ∪ *conjsequencia*);
  end if;
end *verificaentrada*;

Modo Pânico

- Pseudocódigo para recuperação de erros na calculadora descendente recursiva:

```
procedure exp (conjsincr);
begin
  verificaentrada({(,número}, conjsincr);
  if not (marca in conjsincr) then
    termo (conjsincr);
    while marca = + or marca = - do
      casamento (marca);
      termo (conjsincr);
    end while;
    verificaentrada (conjsincr, {(,número});
  end if;
end exp;
```

Modo Pânico

- Pseudocódigo para recuperação de erros na calculadora descendente recursiva:

```
procedure fator (conjsincr);
begin
  verificaentrada ({(,número}, conjsincr);
  if not (marca in conjsincr) then
    case marca of
    ( : casamento (();
      exp ({)});
      casamento ());
    número :
      casamento (número);
    else error;
    end case;
    verificaentrada (conjsincr, {(,número});
  end if;
end fator;
```

Tokens de sincronização

Não Terminal	Símbolo de Entrada					
	id	+	*	(	)	$
E	$E \rightarrow T\ E'$			$E \rightarrow T\ E'$	sinc	sinc
E'		$E \rightarrow T\ E'$			$E \rightarrow \in$	$E \rightarrow \in$
T	$T \rightarrow F\ T'$	sinc		$T \rightarrow F\ T'$	sinc	sinc
T'		$T' \rightarrow \in$	$T' \rightarrow * F\ T'$		$T' \rightarrow \in$	$T' \rightarrow \in$
F	$F \rightarrow id$	sinc	sinc	$F \rightarrow (E)$	sinc	sinc