# Disciplina: Compiladores
# Análise Sintática

Prof. Flávio Márcio de Moraes e Silva

# Top-down parsing

- Um método top-down pode ser visto como aquele que:
  - procura uma derivação mais a esquerda para o string de entrada;
  - constrói a árvore de parse a partir da raiz em pré-ordem.
- Analisadores sintáticos descendentes:
  - Recursivo com retrocesso (backtracking);
  - Recursivo preditivo;
  - Tabular preditivo.

# Análise recursiva com retrocesso

- Derivação sempre expandindo o não-terminal mais a esquerda;
- Quando existe mais de uma regra de produção para um não-terminal, a decisão é em função do símbolo (token) presente na entrada; (Ex: S -> aA | bA | cA, 1º token a, b ou c)
- Se o token na entrada não define univocamente a escolha, então todas alternativas serão tentadas até que se tenha sucesso ou até alcançar uma falha irremediável. Exemplo:
- S -> cAd
- A-> ab | a
- Suponha a string w = cad
- 1ª solução: S => cAd => cabd (backtracking)
- 2ª solução: S => cAd => cad (Sucesso!!!)

# Análise recursiva com retrocesso

- Implementando: crie um procedimento para cada não terminal.

- bool S( ) {
- if ( lookahead == ‘c’ ) {
- match(‘c’); A( ); match(‘d’); return true;
- } else return false;
- }

# Análise recursiva com retrocesso

```
bool A( ) {
// guarda a posição da entrada ( vetor )
int guarda = lookahead;
        if  ( lookahead  == ‘a’) {
                match(‘a’);
                if (lookahead  == ‘b’) {
                        match(‘b’);  return true; }
        } else return false;
lookahead = guarda;
        if  ( lookahead == ‘a’) {
                match(‘a’);
                return true;
        } else return false;
//  outras produções
//  erro no reconhecimento do string
}
```

# Análise recursiva com retrocesso

- Desvantagens:
  - O processo de voltar atrás no reconhecimento e tentar produções alternavas é ineficiente. Repetição da leitura de partes da sentença na entrada.
  - Como o reconhecimento normalmente é acompanhado de ações semânticas (ex: atualizar a tabela de símbolos), a ocorrência do retrocesso pode levar o analisador a desfazer tais ações.
  - Quando erros ocorrem fica difícil identificar com precisão o local do erro, devido à tentativa de aplicação produções alternativas.

# Análise recursiva preditiva

- Podemos eliminar a necessidade de backtracking:
  - eliminando recursividade a esquerda (FN Greibach)
  - fatorando a gramática a esquerda
- Para construir um Predictive Parser devemos saber:
  - dado o símbolo corrente na entrada "**a**" e o não terminal "*A*" *a* ser expandido
  - A -> a1 | a2 | ... | a3
  - qual a única produção a ser usada que deriva o string começando em "**a**"**.**

# Análise recursiva preditiva

- Idéia: determinar a alternativa (produção) a ser usada olhando apenas o 1º símbolo que ela deriva
- Exemplo:
- *stmt* -> **if** *expr* **then** *stmt* **else** *stmt*
- | **while** *expr* **do** *stmt*
- | **begin** *stmt_list* **end**

- Se a token na entrada é **if**: produção 1
- Se a token na entrada é **while**: produção 2
- Se a token na entrada é **begin**: produção 3

# Conjunto FIRST()

- Os terminais que iniciam sentenças deriváveis a partir de uma forma sentencial ***w*** constituem o conjunto **FIRST(*w*)**
- Regras do conjunto FIRST de uma forma sentencial **w**:
  - Regra 1: $w \Rightarrow^{*} a$ , então “a” é um elemento de FIRST(w)
  - Regra 2: $w \Rightarrow^{*} av$ , então “a” é um elemento de FIRST(w), sendo “a” um símbolo terminal e “v” uma forma sentencial qualquer, podendo ser vazia (regra 1).
- Exemplo slide anterior, conjunto FIRST(stmt) = {if, while, begin}
- Dado um símbolo não terminal A que possui várias produções (Ex: A -> B1|B2|...|Bn). Para se implementar um reconhecedor recursivo preditivo para A, exige-se que os conjuntos FIRST de B1, B2, ... , Bn sejam disjuntos dois a dois. Um terminal não pode aparecer em 2 conjuntos.

# Diagrama de Transição

- Auxilia a elaboração de um Predictive Parsing.
- Possibilita a simplificação do programa (parser) resultante.
- Para um analisador:
  - faça um diagrama para cada não-terminal;
  - rotule as arestas com tokens e não-terminais;
  - uma token rotulando uma aresta significa transição se a mesma for o próximo símbolo na entrada;
  - um não-terminal rotulando uma aresta significa a chamada ao procedimento para o não-terminal.

- Obs: após a criação do diagrama os procedimentos para cada um dos não-terminais terão que ser criados de forma similar ao que foi feito para a análise com retrocesso.

# Diagrama de Transição

- Construção:
- elimine a recursividade a esquerda (FN Greibach);
- fatore a gramática a esquerda;
- para cada não-terminal faça:
  - crie um estado inicial e final
  - para cada produção A -> $X_1X_2 \ldots X_n$ crie um caminho do estado inicial para o final rotulado por $X_1, X_2, \ldots, X_n$

- Note que:
  - o parser só avança a entrada ao encontrar um label com um terminal;
  - se existir uma aresta de S para T rotulada por “ε” , então do estado S o parser muda para o estado T, sem avançar a entrada.

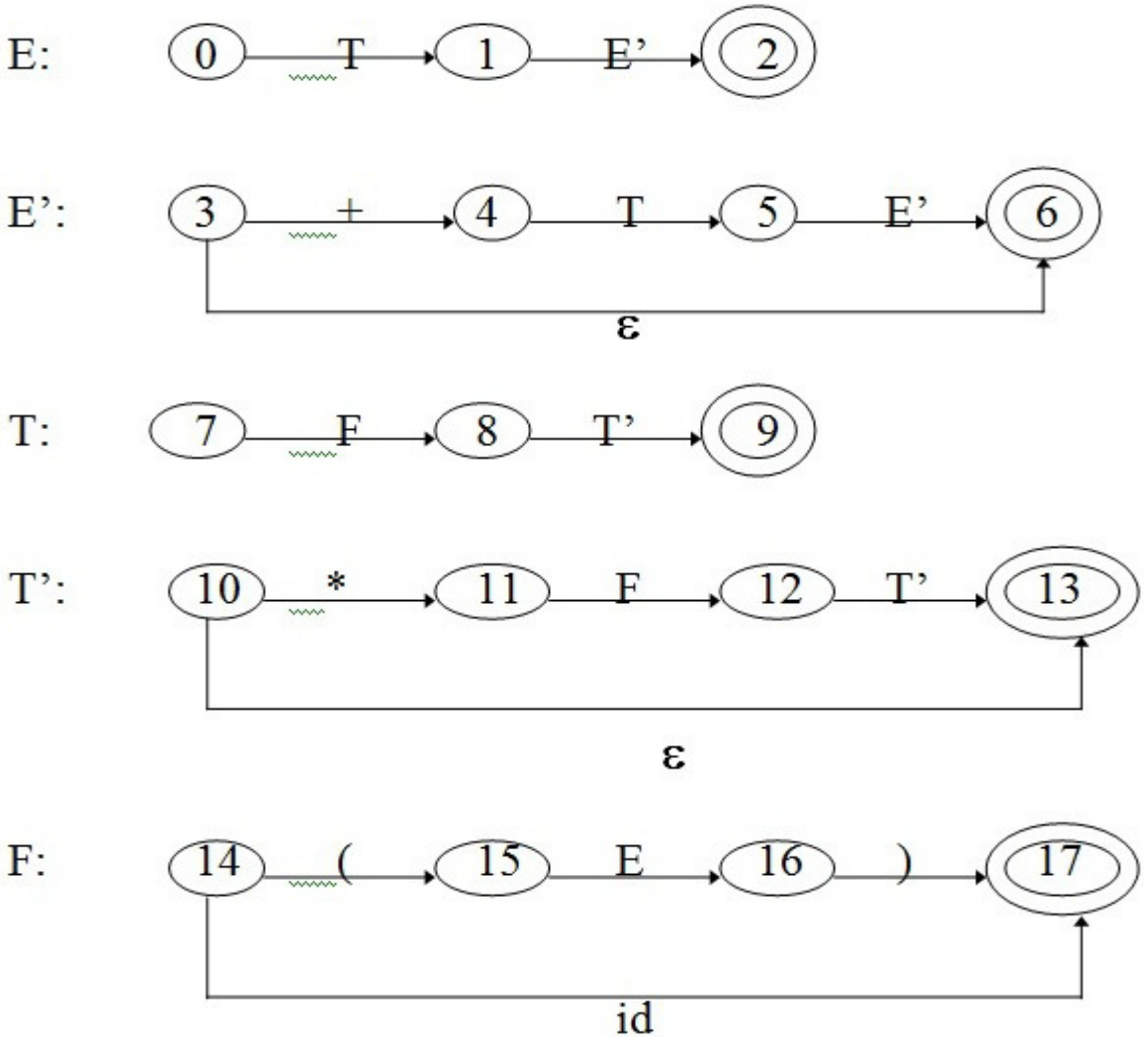
# Exemplo: Construção do Diagrama

- Exemplo:
- *E* -> *E* + *T* | *T*
- *T* -> *T* * *F* | *F*
- *F* -> *( E )* | **id**

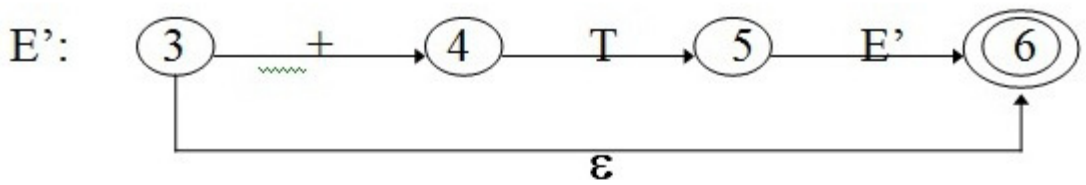
- eliminando a recursividade a esquerda e fatorando:

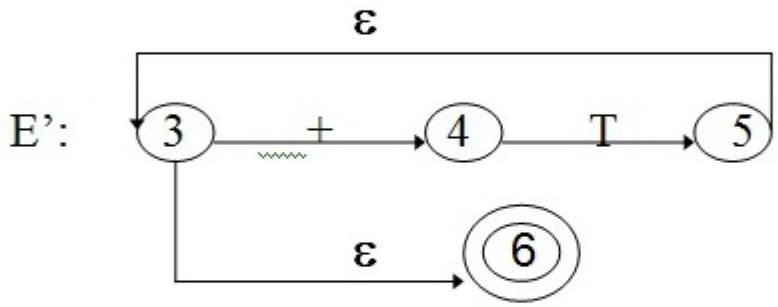
- *E* -> *TE'*
- *E'* -> *+TE'* | ε
- *T* -> *FT'*
- *T'* -> **FT'* | ε
- *F* -> *( E )* | **id**

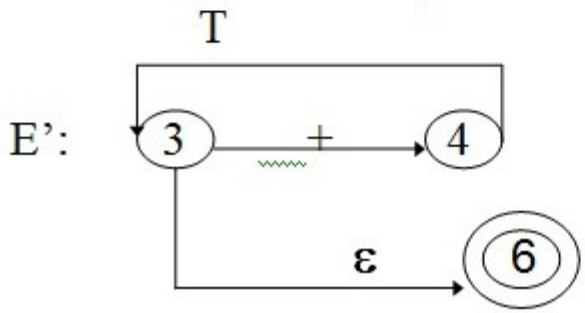
# Diagrama de Transição



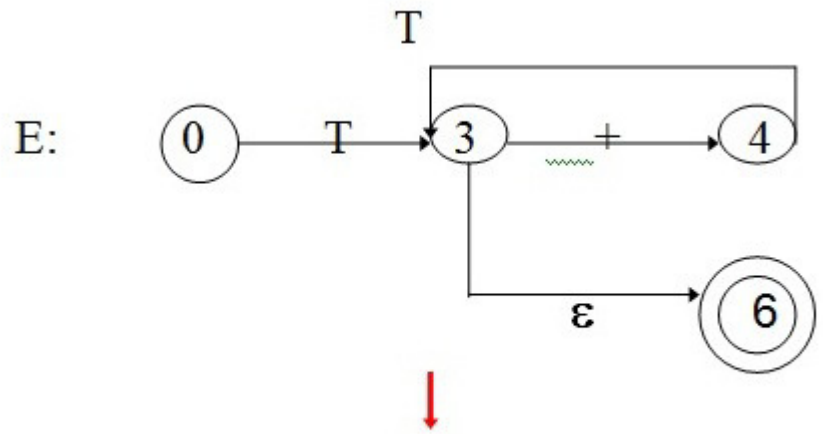
# Simplificando



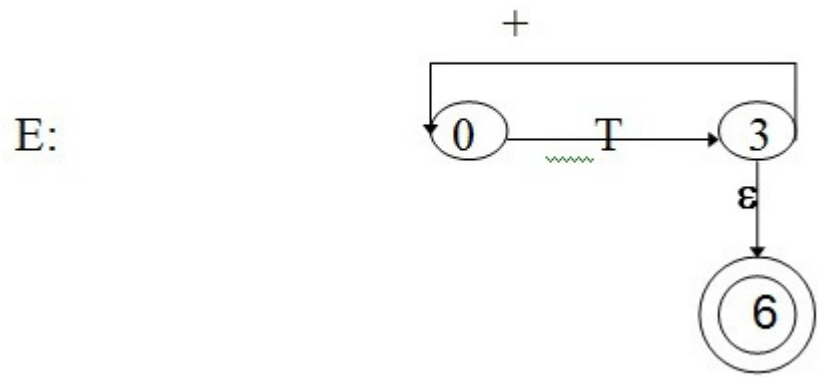



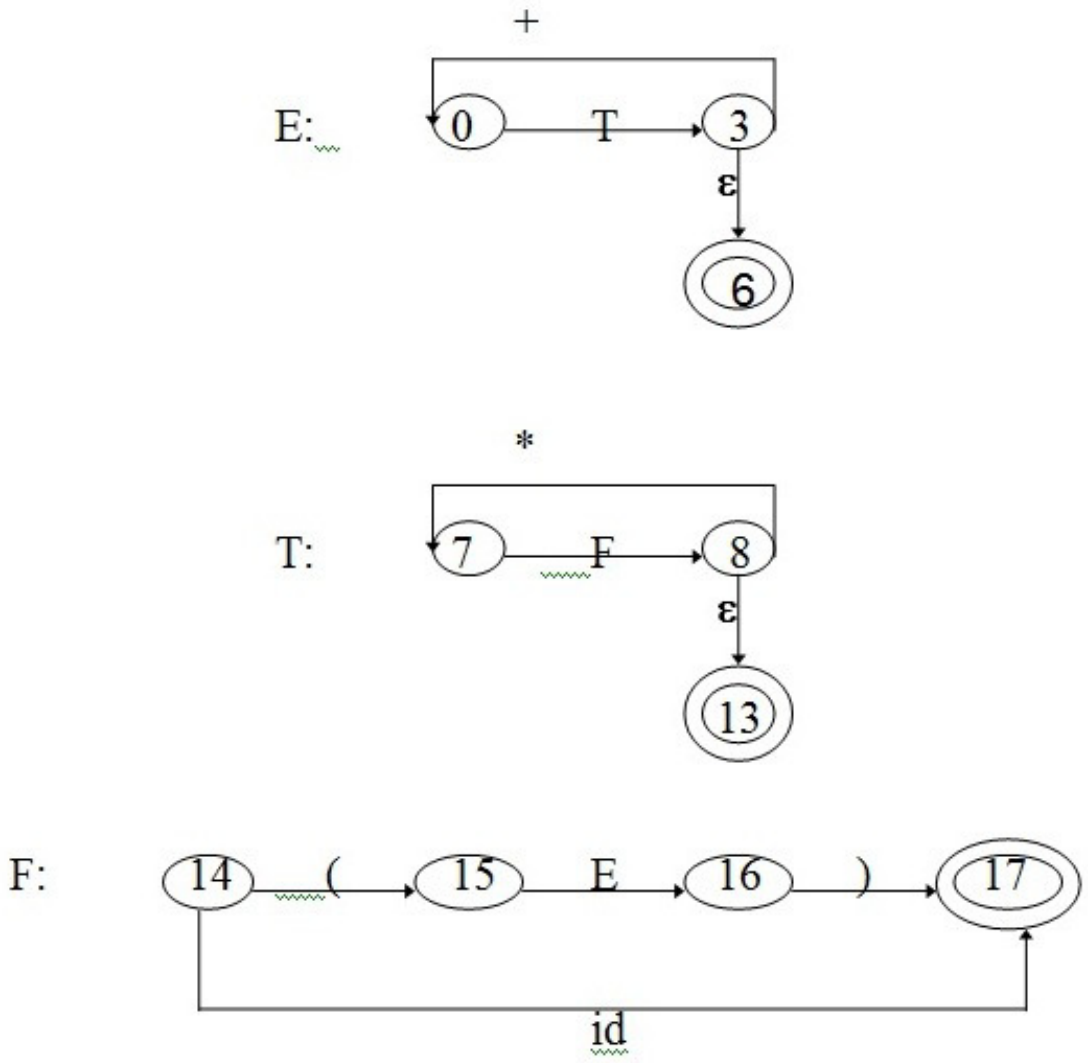
# Substituindo E’ em E




como o estado 0 e 4 são equivalentes:



# Finalmente



# Análise tabular preditiva

- É possível implementar um predictive parsing não recursivo, mantendo-se a pilha explicitamente.
- O parser consulta uma tabela de parsing para determinar a produção a ser usada.
- Componentes:
  - buffer de entrada: guarda o string de tokens ( o fim do string é indicado por $ );
  - pilha: armazena uma sequência de símbolos da gramática com $ marcando o fundo da pilha;
  - tabela de parsing:
    - arranjo bidimensional M[ A, a ];
    - armazena as produções a serem usadas;
    - guia as ações do parser.

# Análise tabular preditiva

- Considere X o símbolo no topo da pilha e "**a**" uma token na entrada. Estes dois símbolos determinam as ações do parser:
  - se X = a = $, o parser para (reconheceu-se o string na entrada),
  - se X = a ≠ $, o parser desempilha X avançando um símbolo na entrada,
  - se X é um não-terminal o parser consulta a entrada M[X,a]. Esta entrada será uma X-produção ou uma entrada de erro:
    - se M[X,a] = { X -> U V W} o parser desempilha X empilhando W V U
    - se M[X,a] = erro o parser chama uma rotina de recuperação de erros

# Análise tabular preditiva

- Predictive Parsing não-recursivo:
- *Entrada:* um string “w” e a tabela de parsing M para a gramática G.
- *Saída:* se “w” está em L(G), a derivação mais a esquerda para “w”; caso contrário uma indicação de erro.
- *Inicialmente*: a pilha contem $S e a entrada w$.

- A seguir um algoritmo que dita o funcionamento do parser:

# Análise tabular preditiva

- ip aponta para o 1º símbolo de w$
- repita
- seja X o topo da pilha e “a” o símbolo apontado por ip
- se X for um terminal então
- se X = a então
- desempilhe X e avance ip
- senão erro( )
- senão
- se M[X, a] = X -> $Y_1$ $Y_2$ ... $Y_k$ então início
- desempilhe X;
- empilhe $Y_k$ $Y_{k-1}$ ... $Y_1$, com $Y_1$ no topo
- imprima a produção X -> $Y_1$ $Y_2$ ... $Y_k$
- fim
- senão erro( );
- até que X = a = $ /* pilha ficou vazia */

# Exemplo: Análise tabular preditiva

- Gramática:
- *E* -> *TE'*
- *E'* -> *+TE'* | ε
- *T* -> *FT'*
- *T'* -> **FT'* | ε
- *F* -> *( E )* | **id**
- Considere o string **id + id * id** e a tabela de parsing:

| | id | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| *E* | ***E->TE'*** | | | ***E->TE'*** | | |
| *E'* | | ***E'->+TE'*** | | | ***E'->ε*** | ***E'->ε*** |
| *T* | ***T->FT'*** | | | ***T->FT'*** | | |
| *T'* | | ***T'->ε*** | ***T'->*FT'*** | | ***T'->ε*** | ***T'->ε*** |
| *F* | ***F->id*** | | | ***F->(E)*** | | |

| PILHA | ENTRADA | SAÍDA |
|---|---|---|
| $E | id + id * id$ | |
| $E’T | id + id * id$ | E -> TE’ |
| $E’T’F | id + id * id$ | T -> FT’ |
| $E’T’id | id + id * id$ | F -> id |
| $E’T’ | + id * id$ | |
| $E’ | + id * id$ | T’ -> ε |
| $E’T+ | + id * id$ | E’ -> +TE’ |
| $E’T | id * id$ | |
| $E’T’F | id * id$ | T -> FT’ |
| $E’T’id | id * id$ | F -> id |
| $E’T’ | * id$ | |
| $E’T’F* | * id$ | T’ -> *FT’ |
| $E’T’F | id$ | |
| $E’T’id | id$ | F -> id |
| $E’T’ | $ | |
| $E’ | $ | T’ -> ε |
| $ | $ | E’ -> ε |

# First() e Follow()

- São funções que permitem determinar as entradas para uma tabela de parsing.
- Se **w** é uma forma sentencial de G, FIRST(**w**) é o conjunto de terminais que iniciam strings derivados de **w** :
- Se **w** =>* ε  então ε está em FIRST(**w**)
-
- Se *A* é um não-terminal de G, FOLLOW(*A*) é o conjunto de terminais que seguem imediatamente *A* em alguma forma sentencial.

# Calculando FIRST(w)

- Aplicar as seguintes regras até que nenhum terminal ou ε possa ser acrescentado ao conjunto:
  - Se X é um terminal, então FIRST(X) é { X }
  - Se X -> ε, acrescente ε a FIRST(X)
  - Se X -> $Y_1$ $Y_2$ ... $Y_k$ acrescente:
    - "**a**" a FIRST(X), se para algum i "**a**" está em FIRST( $Y_i$ ) e ε está em todos os FIRST($Y_1$) ... FIRST($Y_{i-1}$)
    - Se ε está em todos os FIRST($Y_m$) para m = 1, 2 .. k acrescente ε a FIRST(X).
- Obs: Se X -> $Y_1$ $Y_2$ ... $Y_k$ então tudo que está FIRST($Y_1$) está em FIRST(X). Se $Y_1$ -> ε, então acrescentamos FIRST($Y_2$) a FIRST(X) e assim sucessivamente.

# Calculando FOLLOW(A)

- Aplicar as seguintes regras até que nenhum terminal ou $ possa ser acrescentado ao conjunto:
  - Acrescente $ a FOLLOW(S), onde S é o símbolo inicial da gramática.
  - Sendo “w” e “y” formas sentenciais:
    - Se existir uma produção *A* -> w*C*y, então acrescente a FOLLOW(*C*) tudo que estiver em FIRST(y) exceto ε.
    - Se existir uma produção *A* -> w*C,* ou uma produção *A* -> w*C*y onde FIRST(y) contem ε**,** então tudo que está em FOLLOW(*A*) esta em FOLLOW(*C*).

# Exemplo FIRST() e FOLLOW()

- *E -> TE'*
- *E' -> +TE' | ε*
- *T -> FT'*
- *T' -> *FT' | ε*
- *F -> ( E ) | **id***

- First(E) = First(T) = First(F) = { (, id }
- First(E') = { +, ε }
- First(T') = { *, ε }

- Follow(E) = Follow(E') = { ), $ }
- Follow(T) = Follow(T') = { +, ), $ }
- Follow(F) = { +, *, ), $ }

# Algoritmo – Construção da Tabela

- *Entrada*: Gramática G.
- *Saída:* Tabela de Parsing.

- 1) Para cada produção A -> **w** da gramática, faça passos 2, 3 e 4
- 2) Para cada terminal **a** em First(**w**), acrescente A -> **w** a M[A,**a**]
- 3) Se ε está em First(**w**) acrescente A -> **w** a M[A,**b**] para cada terminal **b** em Follow(A).
- 4) Se ε está em First(**w**) e $ em Follow(A) acrescente A -> **w** a M[A,$]
- 5) Considere as entrada restantes (vazias) erros.

- **Como exercício refaça a tabela anterior**

# Exemplo: Análise tabular preditiva

- Gramática:
- *E* -> *TE'*
- *E'* -> *+TE'* | ε
- *T* -> *FT'*
- *T'* -> **FT'* | ε
- *F* -> *( E )* | **id**
- Considere o string **id + id * id** e a tabela de parsing:

| | id | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| *E* | ***E->TE'*** | | | ***E->TE'*** | | |
| *E'* | | ***E'->+TE'*** | | | ***E'->ε*** | ***E'->ε*** |
| *T* | ***T->FT'*** | | | ***T->FT'*** | | |
| *T'* | | ***T'->ε*** | ***T'->*FT'*** | | ***T'->ε*** | ***T'->ε*** |
| *F* | ***F->id*** | | | ***F->(E)*** | | |

# Gramáticas LL(1)

- O algoritmo anterior pode ser aplicado a qualquer gramática G. Mas para algumas gramáticas a tabela conterá múltiplas entradas em uma única posição M[A,a], se isto acontecer, G é recursiva a esquerda ou ambígua. Exemplo:

- S -> iEtSS' | a
- S' -> eS | ε
- E -> b

| | a | b | e | i | t | $ |
|---|---|---|---|---|---|---|
| *S* | ***S->a*** | | | ***S->iEtSS'*** | | |
| *S'* | | | ***S'->ε***<br>***S'->eS*** | | | ***S'->ε*** |
| *E* | | ***E->b*** | | | | |

# Gramáticas LL(1)

- Note que: M[S', e] = S'-> eS e S'-> ε (Follow(S') = {e, $})
- **ou seja a gramática é ambígua.** Podemos eliminar a ambiguidade escolhendo S'-> eS.
- Uma gramática com tabela de parsing sem múltipla entrada é dita LL(1). Nenhuma gramática ambígua ou recursiva a esquerda é LL(1).
- LL(1): (Left to right) análise da sentença da esquerda para a direita, (Leftmost derivation) produzindo uma derivação mais a esquerda, (1) levando em conta apenas 1 símbolo na entrada.
- Uma gramática G é LL(1) se dado **A -> w | y** então:
  - First (w) ∩ First (y) = { } (conjunto vazio);
  - Se **w =>*** ε , ou seja First(w) contém ε, então **y** não deriva string começando com um terminal em Follow(A);
  - No máximo **w** ou **y** deriva o string vazio.

# Gramáticas LL(1)

- **O que fazer quando a tabela tem múltiplas entradas?**
  - 1) Transformar a gramática para a forma normal de Greibach e fatora-la à esquerda: torna G difícil de ler e traduzir. Pode não resolver se a linguagem gerada pela gramática for inerentemente ambígua.
  - 2) Eliminar a produção (**na tabela**) que torna a entrada multivalorada: não existe uma regra universal sem afetar a linguagem reconhecida.