Parsing Preditivo

Antes de ser abordado o *Parsing* Preditivo, será apresentado o Analisador Sintático Descendente Recursivo.

• Analisador Sintático Descendente Recursivo (ASDR)

→ O analisador sintático descendente recursivo é escrito na forma de um conjunto de procedimentos, um procedimento para cada elemento não-terminal da gramática (sem retrocesso).

Exemplo: Sejam as seguintes produções:

<cmd> → **begin** <lista_cmds> **end**
| **while** <condição> **do** <cmd>
| **if** <condição> **then** <cmd>

As palavras_chave **begin**, **while** e **if** indicam qual das alternativas será a única possibilidade de se encontrar um dado comando.

Limitações do ASDR:

- Entra em ciclo (*loop*) para gramáticas recursivas à esquerda;
- Não lida com regras não-determinísticas(A::=αβ e A::=αγ);
- Na implementação requer uma linguagem com recursividade.

- **Problema de recursividade**

→ Para que se possa construir um analisador sintático descendente, uma gramática não pode ter regras recursivas à esquerda (diretas ou indiretas).

→ Uma gramática é recursiva à esquerda se tem produções da forma:

$$U ::= U\alpha \quad \text{ou} \quad U \Rightarrow^{+} U\alpha$$

Nesse caso, qualquer algoritmo que implemente um analisador descendente vai entrar em ciclo ("loop") infinito.

→ **Eliminação da recursividade à esquerda**

Seja a seguinte gramática G(A)

$A ::= A\alpha \mid \beta$

$L(G(A)) = \{\beta\alpha^n \mid n = 0, 1, \ldots\}$

A recursividade pode ser eliminada substituindo-se as produções de G(A) por:

$A ::= \beta A'$

$A' ::= \alpha A' \mid \varepsilon$ é equivalente a $A ::= \beta\{\alpha\}^*$

Exemplo:

G(E) – recursiva (direta) à esquerda	G'(E) – não recursiva à esquerda
E ::= E+ T| T	E ::= T E' E' ::= + T E' | ε
T ::= T* F| F	T ::= F T' T' ::= * F T' | ε
F ::= ID| (E)	F ::= ID | (E)

OBS: Na realidade, a recursividade foi transferida para a direita.

Em geral, a recursividade direta à esquerda pode ser eliminada como segue:

Sejam as produções do tipo:

$A ::= A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$,

com cabeça(β_i) $\neq A$ p/ $i = 1, 2, \ldots n$

Então:

$A ::= \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

$A' ::= \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

OBS: o processo elimina toda recursividade direta à esquerda (desde que os $\alpha_{i's} \neq \varepsilon$), mas, não elimina toda recursividade à esquerda.

Exemplo: Seja a seguinte gramática G(S)

S ::= Aa |b
A ::= Ac |Sd |e

G(S) é recursiva à esquerda em S

$S \Rightarrow Aa \Rightarrow Sda$
$(S \Rightarrow^{+} Sda)$

Solução: Usar um algoritmo geral de eliminação de recursividade à esquerda.

Algoritmo: Elimina a recursividade à esquerda de gramáticas sem ciclos (derivações da forma $A \Rightarrow^{+} A$) e sem ε-produções (produções da forma $A \Rightarrow \varepsilon$)

1. Organize os símbolos não_terminais da gramática em uma certa ordem $A_1, A_2, \ldots, A_n$

2. **para** i := 1 até n **faça**
   para j := 1 até i – 1 **faça**
   substitua cada produção da forma
   $A_i ::= A_j\gamma$ pelas produções $A_i ::= \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,
   onde $A_j ::= \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$;

   elimine a recursividade direta à esquerda das
   $A_{i's}$ – produções

Exemplo: Eliminar a recursividade à esquerda da seguinte Gramática G(S)

$S ::= Aa \mid b$ $A ::= Ac \mid Sd \mid e$	$\Rightarrow$	$A_1 ::= A_2a \mid b$ $A_2 ::= A_2c \mid A_1d \mid e$

S – (A_1) não tem recursividade direta à esquerda

$$\underset{i}{A_2} ::= \underset{j\ \gamma}{A_1}d \Rightarrow A_2 ::= \underbrace{A_2ad}_{\delta 1} \mid \underset{\delta 2}{bd}$$

Eliminando a recursividade direta de A_2 temos:

$A_1 ::= A_2a \mid b$ $A_2 ::= bdA'_2 \mid eA'_2$ $A'_2 ::= cA'_2 \mid adA'_2 \mid \varepsilon$	$\Rightarrow$	$S ::= Aa \mid b$ $A ::= bdA' \mid eA'$ $A' ::= cA' \mid adA' \mid \varepsilon$

- **Problema das produções não-determinísticas**

Regras do tipo:

1) $A ::= \alpha\beta$
2) $A ::= \alpha\gamma$

conduzem o analisador sintático a uma situação de indefinição. A partir do ponto A, na árvore de derivação sintática, para se chegar à cadeia α que regra usar, a regra 1 ou a 2?

Esse problema pode ser facilmente contornado transformando essas duas regras em:

3) A ::= αC
4) C ::= $\beta|\gamma$

- **Problema da linguagem de programação não recursiva para implementar o analisador sintático**

→ Na indisponibilidade de uma linguagem de programação recursiva p/ implementar o analisador (ou por questões de "*eficiência*"!!!), podemos utilizar um Analisador Sintático Descendente Preditor ou *Parsing* Preditivo (ou analisador de gramáticas LL(k)).

→ A idéia do analisador LL(k) ("Left-to-right Left-most-derivation k") é de que basta olharmos no máximo k símbolos à frente na sentença, a partir do ponto em que estamos na ADS, para que possamos decidir que regra de produção aplicar.

Exemplo:

S ::= aS | bS | c	S ::= abS | acS | ad
G(S) é LL(1) w = abc w = bac	G'(S) é LL(2) w = abacad w = acad

→ Em termos de linguagens de programação, quase sempre é possível obter-se uma gramática LL(1) que permita o reconhecimento sintático de programas através de um analisador LL(1), que é bastante simples de implementar.

- **ASD Preditor ou *Parsing* Preditivo**

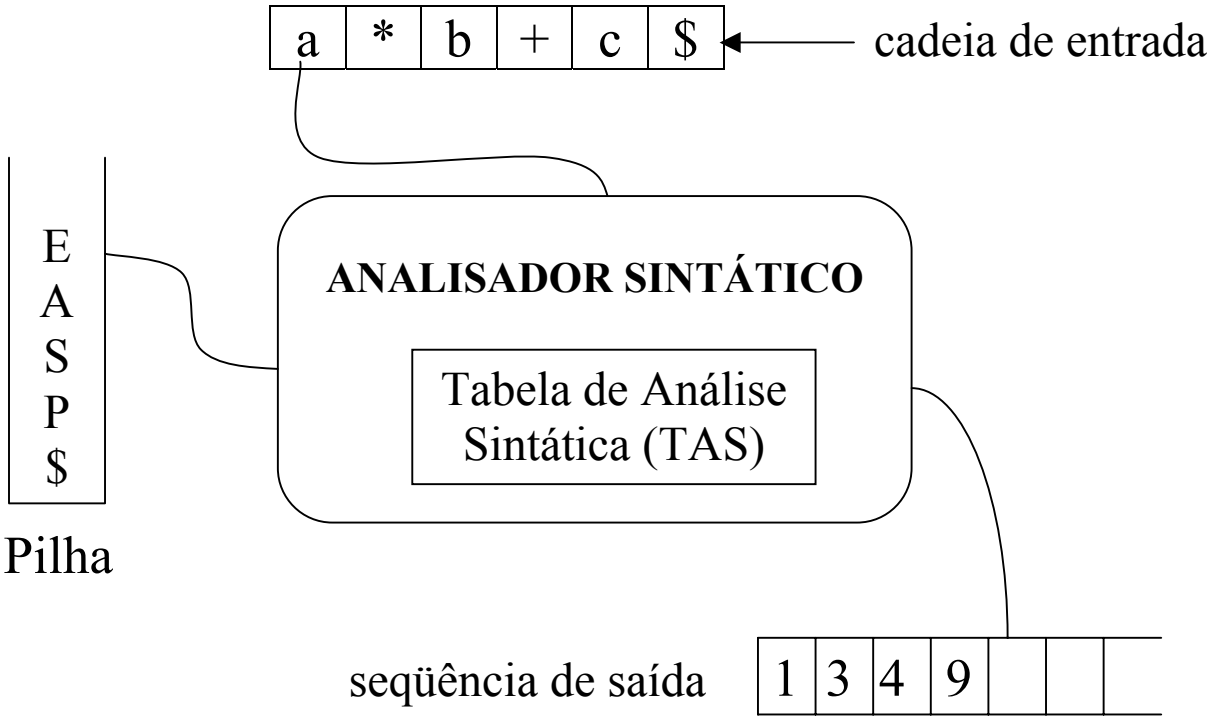
  Implementa o descendente recursivo utilizando explicitamente uma pilha.

  A idéia é a seguinte:

  - O analisador sintático recebe uma seqüência de entrada (a sentença a ser analisada);
  - Manipula uma estrutura de dados tipo pilha (onde monta a ADS);
  - Para cada símbolo de entrada, consulta uma tabela de análise sintática para saber que regra aplicar;
  - Emite uma seqüência de saída (regras que estão sendo aplicadas).

Veja o esquema a seguir.

Modelo de um Analisador Sintático Preditor



→Entrada do analisador:

- Sentença a ser analisada, seguida por um símbolo delimitador ($).

- Pilha: contém uma seqüência de símbolos da gramática, precedida pelo indicador de base de pilha ($).

- Tabela de análise sintática: é uma matriz M[A, a], onde 'A' é um não-terminal e 'a' é um terminal ou dólar ($).

→Saída do analisador:

Constará das produções aplicadas a partir do símbolo inicial (S), na geração da sentença.

→Funcionamento do analisador:

A partir de X, símbolo do *topo da pilha*, e de próximo_símbolo, o atual *símbolo de entrada*, o analisador determina sua ação que pode ser uma das quatro possibilidades a seguir:

1) Se X é um terminal = próximo_símbolo = $, o analisador encerra sua atividade e comunica fim da análise sintática com sucesso;

2) Se X é um terminal = próximo_símbolo ≠ $, o analisador elimina X do topo da pilha e avança para o próximo símbolo de entrada;

3) Se X é um terminal ≠ próximo_símbolo, o analisador acusa um erro de sintaxe (ativa rotina de tratamento de erros);

4) Se X é um não-terminal, o analisador consulta M[X, próximo_símbolo]. Se a resposta for uma regra de produção X ::= MVU, o analisador desempilha X do topo da pilha e empilha UVM (com M no topo da pilha). Para a saída é enviada a regra de produção usada. Se M[X, próximo_símbolo] = ERRO, o analisador acusa um erro de sintaxe (ativa rotina de tratamento de erros).

Exemplo:

Sejam a gramática G(S) e a respectiva tabela de análise sintática.

(1)	S ::= aAS	(3)	A ::= a
(2)	S ::= b	(4)	A ::= bSA

	a	b	$
S	1	2	ERRO
A	3	4	ERRO

Dada a sentença w = abbab$, o analisador sintático assumiria as seguintes configurações durante a análise:

ENTRADA	PILHA	SAÍDA
abbab$	$S	
abbab$	$SAa	1
bbab$	$SA	1
bbab$	$SASb	1 4
bab$	$SAS	1 4
bab$	$SAb	1 4 2
ab$	$SA	1 4 2
ab$	$Sa	1 4 2 3
b$	$S	1 4 2 3
b$	$b	1 4 2 3 2
$	$	1 4 2 3 2

Vejamos o algoritmo do analisador preditor.

→Algoritmo do analisador preditor

```
início
    /* seja X o símbolo do topo da pilha e
         próximo_símbolo o símbolo atual da entrada*/
  enquanto X ≠ $ faça {
    se X é terminal então
         se X = próximo_símbolo então {
              elimine X do topo da pilha;
              leia_próximo_símbolo();
         }
         senão ERRO();
    senão se M[X, próximo_símbolo] = "X::=Y1Y2...Yk" então{
              elimine X do topo da pilha;
              empilhe Yk, ..., Y2, Y1
              }
            senão ERRO();
  }
  se próximo_símbolo ≠ $ então ERRO();
fim
```

→**Obtenção da Tabela** (ou matriz) **de Análise Sintática**

Para construí-la, precisamos introduzir dois novos conceitos (ou relações) em gramáticas. São os conceitos de Primeiro ("First") e Seguidor ("Follow").

Primeiro(α) = $\{x \mid x \in V_T, \alpha =>^* x\beta, \alpha \in V^+, \beta \in V^*\}$
se $\alpha =>^* \varepsilon$ então $\varepsilon \in$ Primeiro(α) }

Seguidor(A) = $\{a \mid S=>^* \alpha Aa\beta, a \in V_T, A \in V_N, \alpha,\beta \in V^*,$
e S símbolo inicial}
se $S =>^* \alpha A$ então $\varepsilon \in$ Seguidor(A)

Exemplo:

Dada a gramática:

E ::= TE'
E' ::= +TE' | ε
T ::= FT'
T' ::= *FT' | ε
F ::= (E) | id

Temos:

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

Seguidor(E) = Seguidor(E') = {), ε }
Seguidor(T) = Seguidor(T') = { +,), ε }
Seguidor(F) = { +, *,), ε }

Obtenção do conjunto Seguidor(A)

Algoritmo: Obtém o conjunto SEGUIDOR para todos os não terminais de uma gramática G(S).

SEGUIDOR(S) ← ε
Para toda produção A → αBβ **faça**
    SEGUIDOR(B) ← PRIMEIRO(β)≠ε;
Repita
  Para toda produção A → αB ou A → αBβ com ε∈PRIMEIRO(β)**faça**
    SEGUIDOR(B) ← SEGUIDOR(A)
Até não adicionar nenhum símbolo a qualquer conjunto SEGUIDOR

Algoritmo para a obtenção da tabela (matriz) de análise sintática

início
  para cada produção A ::= α da gramática, **faça** {
    para cada símbolo terminal *a* ∈ Primeiro(α), **faça**
      adicione a produção A ::= α em M[A, *a*];
    se ε ∈ Primeiro(α), adicione a produção A ::= α
      em M[A, *b*], para cada terminal *b* ∈ Seguidor(A);
    se ε ∈ Primeiro(α) e ε ∈ Seguidor(A), adicione a
      produção A ::= α em M[A, $]
  }
  indique situação de ERRO para todas as posições
    indefinidas de M[A, a];
fim

Exemplo:

Para a gramática abaixo, obtém-se:

(1)	E	::= TE'	(5)	T'	::= *FT'
(2)	E'	::= +TE'	(6)	T'	::= ε
(3)	E'	::= ε	(7)	F	::= (E)
(4)	T	::= FT'	(8)	F	::= id

	id	+	*	(	)	$
E	1			1		
E'		2			3	3
T	4			4		
T'		6	5		6	6
F	8			7		

O algoritmo dado é válido para qualquer gramática, porém, para algumas gramáticas(ambíguas e/ou recursivas à esquerda), a matriz M possui algumas entradas multiplamente definidas;

Exemplo: A gramática abaixo é ambígua.

1) <cmd> ::= if <cond> then <cmd> <pelse>
2) <cmd> ::= <atribuição>
3) <cmd> ::= <ativação>
4) <pelse> ::= else <cmd>
5) <pelse> ::= ε
6) <cond> ::= b
7) <cond> ::= a

A sentença

w = if b then if a then <atribuição> else <ativação>

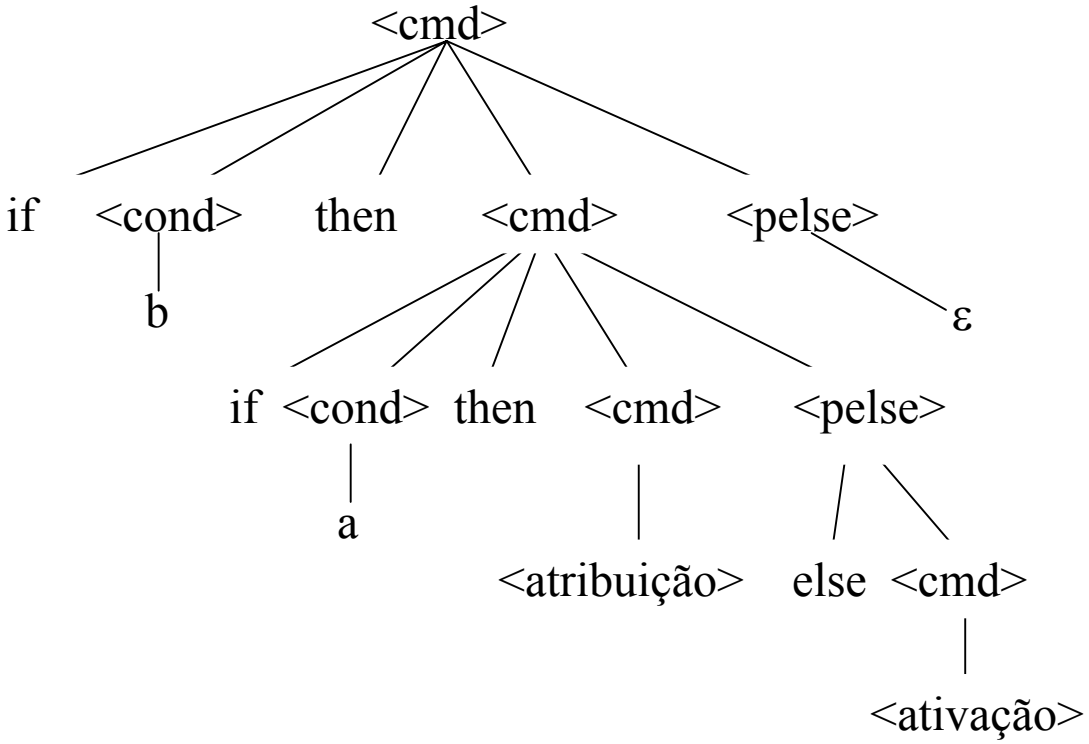
pode ser interpretada de duas formas diferentes.

```
if b then
    if a then
        <atribuição>
    else
        <ativação>
```

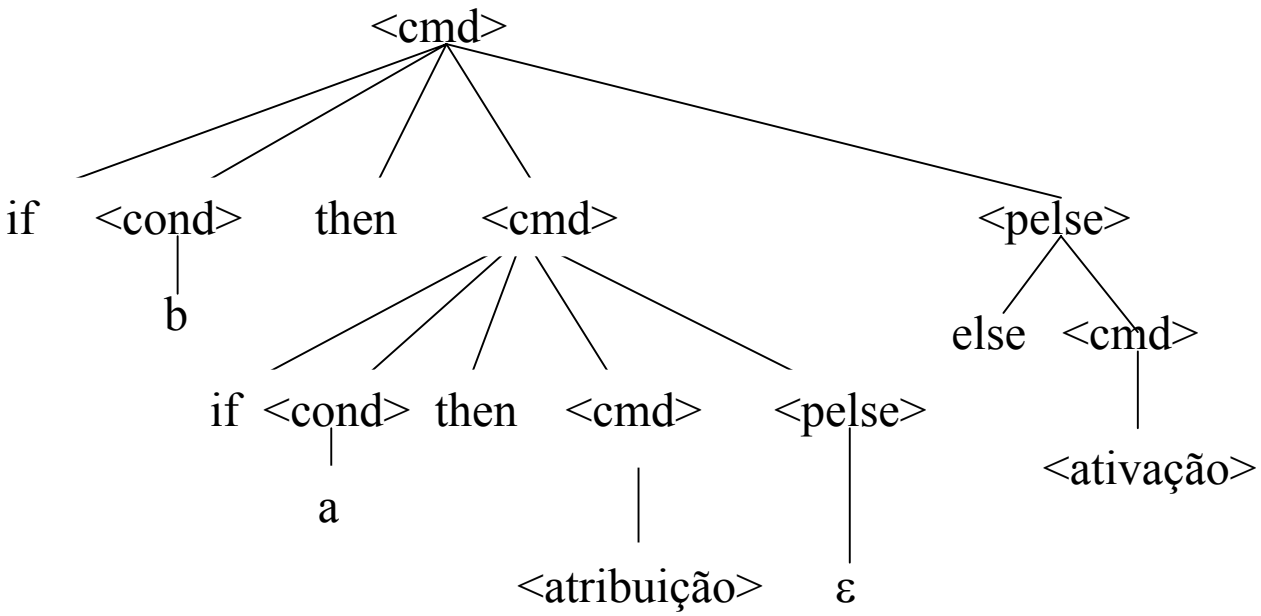
ou:

```
if b then
    if a then
        <atribuição>
else
    <ativação>
```

Em termos de árvore de derivação sintática temos:



ou



As condições para executar o comando <atribuição> são as mesmas nas duas interpretações (a = b = verdadeira)

Para executar o comando <ativação> as condições diferem:
(b = verdadeira e a = falso) p/ a 1ª e (b = falso e a = *) p/ a 2ª

Para essa gramática, teríamos a seguinte matriz de análise sintática:

	a	b	if	then	else	$
<cmd>			1			
<pelse>					4 / 5	5
<cond>	7	6				

→Gramáticas LL(1)

São aquelas cujas tabelas de análise sintática possuem no máximo uma produção para cada par (A, a), onde $A \in V_N$ e $a \in V_T \cup \{\$\}$

Pode ser demonstrado que:

Se $A ::= \alpha|\beta$ são duas produções distintas de G, então G é uma gramática LL(1) se somente se

1) Primeiro(α) ≠ Primeiro(β)
2) Se $\beta \Rightarrow^* \varepsilon$, então α só deriva cadeias do tipo $\alpha \Rightarrow^* a \ldots$ onde $a \notin$ Seguidor(A).

→ Implementação do Analisador Sintático Descendente LL(1)

Um analisador sintático preditor pode ser implementado facilmente utilizando-se certas convenções:

A principal: codificar todos os símbolos usados na representação da gramática (terminais e não-terminais) através de números inteiros.

Símbolos terminais - usar os códigos que foram atribuídos pelo analisador léxico;

Símbolos não-terminais - podem ser codificados através de números negativos (para não serem confundidos com os terminais).

Exemplo: Seja a gramática G(E):

(1)	E	::= TE'	(5)	T'	::= *FT'
(2)	E'	::= +TE'	(6)	T'	::= ε
(3)	E'	::= ε	(7)	F	::= (E)
(4)	T	::= FT'	(8)	F	::= id

podemos utilizar a seguinte codificação:

Símbolo	Código
id	1
+	11
*	13
(	21
)	22
$	99 /* fim de sentença/pilha */
E	-1
E'	-2
T	-3
T'	-4
F	-5

Usar as seguintes estruturas de dados:

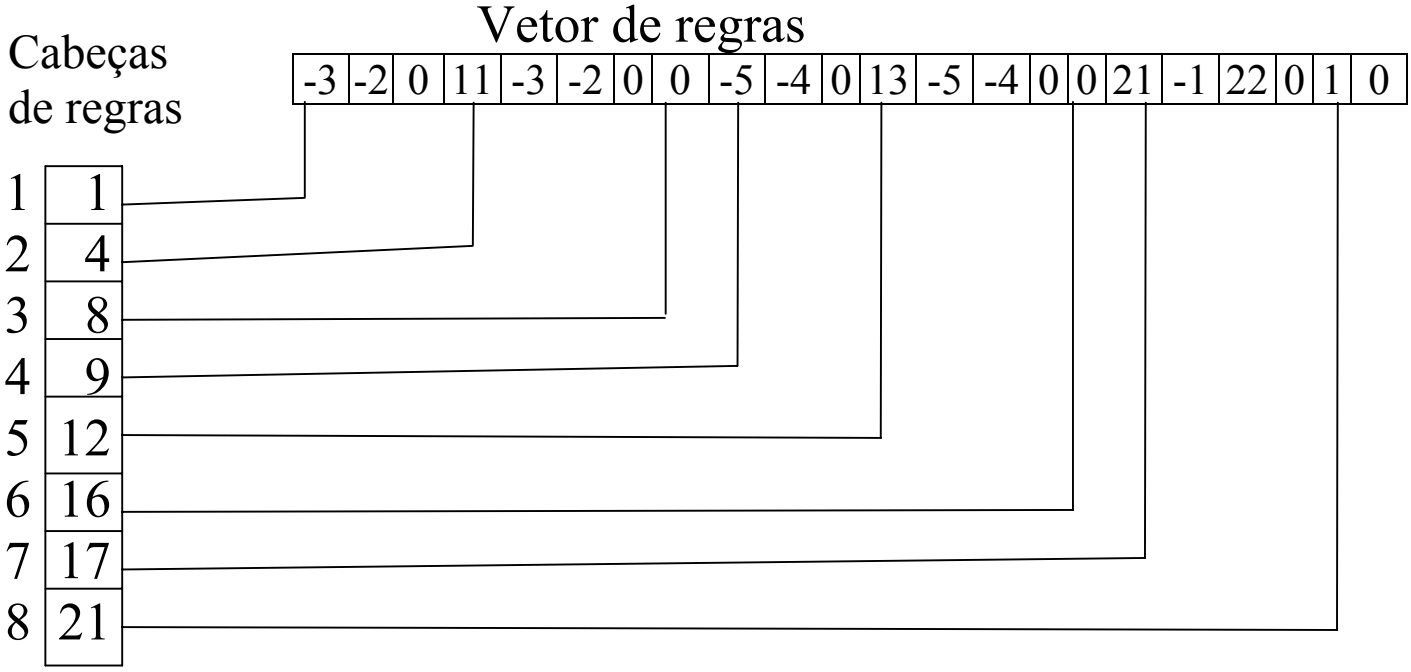
- Pilha de análise: vetor de inteiros;
- Matriz de análise: matriz de inteiros;
- Vetor de regras: vetor de inteiros para conter todas as regras de produção codificadas.

A representação para a matriz de análise e o vetor de regras são indicadas a seguir.

	id	+	*	(	)	$
E	1			1		
E'		2			3	3
T	4			4		
T'		6	5		6	6
F	8			7		

OBS. As posições em branco são situações de erro.

Representação das regras



O vetor de regras contém todas as regras de produção da gramática, já devidamente codificadas para facilitar o trabalho do analisador.

Uma posição M[Não-terminal, Terminal] da matriz de análise sempre vai conter:

- um índice para o vetor de cabeças de regras indicando que produção deve ser aplicada para, partindo-se de Não-terminal, chegar-se a Terminal;

- uma indicação de ERRO para dizer que não é possível derivar Não-terminal e chegar-se em terminal.

• Referências

Notas de aulas do prof. Giuseppe Mongiovi do DI/UFPB, 2002.

Appel, A. W. **Modern Compiler Implementation in C.** Cambridge University Press, 1998. (Capítulo 3, seção 3.2).