# Análise léxica



## Características gerais

- Funcionalidade:
  - reconhecer palavras-chave, identificadores, números, cadeias de caracteres, símbolos compostos e símbolos simples, tranformando-os em códigos e valores (*átomos*)
  - tratar espaços em branco, mudanças de linha, comentários
  - tratar mudanças de arquivos (inclusões) etc.
- Implementação típica: uma função que devolve o código do próximo símbolo de entrada e, eventualmente, o seu valor.
- Alternativas para implementação:
  - abordagem *ad hoc*
  - ferramentas automáticas: *lex* e *flex*
- Algumas linguagens apresentam problemas particulares como, por exemplo, uso de palavras-chave (mas não reservadas) como identificadores.



## Abordagem *ad hoc* (esboço)

- Se o próximo caractere é um símbolo especial; verifique se pode ter mais de um caractere (ex: $<=$) e devolva o seu código
- Se o próximo símbolo é um dígito (ou, às vezes, símbolo de sinal), trate o resto do número e devolva o seu código (inteiro, real, etc) e o seu valor
- Se o próximo símbolo pode iniciar um identificador, trate o resto dele; se for uma palavra reservada, devolva o código correspondente; senão, devolva o código de identificador e o valor da cadeia
- Se o próximo símbolo indica o início de uma cadeia, trate o resto dela e devolva o código de cadeia e o seu valor
- Para fins de mensagens de erro e de depuração, mantenha variáveis convenientes com o nome do arquivo, o número da linha, posição na linha, etc.



## Ferramentas

- Ferramentas baseadas em expressões regulares que geram a função; especificações típicas para *lex* ou *flex*:

```
begin             return(BEGIN_SYMBOL);
if                return(IF_SYMBOL);
while             return(WHILE_SYMBOL);
...               ...
"("               return(OPEN_PAREN);
")"               return(CLOSE_PAREN);
"+"               return(PLUS);
"<="              return(LESS_EQUAL);
...               ...
[a-z][a-z0-9]*    {token=yytext; return(IDENT);}
[0-9]+            {token=yytext; return(INTEGER);}
\".*\"            {token=yytext; return(STRING);}
```

(O símbolo '.' (ponto) indica qualquer caractere.)

# Análise sintática descendente



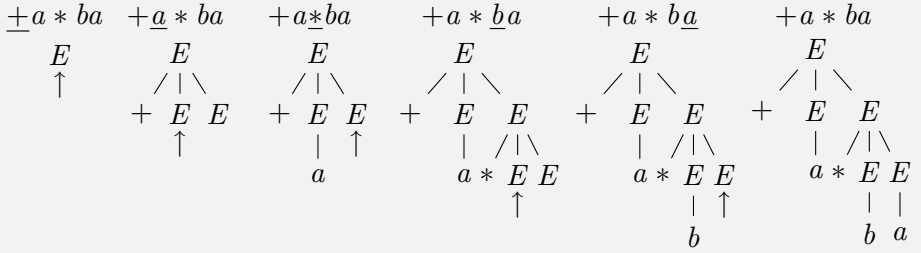
## Exemplo

Gramática de expressões pré-fixas (notação polonesa):

$$E \leftarrow a \mid b \mid +EE \mid *EE$$

Cadeia: $+a*ba$

Está sublinhado em cada passo o símbolo que determina a produção a ser aplicada e indicado com uma seta o não terminal ao qual esta produção se refere (sempre o primeiro mais à esquerda na árvore). Inicialmente, a árvore consiste apenas da sua raiz $E$.






## Observações sobre análise descendente

- Nota-se que a seqüência de produções utlizadas corresponde a uma derivação esquerda. Esta é uma propriedade geral de algoritmos de análise descendente (*top down*).
- O problema básico da análise descendente é a determinação da produção a ser utilizada para expandir o não terminal corrente (o mais à esquerda da forma sentencial corrente constituída pelas folhas da árvore).
- Pode-se observar que o algoritmo vai precisar de uma pilha (explícita ou escondida na recursão) para guardar a forma sentencial corrente.
- Os algoritmos de análise sintática descendente pertencem à classe *LL(k)* (*input from **L**eft to right producing a **L**eftmost derivation with $k$ tokens look-ahead*).
- Há várias maneiras de implementar a análise descendente, sendo uma das mais comuns a implementação através de um conjunto de funções mutuamente recursivas, uma para cada símbolo não terminal.



## Exemplo de implementação

Suporemos que cada chamada da função *expr* (apenas uma neste caso, já que existe apenas um não-terminal $E$) devolve uma árvore de derivação produzida de maneira que fica óbvia neste código indicado em pseudo-C. Neste código, a função *nexttoken* devolve o próximo átomo de entrada enquanto *advance* avança para o átomo seguinte. As funções *mktree1* e *mktree3* constroem árvores a partir dos seus argumentos que são o símbolo da raiz e as subárvores.

```
Tree expr() {
  Token tok = nexttoken();
  if (tok=='a' || tok=='b') {
    advance();
    return mktree1('E',tok);
  } else if (tok=='+' || tok=='*') {
    advance();
    return mktree3('E',tok,expr(),expr());
  } else error();
} /* expr */
```

## Observações

- Obviamente trata-se de um exemplo trivial no qual todas as produções têm o primeiro símbolo terminal e distinto o que permite uma implementação muito simples.
- Num caso mais geral, poder-se-ia calcular o conjunto de primeiros símbolos deriváveis de cada produção (função $\mathrm{FIRST}$) e usar a mesma técnica se os conjuntos forem disjuntos. Por exemplo:

$$E \leftarrow L \mid OEE$$
$$L \leftarrow a \mid b$$
$$O \leftarrow + \mid *$$

- Neste caso, teríamos três funções (por ex., *expr*, *ident* e *op*). A função *expr* não teria nenhum problema em decidir qual das duas alternativas deve ser aplicada, já que os primeiros símbolos deriváveis do não-terminal $L$ são $\{a, b\}$ enquanto que os deriváveis de $O$ são $\{+, *\}$.



## Exemplo de expressões comuns

- Retomemos a gramática não ambígua de expressões comuns:

$$\begin{array}{lcl} E & \leftarrow & E + T \\ E & \leftarrow & T \\ T & \leftarrow & T * F \\ T & \leftarrow & F \end{array} \qquad \begin{array}{lcl} F & \leftarrow & (E) \\ F & \leftarrow & a \\ F & \leftarrow & b \end{array}$$

- Neste caso, não há como a função que implementa o terminal $E$ decidir qual das duas alternativas ($E+T$ ou $T$) deve ser aplicada, uma vez que tanto a partir de $E$ quanto de $T$ podem ser derivadas cadeias que começam com um dos símbolos do conjunto $\{a, b, (\}$. Um problema análogo acontece com a função que implementa o não terminal $T$ (mas não $F$).
- Este problema permanece mesmo que seja possível consultar um número $k>1$ de símbolos para frente, já que é possível derivar de $E$ (e de $T$) cadeias de qualquer comprimento.



## Problemas de análise descendente

- Na realidade, um outro problema, mais complicado ainda, é que a função que implementa o não-terminal $E$, no caso de escolher a alternativa $E+T$, teria que executar, como sua primeira ação, uma chamada recursiva de si mesma, sem ter avançado na cadeia de entrada. Conseqüentemente, ela entraria numa repetição infinita.
- Este é um problema geral com gramáticas que apresentam *recursão esquerda*, ou seja, nas quais existem não-terminais $X$ tais que:

$$X \Rightarrow X\alpha$$

para alguma cadeia $\alpha$. No caso do nosso exemplo, temos:

$$E \Rightarrow E + T$$
$$T \Rightarrow T * F$$

- A recursão esquerda *indireta* com

$$X \stackrel{+}{\Rightarrow} X\alpha$$

apresenta o mesmo problema.

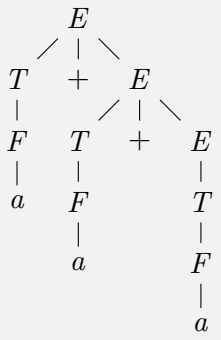


## Eliminação da recursão esquerda

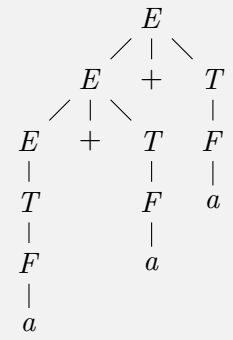
- No caso deste exemplo, é possível eliminar a recursão esquerda introduzindo a recursão direita:

$$E \leftarrow T + E \mid T$$
$$T \leftarrow F * T \mid F$$
$$F \leftarrow a \mid b \mid (E)$$

- Entretanto, esta gramática impõe a associação à direita dos operadores '$+$' e '$*$', o que não corresponde à convenção usada em geral. Assim, com esta gramática, a árvore de derivação para $a+a+a$ seria:




ao invés de



## Fatoração

- Esta última forma ainda apresenta o problema de escolha de produção, já que as duas alternativas para $E$ (e para $T$) podem começar com os mesmos símbolos. Uma solução é usar a *fatoração*:

$$\begin{array}{lll} E \leftarrow T + E \mid T & & E \leftarrow TE' \\ T \leftarrow F * T \mid F & \text{passa para} & E' \leftarrow +E \mid \epsilon \\ F \leftarrow a \mid b \mid (E) & & T \leftarrow FT' \\ & & T' \leftarrow *T \mid \epsilon \\ & & F \leftarrow a \mid b \mid (E) \end{array}$$

- Esta técnica introduz lados direitos vazios que precisam de um tratamento especial. Normalmente, convenciona-se que a alternativa vazia para $E'$ (ou $T'$) será escolhida somente se o próximo símbolo não for '$+$' (ou '$*$'). Esta regra dá resultados corretos para esta gramática, mas podem ser construídos exemplos mais complicados.



## Notação estendida

- Uma solução utilizada na prática é estender a notação de gramáticas para introduzir um operador de repetição (análogo às expressões regulares).
- No caso, teríamos:

$$\begin{array}{l} E \leftarrow T\{+\,T\} \\ T \leftarrow F\{*\,F\} \\ F \leftarrow a \mid b \mid (E) \end{array}$$

onde a construção $\{\alpha\}$ equivale a $\alpha^*$ (neste caso, '$\{$' e '$\}$' são meta-símbolos).



## Implementação da análise descendente

- A última forma da gramática permite uma implementação simples através de um conjunto de funções mutuamente recursivas.
- Normalmente, simplifica-se também a forma da árvore obtida que deixa de ser uma árvore de derivação no sentido estrito. Entretanto, ela preserva toda a estrutura necessária ao compilador.

```
Tree expr() {
  Tree t = term();
  Token tok = nexttoken();
  while (tok=='+') {
    advance();
    t = mktree2('+',t,term());
    tok = nexttoken();
  }
} /* expr */
```



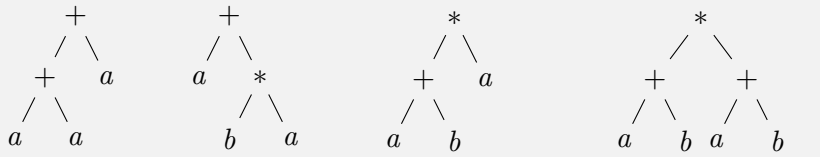
## Implementação da análise descendente (cont.)

- A função *term* é análoga à *expr*.
- A função *factor*:

```
Tree factor() {
  Tree t;
  Token = nexttoken();
  if (tok=='a' || tok=='b') {
    advance();
    return mktree0(tok);
  } else if (tok=='(') {
    advance();
    t = expr();
    if nexttoken()==')'{
      advance();
      return t;
    } else
      error();
  } else
    error();
} /* factor */
```

## Exemplos de árvores construídas

$a + a + a$ $\quad$ $a + b * a$ $\quad$ $(a + b) * a$ $\quad$ $(a + b) * (a + b)$





## Outro exemplo

Gramática ambígua (parcial): $S \leftarrow$ **if** $E$ **then** $S$ | **if** $E$ **then** $S$ **else** $S$

```
Tree statement() {
  Token tok = nexttoken();
  Tree exp, stat1, stat2;
  if (tok==IF_SYMBOL) {
    advance();  exp = expr();
    if (nexttok()!=THEN_SYMBOL)
      error();
    else {
      advance();  stat1 = statement();
      if (nexttok()==ELSE_SYMBOL) {
        advance();  stat2 = statement();
      } else
        stat2 = NULL;
      return mktree3('S',exp,stat1,stat2);
    }
  } else { /* tok!=IF_SYMBOL */
    /* other statements */
  }
} /* statement */
```



## Cartas sintáticas

- As gramáticas (notação estendida) de linguagens de programação são freqüentemente substituídas por *cartas sintáticas* (ou *diagramas sintáticos*). Estas são, na realidade, apenas uma maneira diferente de representar as mesmas produções, mas de maneira gráfica. Elas são muito convenientes no contexto de análise descendente.
- Retomemos a gramática de expressões em notação estendida, incluindo agora os operadores unários '+' e '−', e o operador binário '/'. Novamente, '**(**' e '**)**' são meta-símbolos.

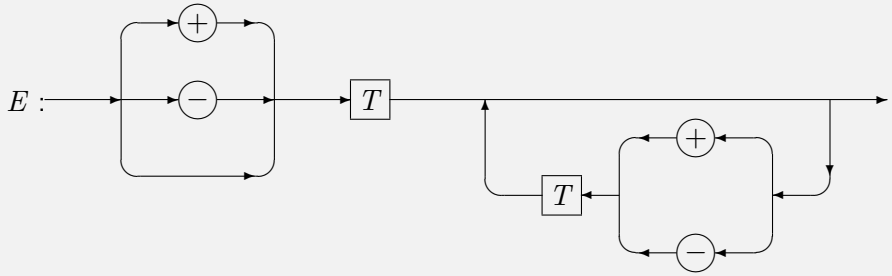
$$E \leftarrow \mathbf{(} + \mid - \mid \epsilon \mathbf{)}\ T\ \mathbf{\{(} + \mid - \mathbf{)}\ T\mathbf{\}}$$
$$T \leftarrow F\mathbf{\{(} * \mid / \mathbf{)}\ F\mathbf{\}}$$
$$F \leftarrow a \mid b \mid (E)$$

- Convenção: símbolos terminais aparecem dentro de círculos; símbolos não-terminais dentro de retângulos.

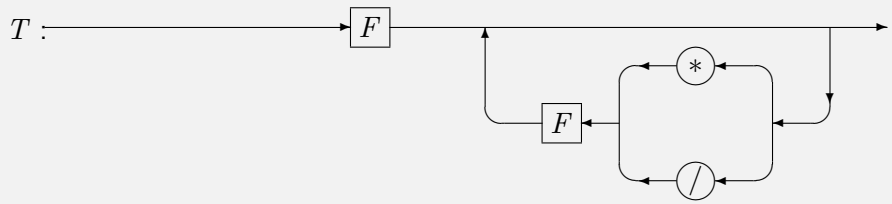


## Cartas sintáticas (cont)

$E \leftarrow \mathbf{(} + \mid - \mid \epsilon \mathbf{)}\ T\ \mathbf{\{(} + \mid - \mathbf{)}\ T\mathbf{\}}$

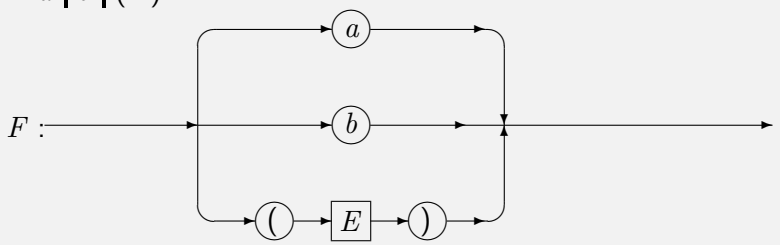



$T \leftarrow F\mathbf{\{(} * \mid / \mathbf{)}\ F\mathbf{\}}$



## Cartas sintáticas (cont)

$F \leftarrow a \mid b \mid (E)$




- Deve-se notar que as cartas sintáticas indicam muito bem a estrutura das respectivas funções que implementam a análise descendente.
- Por outro lado, elas não indicam algumas propriedades como, por exemplo, a associatividade que já foi discutida.



## Utilização

- A análise descendente recursiva é muito usada para implementação manual de analisadores; exemplos: implementações de Pascal e Modula-3.
- Existem várias ferramentas que automatizam a implementação da análise descendente:
  - ANTLR: http://www.antlr.org
  - JavaCC: https://javacc.dev.java.net
  - Coco/R: http://www.ssw.uni-linz.ac.at/Research/Projects/Coco



# Análise sintática ascendente



## Exemplo

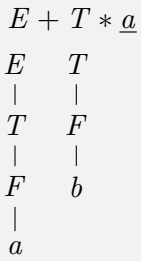
Gramática: $E \leftarrow E + T \mid T, \quad T \leftarrow T * F \mid F, \quad F \leftarrow a \mid b \mid (E)$

Cadeia: $a + b * a$

Está sublinhada, em cada passo, a parte da cadeia de entrada que foi identificada para *redução* (o contrário da derivação) no próximo passo. A maneira de identificar esta subcadeia será explicada mais adiante.

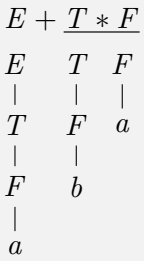
| (1) | (2) | (3) | (4) | (5) |
|---|---|---|---|---|
| $\underline{a} + b * a$ | $\underline{F} + b * a$ | $\underline{T} + b * a$ | $E + \underline{b} * a$ | $E + \underline{F} * a$ |
| | $F$ \| $a$ | $T$ \| $F$ \| $a$ | $E$ \| $T$ \| $F$ \| $a$ | $E$ $F$ \| \| $T$ $b$ \| $F$ \| $a$ |

## Exemplo (cont.)



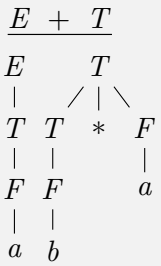

(6)

(Notar que não foram adotados os possíveis redutendos $T$ ou $E+$ $T$.)

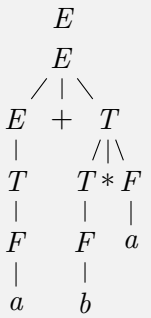



(7)

(Notar que não foi adotado o possível redutendo $F$.)




(8)




(9)



## Observações sobre análise ascendente

- Nota-se que a seqüência de reduções, quando invertida, corresponde a uma derivação direita. Esta é uma propriedade geral de algoritmos de análise ascendente (*bottom up*).
- Os problemas básicos do algoritmo de análise ascendente são:
  - a identificação da próxima subcadeia, chamada *redutendo* (*handle*), a ser reduzida
  - a identificação da produção a ser aplicada – pode haver duas ou mais produções com lados direitos iguais
- Pode-se observar que o algoritmo vai precisar de uma pilha para guardar os resultados (subárvores) das reduções já realizadas.
- Historicamente, existem vários algoritmos de análise ascendente como *análise de precedência simples* e *análise de precedência de operadores*. Entretanto, atualmente são usados quase que exclusivamente os algoritmos da classe *LR(k)* (*input from **L**eft to right producing a **R**ightmost derivation* with $k$ tokens *look-ahead*).
- Todos estes algoritmos executam ações de *empilhamento* (*shift*) e *redução* (*reduce*); daí a denominação *shift/reduce*.



## Algoritmo *LR(1)* básico

- Exemplo 1 – tabela de análise *LR(0)* para a gramática de expressões pré-fixas (produções numeradas para referência); sua construção será indicada mais adiante:

$$1.\ E \leftarrow +EE \quad 2.\ E \leftarrow *EE \quad 3.\ E \leftarrow a \quad 4.\ E \leftarrow b$$

| | $E$ | $+$ | $*$ | $a$ | $b$ | $\#$ |
|---|---|---|---|---|---|---|
| $e_0$ | $\mathbf{e}_1$ | $\mathbf{e}_2$ | $\mathbf{e}_3$ | $\mathbf{e}_4$ | $\mathbf{e}_5$ | |
| $e_1$ | | | | | | **a** |
| $e_2$ | $\mathbf{e}_6$ | $\mathbf{e}_2$ | $\mathbf{e}_3$ | $\mathbf{e}_4$ | $\mathbf{e}_5$ | |
| $e_3$ | $\mathbf{e}_7$ | $\mathbf{e}_2$ | $\mathbf{e}_3$ | $\mathbf{e}_4$ | $\mathbf{e}_5$ | |
| $e_4$ | | $\mathbf{r}_3$ | $\mathbf{r}_3$ | $\mathbf{r}_3$ | $\mathbf{r}_3$ | $\mathbf{r}_3$ |
| $e_5$ | | $\mathbf{r}_4$ | $\mathbf{r}_4$ | $\mathbf{r}_4$ | $\mathbf{r}_4$ | $\mathbf{r}_4$ |
| $e_6$ | $\mathbf{e}_8$ | $\mathbf{e}_2$ | $\mathbf{e}_3$ | $\mathbf{e}_4$ | $\mathbf{e}_5$ | |
| $e_7$ | $\mathbf{e}_9$ | $\mathbf{e}_2$ | $\mathbf{e}_3$ | $\mathbf{e}_4$ | $\mathbf{e}_5$ | |
| $e_8$ | | $\mathbf{r}_1$ | $\mathbf{r}_1$ | $\mathbf{r}_1$ | $\mathbf{r}_1$ | $\mathbf{r}_1$ |
| $e_9$ | | $\mathbf{r}_2$ | $\mathbf{r}_2$ | $\mathbf{r}_2$ | $\mathbf{r}_2$ | $\mathbf{r}_2$ |

- Os rótulos das linhas indicam os *estados* do algoritmo que estarão empilhados; $e_0$ é o estado inicial.
- Os rótulos das colunas indicam os símbolos de entrada ou o resultado da última redução (não-terminal)
- As entradas da tabela indicam as *ações* do algoritmo:
  - $\mathbf{e}_i$ indica empilhar o estado $e_i$
  - $\mathbf{r}_i$ indica aplicar a produção número $i$
  - **a** indica aceitar a entrada
  - as entradas em branco indicam erro



## Algoritmo *LR(1)* básico (cont.)

- Simulação da execução para entrada $+*a+baa$.
  A notação $X_i$ como elemento da pilha indica que foi empilhado o estado $e_i$ quando encontrado na entrada (ou última redução) o símbolo $X$ (apenas para facilitar a leitura).

| | *PILHA* | | *ENTRADA* | | | *PILHA* | | *ENTRADA* | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | $e_0$ | | $+*a+baa\#$ | $\mathbf{e}_2$ | 11 | $e_0{+_2}{*_3}E_7{+_2}E_6$ | $E$ | $a\#$ | $\mathbf{e}_8$ |
| 1 | $e_0{+_2}$ | | $*a+baa\#$ | $\mathbf{e}_3$ | 12 | $e_0{+_2}{*_3}E_7{+_2}E_6E_8$ | | $a\#$ | $\mathbf{r}_1$ |
| 2 | $e_0{+_2}{*_3}$ | | $a+baa\#$ | $\mathbf{e}_4$ | 13 | $e_0{+_2}{*_3}E_7$ | $E$ | $a\#$ | $\mathbf{e}_9$ |
| 3 | $e_0{+_2}{*_3}a_4$ | | $+baa\#$ | $\mathbf{r}_3$ | 14 | $e_0{+_2}{*_3}E_7E_9$ | | $a\#$ | $\mathbf{r}_2$ |
| 4 | $e_0{+_2}{*_3}$ | $E$ | $+baa\#$ | $\mathbf{e}_7$ | 15 | $e_0{+_2}$ | $E$ | $a\#$ | $\mathbf{e}_6$ |
| 5 | $e_0{+_2}{*_3}E_7$ | | $+baa\#$ | $\mathbf{e}_2$ | 16 | $e_0{+_2}E_6$ | | $a\#$ | $\mathbf{e}_4$ |
| 6 | $e_0{+_2}{*_3}E_7{+_2}$ | | $baa\#$ | $\mathbf{e}_5$ | 17 | $e_0{+_2}E_6a_4$ | | $\#$ | $\mathbf{r}_3$ |
| 7 | $e_0{+_2}{*_3}E_7{+_2}b_5$ | | $aa\#$ | $\mathbf{r}_4$ | 18 | $e_0{+_2}E_6$ | $E$ | $\#$ | $\mathbf{e}_8$ |
| 8 | $e_0{+_2}{*_3}E_7{+_2}$ | $E$ | $aa\#$ | $\mathbf{e}_6$ | 19 | $e_0{+_2}E_6E_8$ | | $\#$ | $\mathbf{r}_1$ |
| 9 | $e_0{+_2}{*_3}E_7{+_2}E_6$ | | $aa\#$ | $\mathbf{e}_4$ | 20 | $e_0$ | $E$ | $\#$ | $\mathbf{e}_1$ |
| 10 | $e_0{+_2}{*_3}E_7{+_2}E_6a_4$ | | $a\#$ | $\mathbf{r}_3$ | 21 | $e_0E_1$ | | $\#$ | **a** |

## Observações

- Na simulação, está subentendida a construção das árvores.
- Os estados empilhados "lembram" a posição do algoritmo dentro de um lado direito parcialmente reconhecido. Por exemplo quando o topo da pilha é constituído pelos dois estados $*_3 E_7$ (na realidade $e_3 e_7$), isto indica que o algoritmo já consumiu o símbolo '$*$', seguido de uma cadeia que foi reduzida ao símbolo '$E$'.
- Os algoritmos de análise *LR(0)* e *LR(1)* são idênticos; as análises diferem apenas na maneira de construir a tabela.
- Numa tabela de análise *LR(0)*, uma dada linha contém apenas ações de empilhamento ou apenas ações de redução (ou erros), ou apenas a ação de aceitação. Em outras palavras, o tipo de ação não depende do próximo símbolo lido na entrada.
- No caso de tabelas de *LR(1)*, a escolha do tipo de ação poderá depender do próximo símbolo de entrada.
- No caso geral de *LR(k)*, $k \geq 1$, a escolha da ação poderá depender dos $k$ símbolos de entrada seguintes. Na prática, o caso mais comum é $k = 1$.



## Construção *LR(0)*

- *Item* (do tipo *LR(0)*): uma produção com posição do lado direito.
- Exemplo: os itens da gramática de expressões pré-fixas

$$E \leftarrow \bullet + EE \mid + \bullet EE \mid + E \bullet E \mid + EE \bullet \mid$$
$$\bullet * EE \mid * \bullet EE \mid * E \bullet E \mid * EE \bullet \mid$$
$$\bullet a \mid a \bullet \mid \bullet b \mid b \bullet$$

- *Itens completos*: itens com '$\bullet$' no fim do lado direito.
- $K$ é um conjunto *fechado* de itens se, para todo item $A \leftarrow \alpha \bullet B \beta$ de $K$, todos os itens da forma $B \leftarrow \bullet \gamma$ também estão em $K$.
- Dado um conjunto inicial de itens, o cálculo do seu *fecho* pode ser realizado de maneira iterativa, incluindo os itens pela regra acima, até que não haja mais inclusões possíveis; exemplo:

$$K = \{E \leftarrow + \bullet EE\}$$
$$\text{closure(K)} = \{E \leftarrow + \bullet EE \mid \bullet + EE \mid * \bullet EE \mid \bullet a \mid \bullet b\}$$

- Um *estado* é um conjunto fechado de itens.



## Construção *LR(0)* (cont.)

- Se o estado corrente (topo da pilha) contém um item da forma $A \leftarrow \alpha \bullet X \beta$, e o próximo símbolo a ser consultado (entrada ou resultado da última redução) é $X$, então deve ser empilhado um novo estado que contenha o item $A \leftarrow \alpha X \bullet \beta$, indicando que foi reconhecido mais um símbolo do item original.
- Conseqüentemente, dado um estado $K$ e um símbolo $X$, a função $\text{goto}(K, X)$ será dada por:

$$\text{goto}(K, X) = \text{closure}(K')$$
onde $$K' = \{A \leftarrow \alpha X \bullet \beta \mid \forall\, A \leftarrow \alpha \bullet X \beta \in K\}$$

- A função goto determina as entradas da tabela de análise.
- O estado inicial $e_0$ é criado, normalmente:
  - acrescentando-se a produção $S' \leftarrow S\#$ ($S$ é a raiz original da gramática e $\#$ denota o fim da entrada)
  - colocando $e_0 = \text{closure}(\{S' \leftarrow \bullet S\#\})$
- A partir de $e_0$ são calculados os outros estados e a função goto até que não haja mais estados novos.



## Exemplo 1 de cálculo da tabela *LR(0)*

Gramática de expressões pré-fixas:

$0.\ E' \leftarrow E\#$ $\quad 1.\ E \leftarrow +EE$ $\quad 2.\ E \leftarrow *EE$ $\quad 3.\ E \leftarrow a$ $\quad 4.\ E \leftarrow b$

$$\begin{aligned}
e_0 &: && E' \leftarrow \bullet E\# \\
& && E \leftarrow \bullet + EE \mid \bullet * EE \mid \bullet a \mid \bullet b \\
e_1 = \text{goto}(e_0, E) &: && E' \leftarrow E \bullet \# \\
e_2 = \text{goto}(e_0, +) &: && E \leftarrow + \bullet EE \mid \bullet + EE \mid \bullet * EE \mid \bullet a \mid \bullet b \\
e_3 = \text{goto}(e_0, *) &: && E \leftarrow * \bullet EE \mid \bullet + EE \mid \bullet * EE \mid \bullet a \mid \bullet b \\
e_4 = \text{goto}(e_0, a) &: && E \leftarrow a \bullet \\
e_5 = \text{goto}(e_0, b) &: && E \leftarrow b \bullet \\
e_6 = \text{goto}(e_2, E) &: && E \leftarrow + E \bullet E \mid \bullet + EE \mid \bullet * EE \mid \bullet a \mid \bullet b \\
e_7 = \text{goto}(e_3, E) &: && E \leftarrow * E \bullet E \mid \bullet + EE \mid \bullet * EE \mid \bullet a \mid \bullet b \\
e_8 = \text{goto}(e_6, E) &: && E \leftarrow + EE \bullet \\
e_9 = \text{goto}(e_7, E) &: && E \leftarrow * EE \bullet
\end{aligned}$$

## Observações

- Note-se que a presença de um item completo num estado indica a possibilidade de redução.
- Neste exemplo, cada estado contém somente itens de redução ou itens de empilhamento (deslocamento), mas não ambos. Isto demonstra que a gramática é do tipo *LR(0)*. Caso contrário teríamos um conflito do tipo *desloca/reduz*.
- Um único item completo como $E \leftarrow +EE\bullet$ indica que deve ser aplicada a redução para $E$ dada pela produção cujo lado direito é $+EE$. Se houvesse mais de um item completo, haveria um conflito do tipo *reduz/reduz*.
- Não é necessário incluir $\mathrm{goto}(e_1, \#)$, pois o estado $e_1$ indica que a entrada deve ser aceita.
- Pode-se demonstrar que as ações de redução não podem aparecer nas colunas que correspondem aos símbolos não-terminais.
- Os resultados deste cálculo foram usados para formar a tabela de análise apresentada anteriormente.



## Exemplo 2 de cálculo da tabela *LR(0)*

Gramática de expressões com prioridades:

$0.\ E' \leftarrow E\#\quad 1.\ E \leftarrow E + T\quad 2.\ E \leftarrow T\quad 3.\ T \leftarrow T * F$
$4.\ T \leftarrow F\quad 5.F \leftarrow (E)\quad 6.F \leftarrow a$

$e_0$ : $E' \leftarrow \bullet E\#$
$E \leftarrow \bullet E + T \mid \bullet T$
$T \leftarrow \bullet T * F \mid \bullet F$
$F \leftarrow \bullet(E) \mid \bullet a$

$e_1$ : $E' \leftarrow E \bullet \#$
$E \leftarrow E\bullet + T$

$e_2$ : $E \leftarrow T\bullet$
$T \leftarrow T\bullet * F$

$e_3$ : $T \leftarrow F\bullet$

$e_4$ : $F \leftarrow (\bullet E)$
$E \leftarrow \bullet E + T \mid \bullet T$
$T \leftarrow \bullet T * F \mid \bullet F$
$F \leftarrow \bullet(E) \mid \bullet a$

$e_5$ : $F \leftarrow a\bullet$

$e_6$ : $E \leftarrow E + \bullet T$
$T \leftarrow \bullet T * F \mid \bullet F$
$F \leftarrow \bullet(E) \mid \bullet a$

$e_7$ : $T \leftarrow T * \bullet F$
$F \leftarrow \bullet(E) \mid \bullet a$

$e_8$ : $F \leftarrow (E\bullet)$
$E \leftarrow E\bullet + T$

$e_9$ : $E \leftarrow E + T\bullet$
$T \leftarrow T\bullet * F$

$e_{10}$ : $T \leftarrow T\bullet * F$

$e_{11}$ : $F \leftarrow (E)\bullet$



## Observações

- Não foi indicada a função goto que é óbvia.
- Existem dois estados ($e_2 = \mathrm{goto}(e_0, T)$ e $e_9 = \mathrm{goto}(e_6, T)$) que contêm tanto itens completos quanto incompletos. Isto demonstra que a gramática não é do tipo *LR(0)*, havendo um conflito *desloca/reduz* nas entradas da tabela $(e_2, *)$ e $(e_9, *)$.
- No caso particular desta gramática há uma solução simples para os conflitos. A presença do estado $e_2$ no topo da pilha indica que foi reduzida a $T$ uma subcadeia de entrada e, neste momento, ou este $T$ é reduzido a $E$ (produção $E \leftarrow T$), ou então, se o próximo símbolo de entrada for '$*$', pode ser empilhado o estado $e_7$. Entretanto, usando a função $\mathrm{FOLLOW}(E) = \{+, ), \#\}$, verifica-se que $E$ não pode ser seguido de '$*$' numa forma senencial. Conseqüentemente, a ação de redução somente pode aparecer nas colunas correspondentes a $+$, $)$ e $\#$ do estado $e_2$.
- O mesmo raciocínio pode ser aplicado em todos os casos de redução; assim as ações de redução deixam de ser colocadas em todas as colunas terminais.



## Observações (cont.)

- As gramáticas cujos conflitos podem ser resolvidos usando esta técnica recebem são chamadas *SLR* (*Simple LR*).
- A tabela obtida é do tipo *LR(1)*.
- Esta técnica, em geral, não é suficiente para resolver todos os conflitos da construção *LR(0)*.
- A tabela final *SLR* para esta gramática é:

| | $E$ | $T$ | $F$ | **a** | $+$ | $*$ | $($ | $)$ | $\#$ |
|---|---|---|---|---|---|---|---|---|---|
| $e_0$ | $\mathbf{e}_1$ | $\mathbf{e}_2$ | $\mathbf{e}_3$ | $\mathbf{e}_5$ | | | $\mathbf{e}_4$ | | |
| $e_1$ | | | | | $\mathbf{e}_6$ | | | | $a$ |
| $e_2$ | | | | | $\mathbf{r}_2$ | $\mathbf{e}_7$ | | $\mathbf{r}_2$ | $\mathbf{r}_2$ |
| $e_3$ | | | | | $\mathbf{r}_4$ | $\mathbf{r}_4$ | | $\mathbf{r}_4$ | $\mathbf{r}_4$ |
| $e_4$ | $\mathbf{e}_8$ | $\mathbf{e}_2$ | $\mathbf{e}_3$ | $\mathbf{e}_5$ | | | $\mathbf{e}_4$ | | |
| $e_5$ | | | | | $\mathbf{r}_6$ | $\mathbf{r}_6$ | | $\mathbf{r}_6$ | $\mathbf{r}_6$ |

| | $E$ | $T$ | $F$ | $a$ | $+$ | $*$ | $($ | $)$ | $\#$ |
|---|---|---|---|---|---|---|---|---|---|
| $e_6$ | | $\mathbf{e}_9$ | $\mathbf{e}_3$ | $\mathbf{e}_5$ | | | $\mathbf{e}_4$ | | |
| $e_7$ | | | $\mathbf{e}_{10}$ | $\mathbf{e}_5$ | | | $\mathbf{e}_4$ | | |
| $e_8$ | | | | | $\mathbf{e}_6$ | | | $\mathbf{e}_{11}$ | |
| $e_9$ | | | | | $\mathbf{r}_1$ | $\mathbf{e}_7$ | | $\mathbf{r}_1$ | $\mathbf{r}_1$ |
| $e_{10}$ | | | | | $\mathbf{r}_3$ | $\mathbf{r}_3$ | | $\mathbf{r}_3$ | $\mathbf{r}_3$ |
| $e_{11}$ | | | | | $\mathbf{r}_5$ | $\mathbf{r}_5$ | | $\mathbf{r}_5$ | $\mathbf{r}_5$ |

## Construção *LR(1)*

- *Item* (do tipo *LR(1)*): uma produção com posição do lado direito e um símbolo terminal (símbolo de consulta – *lookahead*) ou $\epsilon$. A notação poderá ser fatorada com conjuntos de símbolos para itens com o mesmo núcleo (*core*: produção e posição).
- Exemplo: alguns itens da gramática de expressões:

  $E \leftarrow E + \bullet T, \# +$
  $F \leftarrow \bullet(E), \# + * \mid \bullet a, \# + *$
  $T \leftarrow T * F \bullet, ) + *$
- *Itens completos*: itens com '•' no fim do lado direito.
- $K$ é um conjunto *fechado* de itens se, para todo item $A \leftarrow \alpha \bullet B\beta, a$ de $K$, todos os itens da forma $B \leftarrow \bullet\gamma, b$, com $b \in \mathrm{FIRST}(\beta a)$ também estão em $K$.
- Dado um conjunto inicial de itens, o cálculo do seu *fecho* pode ser realizado de maneira iterativa, incluindo os itens pela regra acima, até que não haja mais inclusões possíveis.
- Um *estado* é um conjunto fechado de itens.



## Construção *LR(1)* (cont.)

- Se o estado corrente (topo da pilha) contém um item da forma $A \leftarrow \alpha \bullet X\beta, a$, e o próximo símbolo a ser consultado (entrada ou resultado da última redução) é $X$, então deve ser empilhado um novo estado que contenha o item $A \leftarrow \alpha X \bullet \beta, a$, indicando que foi reconhecido mais um símbolo do item original.
- Conseqüentemente, dado um estado $K$ e um símbolo $X$, a função $\mathrm{goto}(K, X)$ será dada por:

  $$\mathrm{goto}(K, X) = \mathrm{closure}(K')$$
  onde $$K' = \{A \leftarrow \alpha X \bullet \beta, a \mid \forall\, A \leftarrow \alpha \bullet X\beta, a \in K\}$$
- A função goto determina as entradas da tabela de análise.
- O estado inicial $e_0$ é criado, normalmente:
  - acrescentando-se a produção $S' \leftarrow S\#$ ($S$ é a raiz original da gramática e # denota o fim da entrada)
  - colocando $e_0 = \mathrm{closure}(\{S' \leftarrow \bullet S\#, \epsilon\})$
- A partir de $e_0$ são calculados os outros estados e a função goto até que não haja mais estados novos.



## Exemplo de cálculo da tabela *LR(1)*

- Gramática de expressões com prioridades:

  $0.\ E' \leftarrow E\#$ $1.\ E \leftarrow E + T$ $2.\ E \leftarrow T$ $3.\ T \leftarrow T * F$
  $4.\ T \leftarrow F$ $5.F \leftarrow (E)$ $6.F \leftarrow a$
- A construção *LR(1)* produz 22 estados. Alguns deles são:

  $e_0$ : $E' \leftarrow \bullet E\#, \epsilon$
  $E \leftarrow \bullet E + T, \# + \mid \bullet T, \# +$
  $T \leftarrow \bullet T * F, \# + * \mid \bullet F, \# + *$
  $F \leftarrow \bullet(E), \# + * \mid \bullet a, \# + *$
  $e_1$ : $E' \leftarrow E \bullet \#, \epsilon$
  $E \leftarrow E \bullet + T, \# +$
  $e_2$ : $E \leftarrow T \bullet, \# +$
  $T \leftarrow T \bullet * F, \# + *$
  $e_3$ : $T \leftarrow F \bullet, \# + *$
  $e_9$ : $E \leftarrow T \bullet, ) +$
  $T \leftarrow T \bullet * F, ) + *$
  $e_{10}$ : $T \leftarrow F \bullet, ) + *$



## Observações

- A construção *LR(1)* pode resultar em um número de estados muito maior que *LR(0)*.
- Nota-se, por exemplo, que os estados $e_2$ e $e_9$, da mesma maneira como $e_3$ e $e_{10}$, são muito semelhantes, com os mesmos *núcleos*, mas diferindo nos conjuntos de *símbolos de consulta*.
- A construção *LALR* (*Lookahead LR*) consiste em juntar estados com núcleos iguais, incluindo conjuntos de símbolos de consulta de ambos. Assim, $e_2$ e $e_9$ bem como $e_3$ e $e_{10}$ poderiam ser juntados para formar:

  $e_{2,9}$ : $E \leftarrow T \bullet, ) +$
  $T \leftarrow T \bullet * F, ) + *$
  $e_{3,10}$ : $T \leftarrow F \bullet, ) + *$

  Neste caso, a função goto deve ser ajustada de acordo.
- O resultado é um número de estados igual àquele fornecido por *LR(0)*.
- A técnica funciona muito bem na prática, mas pode introduzir conflitos do tipo *reduz/reduz* em casos muito especiais.
- Ao invés de passar pela construção *LR(1)*, é possível fazer diretamente a construção *LALR* tornando o processo muito mais eficiente.

## Utilização

- A análise *LR(1)* e *LALR(1)* é mais geral do que *LL(1)*.
- Dependendo da implementação, *LR(1)* pode ser mais rápida, mas este não é um aspecto muito importante.
- Na prática, é inviável realizar o cálculo das tabelas LALR manualmente.
- Existem várias ferramentas automáticas: Yacc, Bison e seus variantes.
- Estas ferramentas, juntamente com as de análise léxica (como *lex* e *flex*), permitem inserções de código para implementar as várias fases do compilador: análise semântica, geração de código, etc.