

Curso de Engenharia de Computação

ECM253 – Linguagens Formais, Autômatos e Compiladores

Análise Sintática Descendente



Análise sintática descendente

■ Conceitos

- A **análise** descendente ou **top-down** se **inicia na raiz**, que corresponde ao **símbolo inicial da gramática**, e então **sistematicamente estende** a árvore **para baixo até casar com as palavras classificadas** retornadas pelo **scanner**;
- **Em cada ponto** da análise, **escolhe-se** um **símbolo não terminal da barra da árvore** atual e então **amplia-se a árvore** com a **adição**, a este símbolo, **de filhos** que **correspondem ao lado direito de uma produção** cujo lado **esquerdo** é o **símbolo não terminal** escolhido;
- **O processo continua até que:**
 - (a) A **barra** da árvore **contém** somente **não terminais** e a **entrada terminou** – neste caso a análise terminou com sucesso;
 - (b) Ocorre um **descasamento** entre a **barra parcialmente construída** e o **fluxo de entrada** – neste caso o **analisador** pode ter **escolhido** alguma **regra errada** – se houver **regras** que ainda **não** foram **testadas**, o **analisador** pode tentá-las – **processo** de **backtracking**. **Senão**, é o caso de falha na análise e o analisador deve sinalizar um **erro de sintaxe**.
- A maioria das linguagens livres de contexto dispensam *backtracking*.

Análise sintática descendente

■ Conceitos

– Algoritmo para um analisador sintático *top-down*

```

1  root ← node for the start symbol, S;
2  focus ← root;
3  push(null);
4  word ← NextWord(); // NextWord() is a scanner function that returns a token
5  while(true) do:
6      if (focus is a nonterminal) then begin: //nonterminal case
7          pick next rule to expand focus A ( $A \rightarrow \beta_1\beta_2...\beta_n$ );
8          build nodes for  $\beta_1\beta_2...\beta_n$  as children of focus;
9          push( $\beta_n, \beta_{n-1}, \dots, \beta_2$ ); //note the inverse order
10         focus ←  $\beta_1$ ;
11     end;
12     else if (word matches focus) then begin: //terminal case
13         word ← NextWord();
14         focus ← pop();
15     end;
16     else if (word = eof and focus = null) then begin:
17         accept input and return root;
18     else
19         backtrack; //backtrack and try another rule
20     end;
21 end;
```

Algoritmo para um analisador sintático top-down

Exemplo

- **Considerar** a **gramática** de expressões a **seguir** (o símbolo *Goal* é o símbolo de partida – embora não necessário para a gramática em si, é importante para o algoritmo):

```

0   Goal  → Expr
1   Expr  → Expr '+' Term
2           | Expr '-' Term
3           | Term
4   Term  → Term '*' Factor
5           | Term '/' Factor
6           | Factor
7   Factor → '(' Expr ')'
8           | number
9           | id
    
```

Algoritmo para um analisador sintático top-down

Exemplo

- Simular a aplicação do algoritmo para a entrada **a+3**. O algoritmo não prescreve a ordem de escolha das produções, então serão selecionadas as regras que conduzem a um término com sucesso, evitando *backtracking* e recursões infinitas à esquerda. A pilha cresce da esquerda para a direita.

- Linhas 1, 2, 3, 4 - inicialização:

root	focus	stack	word
<i>Goal</i>	<i>Goal</i>	null	a

- Linhas 6–10 - pois focus não é terminal – usar regra 0:

root	focus	stack	word
<i>Goal</i>	<i>Expr</i>	null	a

- Linhas 6–10 - pois focus não é terminal – usar regra 1:

root	focus	stack	word
<i>Goal</i>	<i>Expr</i>	null, <i>Term</i> , '+'	a

Algoritmo para um analisador sintático top-down

Exemplo

Simulação – continuação.

- Linhas 6–10 - pois focus não é terminal – usar regra 3:

root	focus	stack	word
<i>Goal</i>	<i>Term</i>	null, <i>Term</i> , '+'	a

- Linhas 6–10 - pois focus não é terminal – usar regra 6:

root	focus	stack	word
<i>Goal</i>	<i>Factor</i>	null, <i>Term</i> , '+'	a

- Linhas 6–10 - pois focus não é terminal – usar regra 9:

root	focus	stack	word
<i>Goal</i>	id	null, <i>Term</i> , '+'	a

- Linhas 12–15 - pois focus é terminal:

root	focus	stack	word
<i>Goal</i>	'+'	null, <i>Term</i>	'+'

Algoritmo para um analisador sintático top-down

Exemplo

Simulação – continuação.

- Linhas 12–15 - pois focus é terminal:

root	focus	stack	word
<i>Goal</i>	<i>Term</i>	null	3

- Linhas 6–10 - pois focus não é terminal – usar regra 6:

root	focus	stack	word
<i>Goal</i>	<i>Factor</i>	null	3

- Linhas 6–10 - pois focus não é terminal – usar regra 9:

root	focus	stack	word
<i>Goal</i>	number	null	3

- Linhas 12–15 - pois focus é terminal:

root	focus	stack	word
<i>Goal</i>	null		eof

- Linhas 16–19 - pois focus é null e word é eof. Neste caso, o algoritmo termina com sucesso retornando *Goal* como indicação que a entrada a+3 foi analisada com sucesso – ela está sintaticamente correta de acordo com a gramática!

Eliminação de recursão à esquerda

■ Conceitos

- A **recursão à esquerda** é um **problema** de **análise descendente** em que o analisador pode **repetir indefinidamente** uma substituição **sem gerar um não terminal** que **case** com o símbolo de **entrada atual** e permita avançar na análise;
- Considerar a **gramática de expressões** a seguir:

$$Expr \rightarrow Expr '+' Term \mid Expr '-' Term \mid Term$$

$$Term \rightarrow Term '*' Factor \mid Term '/' Factor \mid Factor$$

$$Factor \rightarrow '(' Expr ')' \mid \text{number} \mid \text{id}$$

Para uma **entrada** como temp, nada impede que o analisador realize as seguintes derivações: $Expr \Rightarrow Expr '+' Term \Rightarrow Expr '+' Term '+' Term$ e assim indefinidamente (o correto seria: $Expr \Rightarrow Term \Rightarrow Factor \Rightarrow \text{id}$)

- O problema está na recursão à esquerda presente nas regras!

Eliminação de recursão direta à esquerda

- Produções do tipo:

$$\begin{array}{l} A \rightarrow A\alpha \\ | \beta \end{array}$$

Onde $A \in N$ e $\alpha, \beta \in (N \cup T)^+$. Neste caso, **reescrevem-se as regras adicionando um símbolo auxiliar A'** , assim:

$$\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \\ | \epsilon \end{array}$$

Notar que a **recursão à esquerda** foi **substituída** por uma **recursão à direita** equivalente.

Eliminação de recursão direta à esquerda

- **Exemplo.** Aplicando as regras de eliminação à gramática do slide 8, tem-se a seguinte gramática equivalente, porém sem recursão à esquerda:

```

1   $Expr \rightarrow Term\ Expr'$ 
2   $Expr' \rightarrow '+'\ Term\ Expr'$ 
3       $| '-'\ Term\ Expr'$ 
4       $| \epsilon$ 
5   $Term \rightarrow Factor\ Term'$ 
6   $Term' \rightarrow '*'\ Factor\ Term'$ 
7       $| '/'\ Factor\ Term'$ 
8       $| \epsilon$ 
9   $Factor \rightarrow '('\ Expr\ ')'$ 
10       $| number$ 
11       $| id$ 

```

Eliminação de recursão geral à esquerda

■ **Conceitos**

- Considerar as regras a seguir:

$$S \rightarrow Aa$$

$$| b$$

$$A \rightarrow Ac$$

$$| Sd$$

$$| \epsilon$$

Existe, além da recursão direta na regra $A \rightarrow Ac$, uma **recursão indireta à esquerda** a partir da regra S , que pode ser assim verificada:

$$S \Rightarrow Aa$$

$$\Rightarrow Sda$$

$$\Rightarrow Aada$$

...

Eliminação de recursão geral à esquerda

▪ Algoritmo de eliminação de recursão geral à esquerda

- Consiste em remover as recursões indiretas à esquerda, se existirem, e depois remover as recursões diretas à esquerda, se existirem.

```

1  impose an order on the nonterminals  $A_1, A_2, \dots, A_n$ ;
2  for  $i \leftarrow 1$  to  $n$  do begin:
3      for  $j \leftarrow 1$  to  $i-1$  do begin:
4          if exists a production  $A_i \rightarrow A_j \gamma$  then begin:
5              replace  $A_i \rightarrow A_j \gamma$  with one or more productions that expand  $A_j$ ,
6              i.e., if  $A_i \rightarrow A_j \gamma$  and  $A_j \rightarrow \delta_1 | \delta_2 | \dots | \delta_k$  then remove  $A_i \rightarrow A_j \gamma$  and
7              replace it with  $A_i \rightarrow \delta_1 \gamma | \delta_2 \gamma | \dots | \delta_k \gamma$ ;
8          end;
9      end;
10     // apply the rule for direct left recursion removal
11     rewrite the productions to eliminate any direct left recursion on  $A_i$ ;
12 end;
```

Remoção de recursão geral à esquerda

Exemplo

- A gramática a seguir possui recursões à esquerda:

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid \epsilon$$

- Considerando que a ordem imposta aos símbolos seja S e depois A , quando $i = 1$, o comando de repetição interno não é executado, bastando apenas eliminar as recursões à esquerda imediatas em relação à S , não alterando em nada a gramática apresentada;
- Quando $i = 2$, a repetição interna será executada. A produção $A \rightarrow Sd$ será substituída de acordo com as produções $S \rightarrow Aa \mid b$, e a gramática se tornará:

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

Remoção de recursão geral à esquerda

▪ Exemplo

- Por fim, removem-se todas as recursões imediatas à esquerda:

$$\begin{aligned} S &\rightarrow Aa \\ &\quad | b \\ A &\rightarrow bdA' \\ &\quad | A' \\ A' &\rightarrow cA' \\ &\quad | adA' \\ &\quad | \epsilon \end{aligned}$$

Análise sintática livre de backtracking

■ Conceitos

- Considerar a **execução** do **algoritmo** do slide 3 com a **entrada** $y*3+4$ e a **gramática** do slide 10. Aqui se acrescentou o não terminal *Goal*; os números sobre as flechas indicam as produções escolhidas e o símbolo '_' representa o símbolo de entrada a ser considerado:

Passo	Regra utilizada	Entrada	Passo	Regra utilizada	Entrada
1	$Goal \xRightarrow{0} Expr$	$y*3+4$	11	$\xRightarrow{2} id '*' number '+' Term Expr'$	$y*3+4$
2	$\xRightarrow{1} Term Expr'$	$y*3+4$	12	avança a entrada	$y*3+4$
3	$\xRightarrow{5} Factor Term' Expr'$	$y*3+4$	13	$\xRightarrow{5} id '*' number '+' Factor Term' Expr'$	$y*3+4$
4	$\xRightarrow{11} id Term' Expr'$	$y*3+4$	14	$\xRightarrow{10} id '*' number '+' number Term' Expr'$	$y*3+4$
5	avança a entrada	$y*3+4$	15	avança a entrada	$y*3+4$
6	$\xRightarrow{6} id '*' Factor Term' Expr'$	$y*3+4$	16	$\xRightarrow{8} id '*' number '+' number Expr'$	$y*3+4$
7	avança a entrada	$y*3+4$	17	$\xRightarrow{4} id '*' number '+' number$	$y*3+4$
8	$\xRightarrow{10} id '*' number Term' Expr'$	$y*3+4$			
9	avança a entrada	$y*3+4$			
10	$\xRightarrow{8} id '*' number Expr'$	$y*3+4$			

Análise sintática livre de backtracking

■ Conceitos

- O algoritmo de **análise sintática descendente** do slide 3 quando tem de escolher o **lado direito** de um **não terminal** A ele o faz de **modo aleatório**, com **possibilidade de erro** e ter de realizar posteriormente um procedimento de **backtracking**;
- É possível **eliminar** a possibilidade de **backtracking**. Utilizando o **exemplo** do **slide 15**, pode-se observar que:
 - No **passo 4**, tem-se o não terminal $Term'$ (**foco**) e o terminal $'*'$ na entrada (**word**). A questão é: **qual lado direito** de $Term'$ utilizar para tentar, futuramente, casar com o terminal $'*'$? As **opções** são: $'' Factor Term'$, $'/' Factor Term'$ e ϵ . Para se ter **certeza** de qual substituição escolher, basta determinar os **conjuntos** de **palavras** terminais que **iniciam** cada uma dessas **opções**. Esses **conjuntos** são denominados de conjuntos *FIRST*. Então, **escolhe-se** como substituição o **lado direito** cujo *FIRST* contém a palavra da entrada atual (word) que, neste caso, é $'' Factor Term'$, pois $word \in FIRST('' Factor Term')$;

Análise sintática livre de backtracking

■ Conceitos

- Ainda utilizando o **exemplo** do **slide 15**, também pode-se observar que:
 - No **passo 9** do slide 15, após ter avançado na entrada, deve-se escolher um lado direito pra **substituir o não terminal** $Term'$ novamente. As **opções** são: $'* Factor Term'$, $'/' Factor Term'$ e ϵ . No entanto, a palavra atual (word) é '+', que não pertence a nenhum conjunto $FIRST$ do lado direito de $Term'$. Neste caso, quando não há alternativa, escolhe-se a produção vazia (ϵ). Mas, para diferenciar entre entradas legais e erros de sintaxe, o analisador precisa saber que palavras podem aparecer logo após a substituição com ϵ . Neste caso, **para se tomar a próxima decisão**, é necessário calcular o conjunto $FOLLOW$ de $Term'$. Ao se inspecionar a gramática, percebe-se que os terminais que seguem após o símbolo $Term'$ são eof, '+', '- ' e ') '. Logo, sabendo que word é '+' e, portanto, pertence à $FOLLOW(Term')$, então escolhe-se o lado direito $'+' Term Expr'$ de $Expr'$.
- Uma gramática livre de *backtracking* é também denominada de **gramática preditiva**;
- Uma gramática livre de *backtracking* **depende** da definição dos **conjuntos** $FIRST$ e $FOLLOW$.

Conjuntos *FIRST* e *FOLLOW*

■ Conceitos

- São **utilizados** em **análise sintática descendente** e **ascendente**;
- $FIRST(\alpha)$ é uma **função** que **retorna** um **conjunto** de **palavras** (terminais) que podem **aparecer** como a **primeira palavra** em alguma **cadeia derivada** de α .
- O **domínio** de $FIRST$ é o conjunto de **símbolos** da **gramática**, $T \cup NT \cup \{\epsilon, eof\}$ e seu **contradomínio** é $T \cup \{\epsilon, eof\}$. Se a é um **terminal qualquer** ou ϵ ou eof , então $FIRST(a)$ tem **exatamente um membro**, que é o próprio a ;
- Para um **não terminal** A , $FIRST(A)$ contém o **conjunto completo** de palavras (terminais) que podem aparecer como **palavra inicial** em uma **forma sentencial derivada** de A ;
- Se $FIRST(A)$, $A \in NT$, inclui ϵ , então $FIRST(\epsilon) = \epsilon$ **não corresponde a nenhuma palavra retornada pelo analisador léxico**. Desse modo, para diferenciar entre entradas legais e erros de sintaxe, o **analisador sintático precisa saber quais símbolos iniciais podem aparecer após A nesta situação**, que é precisamente o conteúdo do **conjunto *FOLLOW***;
- Então a **função *FOLLOW*(A)** **calcula**, para um **não terminal** A , o **conjunto de palavras** que podem **ocorrer imediatamente após A** em uma sentença.

Conjuntos *FIRST* e *FOLLOW*

■ Algoritmo para calcular *FIRST*

```

1  for each  $\alpha \in T \cup \{\text{eof}, \epsilon\}$  do:
2       $FIRST(\alpha) \leftarrow \alpha$ ;
3  end;
4  for each  $A \in NT$  do:
5       $FIRST(A) \leftarrow \emptyset$ ;
6  end;
7  while (FIRST are still changing) do:
8      for each  $p \in P$ , where  $p$  has the form  $A \rightarrow \beta$  and  $\beta$  is  $\beta_1\beta_2...\beta_k$  do:
9           $rhs \leftarrow FIRST(\beta_1) - \{\epsilon\}$ ;
10          $i \leftarrow 1$ ;
11         while ( $\epsilon \in FIRST(\beta_i)$  e  $i \leq k-1$ ) do:
12              $rhs \leftarrow rhs \cup (FIRST(\beta_{i+1}) - \{\epsilon\})$ ;
13              $i \leftarrow i+1$ ;
14         end;
15         if  $i = k$  and  $\epsilon \in FIRST(\beta_k)$  then
16              $rhs \leftarrow rhs \cup \{\epsilon\}$ ;
17         end;
18          $FIRST(A) \leftarrow FIRST(A) \cup rhs$ ;
19     end;
20 end;
```

Conjuntos *FIRST* e *FOLLOW*

- **Exemplo de cálculo de *FIRST*** Quando se aplica o algoritmo para calcular os conjuntos *FIRST* à gramática do slide 10, tem-se o seguinte resultado (após várias folhas de papel para simular...):

	number	id	'+'	'-'	'*'	'/'	'('	')'	eof	ϵ
<i>FIRST</i>	{number}	{id}	{'+'}	{'-'}	{'*'}	{ '/' }	{ '(' }	{ ')' }	{eof}	{ ϵ }

	<i>Expr</i>	<i>Expr'</i>	<i>Term</i>	<i>Term'</i>	<i>Factor</i>
<i>FIRST</i>	{'(', id, number}	{'+', '-', ϵ }	{'(', id, number}	{ '*', '/', ϵ }	{'(', id, number}

Conjuntos *FIRST* e *FOLLOW*

■ Algoritmo para calcular *FOLLOW*

```

1  for each  $A \in NT$  do
2       $FOLLOW(A) \leftarrow \emptyset$ ;
3  end;
4   $FOLLOW(S) \leftarrow \{eof\}$ ; //S is the initial symbol - add it if it does not exist
5  while (FOLLOW sets are still changing) do:
6      for each  $p \in P$ , of the form  $A \rightarrow \beta_1\beta_2...\beta_k$  do:
7           $TRAILER \leftarrow FOLLOW(A)$ ;
8          for  $i \leftarrow k$  downto 1 do:
9              if  $\beta_i \in NT$  then begin:
10                  $FOLLOW(\beta_i) \leftarrow FOLLOW(\beta_i) \cup TRAILER$ ;
11                 if  $\epsilon \in FIRST(\beta_i)$  then begin:
12                      $TRAILER \leftarrow TRAILER \cup (FIRST(\beta_i) - \{\epsilon\})$ ;
13                 end
14                 else begin:
15                      $TRAILER \leftarrow FIRST(\beta_i)$ ;
16                 end;
17             end;
18             else begin:
19                  $TRAILER \leftarrow FIRST(\beta_i)$ ;
20             end;
21         end;
22     end;
23 end;
```

Conjuntos *FIRST* e *FOLLOW*

- **Exemplo de cálculo de *FOLLOW*** Quando se aplica o algoritmo para calcular os conjuntos *FOLLOW* à gramática do slide 10, tem-se o seguinte resultado:

	<i>Expr</i>	<i>Expr'</i>	<i>Term</i>	<i>Term'</i>	<i>Factor</i>
<i>FIRST</i>	{eof, ')')}	{eof, ')')}	{eof, '+', '-', ')')}	{eof, '+', '-', ')')}	{eof, '+', '-', '*', '/', ')')}

Conjunto $FIRST^+$

■ Conceitos

- O conjunto $FIRST^+$ é o **conjunto $FIRST$ aumentado**, assim definido:

$$FIRST^+(A \rightarrow \beta) = \begin{cases} FIRST(\beta), & \text{se } \epsilon \notin FIRST(\beta) \\ FIRST(\beta) \cup FOLLOW(A), & \text{caso contrário} \end{cases}$$

- Notar que $FIRST^+$ é **definido** a regras completas ($A \rightarrow \beta$);
- Ele é utilizado para **caracterizar uma gramática livre de retrocesso** (*backtracking*). A **propriedade** que indica se uma gramática é livre de retrocesso **indica** que, para qualquer **não terminal** A com **múltiplos lados direitos**, $A \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$, deve-se ter que:

$$FIRST^+(A \rightarrow \beta_i) \cap FIRST^+(A \rightarrow \beta_j) = \emptyset, \forall (1 \leq i, j \leq n, i \neq j)$$

Eliminando backtracking

- Vamos ampliar a gramática do slide 9, com regras para chamada de função e referência de elemento indexado. Estas regras não estão livres de *backtracking*:

$$\begin{aligned}
 Factor &\rightarrow '(' Expr ')' \\
 &\quad | \text{ number} \\
 &\quad | \text{ id} \\
 &\quad | \text{ id '[' ArgList ']' } \\
 &\quad | \text{ id '(' ArgList ')' } \\
 ArgList &\rightarrow Expr MoreArgs \\
 MoreArgs &\rightarrow ',' Expr MoreArgs \\
 &\quad | \epsilon
 \end{aligned}$$

Eliminando *backtracking*

- Para remover a necessidade de *backtracking*, pode-se reescrevê-la assim (fatoração à esquerda):

$$\begin{aligned} Factor &\rightarrow \text{id } Arguments \\ Arguments &\rightarrow '[' ArgList ']' \\ &\quad | '(' ArgList ')' \\ &\quad | \epsilon \end{aligned}$$

- Forma geral de fatoração à esquerda**

- Se uma regra A possui o formato $A \rightarrow \alpha\beta_1|\alpha\beta_2|\dots|\alpha\beta_n|\gamma_1|\gamma_2|\dots|\gamma_j$ em que α é um prefixo comum e γ_i são lados direitos de A que não possuem um prefixo comum, então esta regra pode ser substituída pelas regras as seguir:

$$\begin{aligned} A &\rightarrow \alpha B|\gamma_1|\gamma_2|\dots|\gamma_j \\ B &\rightarrow \beta_1|\beta_2|\dots|\beta_n \end{aligned}$$

Métodos de análise sintática descendente

■ Análise sintática recursiva descendente

- O analisador é estruturado como um conjunto de **procedimentos mutuamente recursivos**, um **para cada não terminal na gramática**;
- O **procedimento correspondente ao não terminal A** reconhece uma **ocorrência de A** no fluxo de entrada;
- Para **reconhecer um não terminal B** em algum **lado direito de A**, o analisador chama o **procedimento correspondente a B**;
- Em outras palavras, a **própria gramática** serve como um **guia** para a **implementação do analisador**;
- Para a **estruturação do analisador sintático** é empregada a **metalinguagem EBNF**, pois **elimina as recursões infinitas e não-determinismo** na construção do analisador.

Análise sintática recursiva descendente

■ Exemplo

- Considerar as seguintes **regras EBNF** para expressões aritméticas simples:

```
expr = termo,{soma,termo};  
soma = '+' | '-';  
termo = fator,{mult,termo};  
mult = '*';  
fator = '(',exp,')' | num
```

Análise sintática recursiva descendente

■ Exemplo em C

- Este é um exemplo bem simples cujo foco é a análise sintática – o analisador léxico está implementado diretamente no programa e o semântico é o cálculo das operações quando a sintaxe está correta.
- **Definições de variáveis e protótipos de funções**

```
/*Armazena marca*/  
char marca;  
/*Protótipos de funções*/  
/*Para análise sintática*/  
int expr();  
int termo();  
int fator();  
/*Funções de apoio*/  
void casar(char);  
void erro(void);
```

Análise sintática recursiva descendente

- Exemplo em C

- Programa principal

```
int main(void)
{
    int resultado;
    /* armazene a primeira marca */
    marca = getchar();
    /* É aqui que começa a análise: <expr> */
    resultado = expr();
    if(marca=='\n')
        printf("Resultado = %d\n", resultado);
    return 0;
}
```

Análise sintática recursiva descendente

Exemplo em C

– Analisando expressões

```
int expr(void){
    /*<expr>:=<termo>{<soma><termo>}*/
    int temp = termo();
    /*casar com {<soma><termo>}*/
    while((marca=='+')||(marca=='-'))
        switch(marca){
            case '+':
                casar('+');
                temp += termo();
            break;
            case '-':
                casar('-');
                temp -= termo();
            break;
        }
    return temp;
}
```

Análise sintática recursiva descendente

- Exemplo em C
 - Analisando termos

```
int termo(void)
{
    /*<termo>::=<fator>{<mult><fator>}*/
    int temp=fator();
    while (marca=='*')
    {
        casar('*');
        temp *= fator();
    }
    return temp;
}
```

Análise sintática recursiva descendente

- Exemplo em C
 - Analisando fatores

```
int fator(void){
    int temp;
    /*<fator>:=(expr)|num*/
    if (marca=='('){
        casar('(');
        temp=expr();
        casar(')');
    }
    else {
        if(isdigit(marca)){
            /*devolve a marca para a entrada*/
            ungetc(marca,stdin);
            /*para então ler o número com scanf*/
            scanf("%d", &temp);
            marca=getchar();
        }
        else
            erro();
    }
    return temp;
}
```


Análise sintática recursiva descendente

- Exemplo em C
 - Funções de apoio

```
/* Exibir mensagem de erro */
void erro(void)
{
    fprintf(stderr, "Erro!!!\n");
}

/* verifica se a marca casa com a esperada */
void casar(char marcaEsperada)
{
    /*se a marca é a esperada, leia o próximo*/
    if(marca == marcaEsperada)
        marca = getchar();
    else
        erro();
}
```

Métodos de análise sintática descendente

■ Análise sintática descendente dirigido por tabela

- É possível **gerar um analisador sintático descendente** a partir de uma **gramática livre de *backtracking*** e os **conjuntos $FIRST$, $FOLLOW$ e $FIRST^+$** ;
- Os **conjuntos $FIRST^+$** **orientam** completamente as **decisões do analisador**;
- Assim, com a **presença de pelo menos um símbolo de *lookahead*** e um **não terminal em foco**, é **automática a seleção de outra produção** – e esse mecanismo é **codificado** em uma **tabela** (como em tabela de estados);
- Os **analisadores** que seguem **este método** são **conhecidos genericamente** como **$LL(k)$** , sendo o **$LL(1)$** a abordagem mais comum.

Análise sintática descendente dirigido por tabela

- Um analisador sintático **LL(k)** é aquele que **processa a cadeia de entrada da esquerda para a direita (L)** e **constrói uma derivação mais à esquerda (L)** da sentença.
- O **símbolo k** na sua denominação indica que o analisador em questão necessitará de **k símbolos de verificação** para efetuar a análise;
- **Analisadores LL(k)** realizam o processo de **análise sintática** de modo **descendente**: verificam a sintaxe da cadeia de entrada construindo uma árvore sintática do símbolo não-terminal de início (raiz) até os símbolos terminais (folhas);
- Um **analisador sintático LL(1)** utiliza um **método de análise determinístico, descendente** e utilizando um **único símbolo de verificação**.

Análise sintática LL(1)

■ Tabela LL(1)

- É utilizada para guiar a decisão do analisador quando o foco é um símbolo não terminal e deseja-se escolher uma regra correta em conjunto com o símbolo de *lookahead*; **Algoritmo para construir a tabela LL(1) de análise**

```

Build FIRST, FOLLOW and FIRST+ sets;
for each nonterminal A do:
  for each terminal w do:
    Table[A,w] ← error;
  end;
  for each production p of the form A → β do:
    for each terminal w ∈ FIRST+(A → β) do:
      Table[A,w] ← p;
    end;
    if eof ∈ FIRST+(A → β)
      then Table[A,eof] ← p
  end;
end;

```

Análise sintática LL(1)

■ Algoritmo LL(1)

```

word ← NextWord(); // call the scanner
push eof onto Stack;
push the start symbol, S, onto Stack;
focus ← top of Stack;
loop forever:
  if(focus = eof and word = eof) // ended successfully
    then report success and exit the loop;
  else if (focus ∈ T or focus = eof ) then begin: // focus is a terminal
    if (focus matches word) then begin:
      pop Stack;
      word ← NextWord();
    end;
    else report an error looking for symbol at top of stack;
  end;
  else begin: // focus is a nonterminal
    if Table[focus, word] is  $A \rightarrow \beta_1\beta_2...\beta_k$  then begin:
      pop Stack;
      for  $i \leftarrow k$  to 1 by -1 do:
        if ( $\beta_i \neq \epsilon$ )
          then push  $\beta_i$  onto Stack;
      end;
    end;
    else report an error expanding focus;
  end;
  focus ← top of Stack;
end;

```

Análise sintática LL(1)

Exemplo

- Reescrevendo a gramática do slide 10, e enumerando as regras, tem-se:

```

0   Goal  → Expr
1   Expr  → Term Expr'
2   Expr' → '+' Term Expr'
3           | '-' Term Expr'
4           | ε
5   Term  → Factor Term'
6   Term' → '*' Factor Term'
7           | '/' Factor Term'
8           | ε
9   Factor → '(' Expr ')'
10          | number
11          | id

```

Análise sintática LL(1)

Exemplo

- A tabela LL(1) para a gramática do slide 38 é:

	eof	+	-	*	/	()	id	number
<i>Goal</i>	–	–	–	–	–	0	–	0	0
<i>Expr</i>	–	–	–	–	–	1	–	1	1
<i>Expr'</i>	4	2	3	–	–	–	4	–	–
<i>Term</i>	–	–	–	–	–	5	–	5	5
<i>Term'</i>	8	8	8	6	7	–	8	–	–
<i>Factor</i>	–	–	–	–	–	9	–	11	10

- Simular com o algoritmo LL(1) a expressão: $a+b*c$ (tem o desenvolvimento completo em “Construindo Compiladores”).

Referências bibliográficas

AHO, A. V.; SETHI, R.; LAM, M. S. **Compiladores: princípios, técnicas e ferramentas**. 2. ed. [s.l.] Pearson, 2007.

COOPER, K.; TORCZON, L. **Construindo compiladores**. 2. ed. Rio de Janeiro: Elsevier, 2014.

LOUDEN, K. C. **Compiladores: princípios e práticas**. [s.l.] Pioneira Thomson Learning, 2004.