MC910 - Implementação de Linguagens de Programação
Análise Sintática - Parte I

A Análise Sintática pode ser feita de dois modos:

- *Ascendente:* a árvore de derivação é montada de cima para baixo. Começamos com o símbolo inicial da gramática na raiz, e expandimos nós com não-terminais até que todas as folhas contenham apenas terminais.
- *Descendente:* a árvore de derivação é montada de baixo para cima. Lemos os símbolos, e em determinados momentos, notamos que é possível fazer uma *redução* (o contrário da derivação), e então substituímos parte da cadeia de entrada por uma árvore. No final da análise, teremos uma árvore de derivação completa.

A análise descendente é normalmente implementada como uma série de procedimentos recursivos. Programar um analisador descendente recursivo é mais intuitivo e fácil do que um analisador ascendente, mas os analisadores descendentes aceitam menos gramáticas que os ascendentes. Além disso, a confecção de um analisador ascendente pode se tornar muito fácil se usarmos uma ferramenta geradora de analisador sintático como o CUP ou o Bison.

1 Análise Sintática Descendente

O algoritmo 1 mostra uma maneira de fazer a análise sintática descendente.

O algoritmo 1 pode consultar uma tabela para saber qual produção aplicar em cada situação.

Veja a gramática a seguir:

1 $S ::= (A, B)$
2 $S ::= A$
3 $S ::= < S; S >$
4 $A ::= a$
5 $A ::= x$
6 $B ::= b$
7 $B ::= y$

Algoritmo 1 Análise Sintática Descendente

$\alpha \leftarrow$ cadeia de entrada;
$D \leftarrow S$; {D será a árvore de derivação}
folha_corrente $\leftarrow D$;
while $\exists$ folha com não-terminal **do**
  $X \leftarrow$ folha_corrente;
  if X é não terminal **then**
    Escolha produção $X ::= X_1X_2X_3\cdots$;
    Os X_i agora serão folhas (filhas de X);
    folha_corrente $\leftarrow X_1$
  else if $\alpha = X\beta$ **then**
    $\alpha = \beta$;
    folha_corrente $\leftarrow$ próx. folha;
  else
    Retrocesso: desfaça a última produção;
  end if
end while
if $\alpha = \lambda$ **then**
  A cadeia foi aceita;
else
  A cadeia não foi aceita;
end if

Faremos a análise da sentença $< x; (a, y) >$

Começamos com o símbolo inicial apenas na árvore, e tentaremos aplicar as regras de derivação até terminarmos a árvore.

Entrada (restante): $< x; (a, y) >$
Árvore: $\boxed{\text{S}}$

A folha corrente é um não-terminal, portanto aplicaremos uma regra (usaremos a número 2) para expandir esta folha.

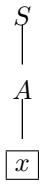
Entrada (restante): $< x; (a, y) >$

Árvore:
S
|
$\boxed{A}$

Agora a folha corrente é o não-terminal A. Usaremos a regra 5:

Entrada (restante): $< x; (a, y) >$

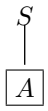
Árvore:



Agora a folha corrente é um terminal. Devemos verificar, então, se a cadeia de entrada começa com este terminal. (No algoritmo, se a folha corrente X for terminal, verificamos se a entrada é $X\beta$). Como a entrada não começa com X, teremos que retroceder (desfazer a última derivação). Voltamos à árvore:

Entrada (restante): $< x; (a, y) >$

Árvore:



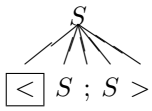
Tentaremos também expandr A usando a regra 4, mas a cadeia de entrada não começa com a. Como não há mais opções para expandir o não-terminal A, teremos que desfazer a derivação anterior ($S \Rightarrow A$).

Entrada (restante): $< x; (a, y) >$
Árvore: S

Agora, tentaremos derivr usando a regra 3:

Entrada (restante): $< x; (a, y) >$

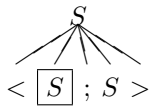
Árvore:



A folha corrente é um terminal ("<"). E a entrada começa com este mesmo terminal! Assim, podemos consumir este símbolo da entrada e tomar a próxima folha como corrente:

Entrada (restante): $< x; (a, y) >$

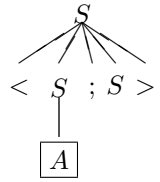
Árvore:



Temos um não-terminal, e aplicamos a regra 2:

Entrada (restante): $x; (a, y) >$

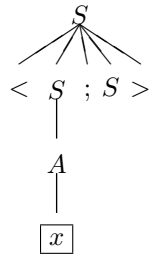
Árvore:



Novamente, a folha corrente é um não-terminal. Aplicamos então a regra 5.

Entrada (restante): $x; (a, y) >$

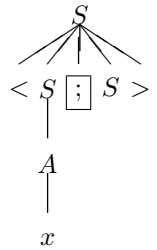
Árvore:



E temos o terminal x na folha corrente. A entrada começa com x, portanto seguimos adiante depois de retirar este símbolo da entrada. Agora, a folha corrente passa a ser aquela com o ;.

Entrada (restante): $; (a, y) >$

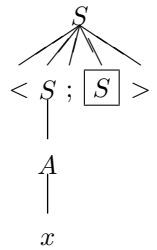
Árvore:



Novamente, o símbolo da entrada é o mesmo da folha corrente. No próximo passo, a folha corrente passa a ser o próximo S.

Entrada (restante): $(a, y) >$

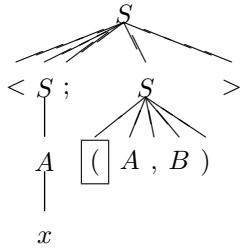
Árvore:



E escolhemos outra produção - a de número 1.

Entrada (restante): $(a, y) >$

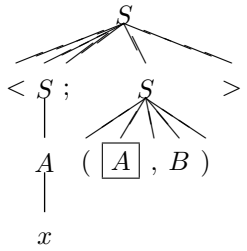
Árvore:



O próximo símbolo é um "(", então prosseguimos:

Entrada (restante): $a, y) >$

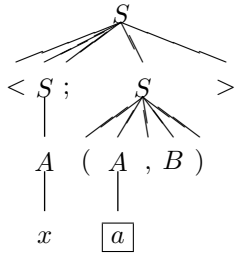
Árvore:



Agora, temos um não-terminal. Escolhemos a produção 4:

Entrada (restante): $a, y) >$

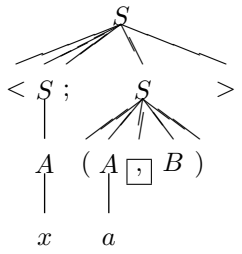
Árvore:



E como a é o próximo símbolo, continuamos.

Entrada (restante): $, y) >$

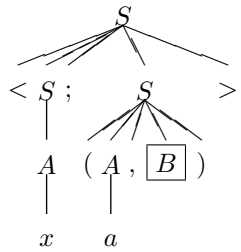
Árvore:



A vírgula na árvore corresponde ao próximo símbolo da entrada; passamos para a próxima folha, então.

Entrada (restante): $y) >$

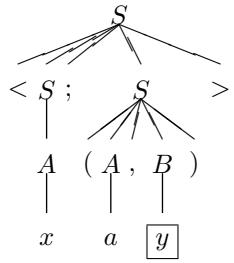
Árvore:



Temos um não-terminal. Escolheremos a produção número 7.

Entrada (restante): $y) >$

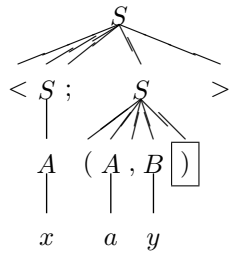
Árvore:



Como o y corresponde ao próximo símbolo, prosseguimos:

Entrada (restante): $) >$

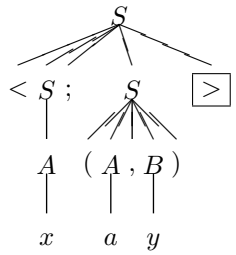
Árvore:



Agora,) é o próximo símbolo. Novamente, seguimos adiante:

Entrada (restante): $>$

Árvore:



Como $>$ é o próximo símbolo, continuamos. Mas como não há mais folhas

a se visitar e a cadeia de entrada está vazia, aceitamos a cadeia.

Note que erramos na escolha da primeira produção, e tivemos que retroceder. Na verdade, poderíamos ter escolhido a produção errada em vários outros pontos. Um algoritmo que funcione desta forma será muito ineficiente; veremos como resolver o problema nas próximas seções.

1.1 Notação

Os seguintes conceitos serão usados na discussão sobre gramáticas LL(1):

$\psi(X)$ O conjunto de todos os símbolos terminais que podem estar *no início* de uma forma sentencial derivada de X é denotado "$\psi(X)$"

$\Delta(X)$ O conjunto de todos os símbolos terminais que podem *seguir* uma forma sentencial derivada de X é denotado "$\Delta(X)$"

Recursão Uma produção da forma $A ::= A\gamma$ é recursiva à esquerda. Já a produção $A ::= \gamma A$ é recursiva à direita. Estes casos são de recursão *direta*, porque com uma única produção já percebemos a recursão. Se $A \overset{+}{\underset{e}{\Rightarrow}} A\gamma$, também existe a recursão, mas não é direta.

1.2 Gramáticas Aceitas

Queremos que nosso algoritmo sempre escolha a produção correta para um não-terminal, mas para isso teremos que restringir o tipo de gramática que podemos usar.

Uma gramática que apresente um dos seguintes problemas não pode ser analisada pelo nosso algoritmo.

- *Recursão Esquerda*
  Produções do tipo $A ::= A\gamma$ trarão problemas ao algoritmo. Suponha que a folha corrente é o não terminal A, e que temos as produções $A ::= Ay|x$. Ao escolhermos a produção $A ::= Ay$, a nova folha corrente será outro A. O algoritmo aplicará a mesma regra novamente, e entrará em um loop infinito.

  Note que isto se aplica também a derivações do tipo $A \overset{+}{\Rightarrow} A\alpha$, pois o algoritmo poderia alternar algumas produções indefinidamente. Exemplo:
  $A ::= Bx$
  $B ::= Ay$

  $A ::= \Rightarrow Bx \Rightarrow Ayx \Rightarrow Bxyx$

  O algoritmo alternaria A e B como folha corrente.

  A recursão direita já não é um problema para nosso algoritmo. Se tivermos $A ::= xA|yA$, sempre saberemos qual produção usar.

- *Produções iniciando com o mesmo terminal*
  Produções do tipo $A ::= x\beta | x\gamma$ são problemáticas, porque nosso algoritmo, mesmo sabendo que x é o próximo símbolo da entrada, não saberá qual das duas regras aplicar. Um exemplo muito claro disso é o do if-then-else:
  cmd ::= **if** expr **then** cmd **else** cmd
  cmd ::= **if** expr **then** cmd

Poderíamos resolver os dois problemas definindo as seguintes regras para nossas gramáticas:

- O lado direito de uma produção não pode começar com um não-terminal. (Resolvemos o promeiro problema)
- Se um não terminal pode derivar mais de uma forma sentencial, então elas devem começar com terminais diferentes entre si. (Resolvemos o segundo problema)

Mas infelizmente, o sonjunto de gramáticas que satisfaz estas regras é muito reduzido. Um exemplo é a seguinte gramática:

$E ::= +EE$
$E ::= *EE$
$E ::= a$
$E ::= b$

1.2.1 Gramáticas LL(1)

Os dois problemas anteriores parecem estar relacionados: não queremos ter produções para o mesmo não-terminal que derivem formas sentenciais começando com o mesmo símbolo. Podemos, então, usar a seguinte regra (informalmente): nossas gramáticas não podem ter produções $A ::= X\gamma$ e $A ::= Y\beta$ se houver alguma possibilidade de X e Y derivarem o mesmo símbolo (tanto terminal como não terminal).

Formalmente:

Se $A ::= X_1\alpha_1 | X_2\alpha_2 | ... | X_n\alpha_n$ então $\forall i, j \leq n, \psi(X_i) \cap \psi(X_j) = \emptyset$

Uma gramática que satisfaça esta exigência é chamada de gramática LL(1), que significa "**L**eft-to-right parsing, producing the **L**eftmost derivation, with **1** lookahead symbol". Isto porque uma propriedade importante é que toda gramática LL(1) pode ser analisada de forma descendente (com lookahead igual a 1).

Outra propriedade muito importante das gramáticas LL(1) é que elas não são ambíguas.

Veremos agora como transformar uma gramática em LL(1), resolvendo os dois problemas mostrados anteriormente.

1.2.2 Eliminando a Recursão Esquerda

Usaremos a seguinte notação: $\{\alpha\}$ é α, 0 ou mais vezes.

Considere as seguintes produções:

$$A ::= \delta_1|\delta_2|\delta_3|A\xi$$

Elas dizem basicamente que A pode ser expandido para um dos δ_i concatenado com uma ou mais ocorrências de ξ. Podemos então reescrever este tipo de produção da seguinte forma:

$$A ::= (\delta_1|\delta_2|\delta_3)\{\xi\}$$

Com a condição de que A nunca pode ser seguido de $\gamma \in \psi(\xi)$:

$$\Delta(A) \cap \psi_p^*(\xi) = \emptyset$$

A recursão esquerda foi eliminada, e o algoritmo sempre saberá que produção aplicar.

Exemplo:

$$E ::= E + T|T$$

Se tornaria

$$E ::= T\{+T\}$$

Poderíamos ter tenatado resolver o problema acima invertendo a ordem dos símbolos no lado direito da produção:

$$E ::= T + E|T$$

Mas neste caso teríamos duas produções para E derivando formas sentenciais começando com T. Poderíamos escrever esta produção assim:

$$E ::= T[+E]$$

Onde os colchetes indicam que aquela parte da produção pode ou não ser usada. Na prática, isto funciona muito bem, porque o símbolo "+" é um terminal, e o algoritmo pode olhar para ele e decidir imediatamente se vai usar a segunda parte da produção ou não. O primeiro exemplo de analisador descendente recursivo, mais à frente, fará exatamente isso.

1.2.3 Fatorando a Gramática

Se houver produções para um não-terminal começando com o mesmo terminal, faremos a fatoraçõ da gramática.

Seja β uma forma sentencial. Quando encontrarmos produções da forma:

$A ::= \beta\gamma_1|\beta\gamma_2|\beta\gamma_3|... \quad (\beta \neq \lambda)$

Reescreveremos a produção da seguinte maneira:

$A ::= \beta(\gamma_1|\gamma_2|\gamma_3|...)$

Usaremos como exemplo a especificação do comando if:

cmd ::= **if** expr **then** cmd **else** cmd
cmd ::= **if** expr **then** cmd

Neste caso, o equivalente a β é "**if** expr **then** cmd", γ_1 seria "**else** cmd", e γ_2 é λ. As produções tornam-se:

cmd ::= **if** expr **then** cmd cmd-else
cmd-else ::= **else** cmd $|\lambda$

E está eliminado o problema. Veja que sempre será possível saber que produção aplicar.

1.3 Analisador descendente recursivo

Vimos no início do texto o algoritmo iterativo para análise descendente. Apesar de iterativo, ele claramente realiza a expansõ da árvore sintática de forma recursiva. Uma maneira muito simples e intuitiva de implementar a análise descendente é usando um conjunto de procedimentos recursivos (e mutuamente recursivos, também). Usa-se um procedimento para cada regra da gramática.

Por exemplo, para a regra $E ::= T + E$, o seguinte procedimento pode ser usado:

```
E () {
     T ();
     if (prox != '+')
          erro ();
     else
          consome ();
     E ();
}
```

Mas e se tivermos $E ::= T + E|T$ (ou $E ::= T\{+T\}$)? a função fica assim:

```
E () {
     T ();
     if (prox == '+') {
          consome ();
          E ();
```

```
    }
}
```

O programa acima montará a árvore para a regra $E ::= T + E|T$. Se quisermos usar $E ::= T\{+T\}$, faríamos:

```
E () {
    T ();
    while (prox == '+') {
        consome ();
        T ();
    }
}
```

Note que nestes trechos de programa o próximo símbolo da entrada sempre é visível (está na variável "prox"), e a função "consome" retira o primeiro símbolo da entrada para verificar o próximo (faz o algoritmo avançar na cadeia de entrada).

Veja a gramática a seguir:

CMD ::= ID $<-$ EXP
CMD ::= while EXP-B CMD
CMD ::= **if** EXP-B **then** CMD ELSE
ELSE ::= **else** CMD | λ
EXP-B ::= **true** | **false** | EXP-B OP-B EXP-B | (EXP-B) | **not** EXP-B

O seguinte programa C (incompleto) reconhece esta gramática:

```
cmd () {
        switch (prox) {
        case IF:
                exp_b ();
                consome ();
                if (prox != THEN)
                        erro_sint ();
                cmd ();
                _else ();
                break;
        case ID:
                if (prox == '<')
                        consome ();
                        if (prox == '-')
                                exp ();
                        else
                                erro_sint ();
                break;
        case WHILE:
```

```
                exp_b ();
                cmd ();
        }
}

_else () {
        if (prox == ELSE) {
                consome ();
                cmd ();
        }
}

exp_b () {
        switch (prox) {
        case TRUE:
        case FALSE:
                consome ();
                break;
        case NOT:
                consome ();
                exp_b ();
                break;
        case '(':
                consome ();
                exp_b ();
                if (prox == ')')
                        consome ();
                else
                        erro_sint ();
                break;
        default:
                exp_b ();
                op_b ();
                exp_b ();
                break;
        }
}

op_b () {
        switch (prox) {
                case '+': case '-':
                case '/': case '*':
                        consome ();
                        break;
                default:
                        erro_sint ();
```

```
            }
        }
```