Licenciatura em Engenharia Informática – DEI/ISEP
Linguagens e Programação 2021/2022
Aula Teórico-Prática

Ficha TP 2

Gramáticas

Objetivos:

- Introdução aos conceitos e notações relacionados com gramáticas;
- Árvores de derivação;
- O problema da ambiguidade e sua resolução;
- Conversão entre autómatos finitos e gramáticas;
- *Parser* em descida recursiva.

1 Gramáticas

Uma gramática é uma ferramenta poderosa para a descrição e análise de linguagens. Uma gramática é constituída por um conjunto de regras segundo as quais as frases válidas da linguagem são construídas. Considere-se o seguinte excerto de uma gramática:

<**frase**> → <**sintagma-nominal**><**sintagma-verbal**>
<**sintagma-nominal**> → <**artigo**><**nome**>|<**nome**>
<**sintagma-verbal**> → <**verbo**><**sintagma-nominal**>
<**artigo**> → *o* | *a* | *os* | *as* |
<**nome**> → *Pedro* | *Maria* | *criança* | rapazes | *cartas* | *futebol*
<**verbo**> → *conhece* | *conhecem* | *é* | *são* | *joga* | *jogam*

Com estas regras, também designadas por produções, é possível analisar frases como as seguintes:

os rapazes jogam futebol.

o Pedro conhece a Maria.

a Maria é criança.

Considere-se a seguinte derivação à esquerda:

<**frase**> ⇒ <**sintagma-nominal**><**sintagma-verbal**>
⇒ <**artigo**><**nome**><**sintagma-verbal**>
⇒ *os* <**nome**><**sintagma-verbal**>
⇒ *os rapazes* <**sintagma-verbal**>
⇒ *os rapazes* <**verbo**><**sintagma-nominal**>

⇒ *os rapazes jogam* <**sintagma-nominal**>
⇒ *os rapazes jogam* <**nome**>
⇒ *os rapazes jogam futebol*

O processo de derivação pode ser representado graficamente através de uma árvore (de derivação) conforme a figura 2.1.

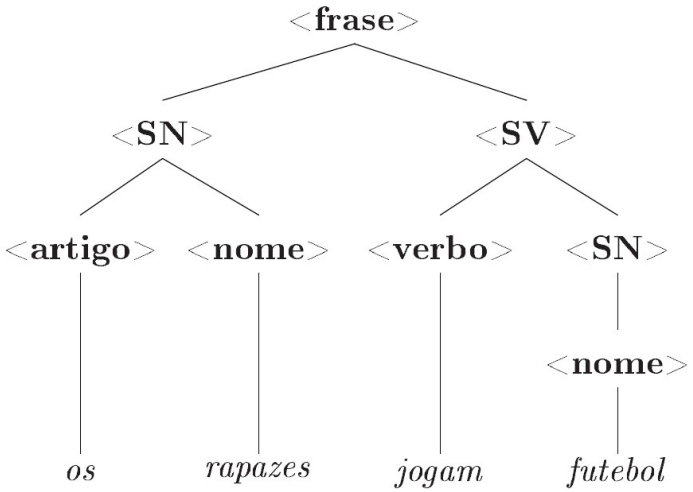


Figura 2.1: Árvore sintática

1.9 Definições e notação

Uma gramática é um conjunto de regras segundo o qual as frases válidas de uma linguagem são construídas. Formalmente, uma gramática é definida por um tuplo:

$$G = (V, \sum, P, S)$$

no qual:

- **V** - é um conjunto finito, não vazio, de **variáveis** (símbolos **não terminais**). Estes são os símbolos da gramática que podem ser substituídos por uma sequência de símbolos;
- **∑** - é um conjunto finito, não vazio, dito **alfabeto** ou conjunto de símbolos **terminais**. Estes são os símbolos da gramática que não podem ser substituídos, correspondem às palavras válidas na linguagem;
- **P –** é um conjunto de produções ou regras gramaticais. Uma regra gramatical define a forma como os símbolos não terminais podem ser substituídos. A sua forma geral é a seguinte:

  $$X \rightarrow Y_1Y_2Y_3...Y_n$$

  Neste exemplo o símbolo não terminal X é definido como equivalente à concatenação dos símbolos $Y_1Y_2Y_3...Y_n$. Note-se que a parte esquerda tem de conter, obrigatoriamente, um símbolo não terminal, enquanto que a parte direita é composta por uma sequência de terminais e não-terminais;
- **S** - uma gramática contém, obrigatoriamente, um símbolo não-terminal (**símbolo inicial**) a partir do qual todas as frases são derivadas.

Uma **derivação** é uma sequência de aplicações de regras das gramáticas que produzem um cadeia de símbolos terminais. O processo de substituição pode implicar zero ou mais passos, sendo que, após todas as substituição é obtida uma frase contendo apenas símbolos terminais. Assim, diz-se que $u \Rightarrow v$, se $u = x\alpha y \wedge v = x\beta y$ sendo que $x, y \in (V \cup \sum)^* \wedge \alpha \rightarrow \beta \in P$.

O número de passos utilizados na derivação é notado da seguinte forma:

$u \Rightarrow v$ - diz-se que u deriva em v num passo;
$u \Rightarrow^* v$ - diz-se que u deriva em v em zero ou mais passos;
$u \Rightarrow^+ v$ - diz-se que u deriva em v em um ou mais passos;

A linguagem gerada por uma gramática L(G) é dada pelo conjunto de todas as derivações que, partindo do estado inicial S, originam uma sequência de símbolos do alfabeto:

$$L(G) = \{u \in \sum^* : S \Rightarrow^* u\}$$

Hierarquia de Chomsky

Noam Chomsky, um linguista americano, propôs em 1959 uma classificação para as gramáticas formais, conhecida por Hierarquia de Chomsky[1], dividindo-as em quatro tipos de acordo com as restrições impostas às produções. Estas vão desde as gramáticas do tipo 0, as mais abrangentes, até às do tipo 3, as mais restritivas (ver diagrama da figura 2.2).

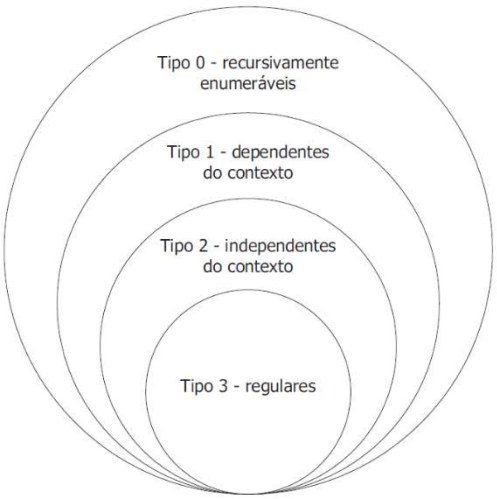


Figura 2.2: Hierarquia de Chomsky

- **tipo 0** - gramáticas livres ou sem restrições, são as mais abrangentes possíveis. Podem gerar linguagens complexas, difíceis de reconhecer. As produções são da forma $\alpha \rightarrow \beta$ na qual tanto α como β são sequências arbitrárias de símbolos terminais e não terminais. O lado esquerdo da produção não pode ser vazio;

[1] Noam Chomsky, On Certain Formal Properties of Grammars, Information and Control, Vol 2 (1959), 137-167

- **tipo 1** - gramáticas dependentes do contexto. As produções são da forma $\alpha A\beta \rightarrow \alpha\gamma\beta$ na qual α, β e γ são sequências arbitrárias de símbolos terminais e não terminais, sendo que γ não é nulo e A é um não terminal singular. Por outras palavras, A pode ser substituído por γ sempre que seja precedido por α e sucedido por β, isto é, num determinado contexto. Estas gramáticas têm que respeitar a condição $|\alpha A\beta| \leq |\alpha\gamma\beta|$, existindo uma única exceção a esta regra, a produção inicial pode ser do tipo $S \rightarrow \varepsilon$ para permitir a palavra vazia;

- **tipo 2** - gramáticas independentes do contexto. As produções são da forma $A \rightarrow \alpha$ na qual α é uma sequência arbitrária de símbolos terminais e não terminais, sendo A um símbolo não terminal singular. Tal significa que qualquer ocorrência de A pode ser substituída por α independentemente do contexto;

- **tipo 3** - gramáticas regulares. As produções são da forma $A \rightarrow a$, $A \rightarrow aB$ ou $A \rightarrow \varepsilon$ na qual A e B são não terminais singulares e *a* um terminal. Estas são as formas de gramáticas mais restritas em termos de poder de representação.

  De notar que as gramáticas regulares além das restrições no lado esquerdo das produções (devido a serem gramáticas independentes do contexto) têm também restrições do lado direito das produções. Estas restrições diminuem o poder expressivo da gramática.

As gramáticas do **tipo 3** dizem-se **lineares à direita** se as produções são da forma $A \rightarrow a$, $A \rightarrow aB$ ou $A \rightarrow \varepsilon$. Enquanto que nas **lineares à esquerda** as produções são da forma $A \rightarrow a$, $A \rightarrow Ba$ ou $A \rightarrow \varepsilon$. Note-se que apenas as gramáticas do **tipo 3** são lineares. Por outro lado toda a gramática linear, à esquerda ou direita, é regular.

Alguns autores consideram as gramáticas do **tipo 3** (ou regulares) estendidas, nas quais podemos substituir as regras do tipo $A \rightarrow aB$ por regras do tipo $A \rightarrow wB$ em que $w \in \sum^{+}$, no entanto, segundo a hierarquia de Chomsky[2], para este tipo de gramáticas, quando existe mais de um terminal numa regra, são enquadradas nas gramáticas de **tipo 2**.

Notação BNF e EBNF (Extended BNF)

A notação BNF (Backus Naur Form ou Backus Normal Form) foi originalmente criada por John Backus e Peter Naur, no final dos anos 50, para descrever a linguagem ALGOL. Desde então a sua utilização generalizou-se para a especificação de linguagens de programação.

Os meta-símbolos utilizados na notação BNF são:

::=	-	representa “definido como”;
|	-	indica uma alternativa;
< >	-	indica uma alternativa.

Considere-se o seguinte exemplo no qual é especificada a gramática relativa à instrução *if-then-else* e à definição de um identificador, vulgarmente utilizados em linguagens de programação, em notação BNF.

[2] Noam Chomsky, Aspects of the Theory of Syntax, Cambridge: M.I.T. Press, 1965.

```
<if> ::= if<condição> then <instruções> endif
        |if <condição> then <instruções> else <instruções> endif
<identificador> ::= <letra><alfanums>| _<alfanums>
<alfanums> ::= <alfanum><alfanums> | ε
<alfanum> ::= <letra> | <algarismo> |_
<letra> ::= a | A | b | … | z | Z
<algarismo> ::= 0 | 1 | 2 | … | 8 | 9
```

A notação EBNF estende a notação BNF com os seguintes meta-símbolos:

[ ] - indica uma parte opcional;

{ } - indica uma parte que se pode repetir 0 ou mais vezes;

( ) - indica precedências dentro da regra;

" " - indica um carácter a tratar como terminal e.g., "<".

Considere-se a seguinte especificação da gramática relativa à instrução *if-then-else* e à definição de um identificador, com recurso à notação EBNF.

```
<if> ::= if <condição> then <instruções> [ else <instruções>] endif
<identificador> ::= (<letra> | _ ) {(<letra> | <algarismo> | _ ) }
<letra> ::= a | A | b | … | z | Z
<algarismo> ::= 0 | 1 | 2 | … | 8 | 9
```

Mais recentemente, na literatura tornou-se comum a indicação dos símbolos não terminais a negrito sem os símbolos "<" e ">" e os terminais em texto normal.

1.10 Árvores de Derivação e Ambiguidade

Considere-se uma gramática independente de contexto G = (V, ∑, P, S) e uma derivação em G, designada S $\Rightarrow^*$ u. Esta derivação pode ser apresentada graficamente através de uma árvore de derivação da seguinte forma:

- o nó inicial é o símbolo inicial S;
- em cada passo da derivação, para cada variável A que seja uma folha da árvore, faz-se corresponder a produção A $\rightarrow \alpha_1\alpha_2\ldots\alpha_n$ a uma sub-árvore, em que A é o pai e $\alpha_1\alpha_2\ldots\alpha_n$ são os filhos;

Após este processo, a árvore de derivação terá a seguinte configuração:

- o nó principal é o símbolo inicial S;
- os símbolos não terminais são nós interiores;
- os símbolos terminais são nós folhas.

Uma gramática diz-se **ambígua**, se é possível obter uma mesma frase a partir de duas árvores de derivação distintas. Relembre-se que uma frase é uma sequência de terminais. Considere-se a seguinte gramática:

$$S \rightarrow aS \mid Sa \mid b$$

A frase *aba* pode ser obtida através de qualquer uma das seguintes sequências de derivação, cujas árvores respetivas são apresentadas na figura 2.3.

$$S \Rightarrow aS \Rightarrow aSa \Rightarrow aba$$

$$S \Rightarrow Sa \Rightarrow aSa \Rightarrow aba$$

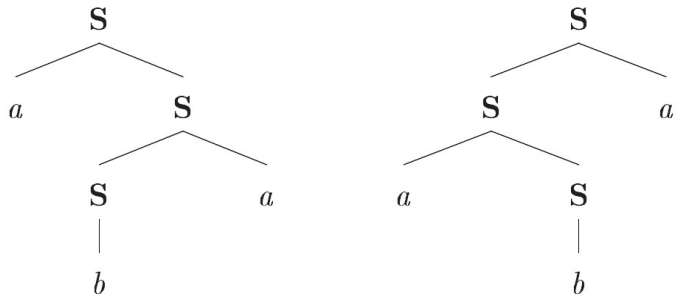


Figura 2.3: Árvores sintáticas para linguagem ambígua

Quando a ambiguidade depende da gramática pode ser eliminada, nos casos em que depende da linguagem, a ambiguidade não pode ser eliminada. Este último tipo de linguagens designa-se por linguagens inerentemente ambíguas sendo apresentado de seguida um exemplo.

$$L = \{a^i b^i c^j : i,j \geq 1\} \cup \{a^i b^j c^j : i,j \geq 1\}$$

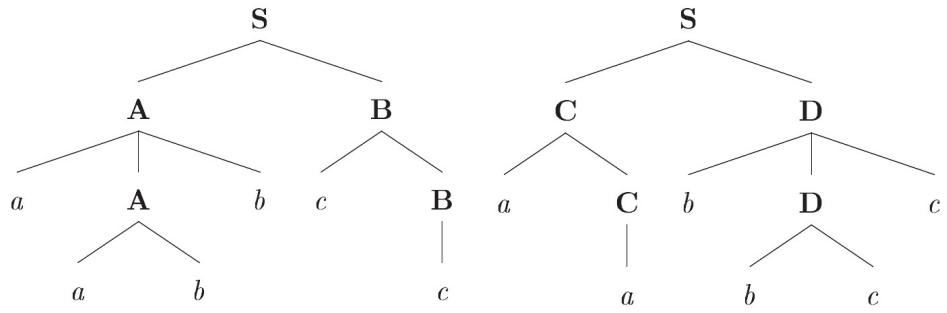
Esta linguagem pode ser reconhecida pela seguinte gramática:

$$\begin{aligned} S &\rightarrow AB \mid CD \\ A &\rightarrow aAb \mid ab \\ B &\rightarrow cB \mid c \\ C &\rightarrow aC \mid a \\ D &\rightarrow bDc \mid bc \end{aligned}$$

Esta linguagem é ambígua para palavras em que $i = j$, pois podem ser obtidas através de qualquer uma das alternativas da produção S. Isto pode ser verificado nas sequências de derivação, usando a derivação mais à esquerda, apresentadas de seguida e nas árvores de derivação apresentadas na figura 2.4.

$$S \Rightarrow AB \Rightarrow aAbB \Rightarrow aabbB \Rightarrow aabbcB \Rightarrow aabbcc$$

$$S \Rightarrow CD \Rightarrow aCD \Rightarrow aaD \Rightarrow aabDc \Rightarrow aabbcc$$



a) Usando a produção S → A B b) Usando a produção S → CD

Figura 2.4: Árvores sintáticas para linguagem inerentemente ambígua

1.11 Eliminação da ambiguidade

Conforme referido anteriormente a ambiguidade só pode ser eliminada quando não é inerente à linguagem. Assim, considere-se a seguinte gramática:

E → <E><OP><E> | <ID>
OP → + | - | * | /
ID → x | y | z

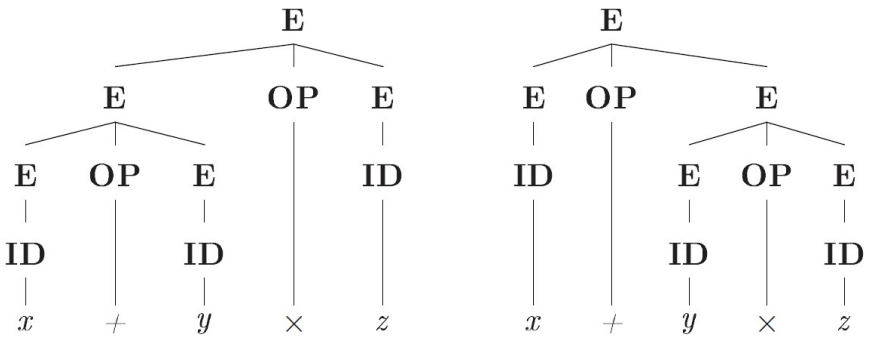
Se considerarmos a frase "$x + y * z$", facilmente se demonstra que esta gramática é ambígua, conforme ilustrado na figura 2.5 existem pelo menos duas árvores de derivação distintas para a referida frase. De seguida apresentam-se duas sequências de derivação para esta frase, usando a derivação mais à esquerda.

<E> ⇒ <E> <OP> <E> ⇒ <E> <OP> <E> <OP> <E> ⇒
⇒ <ID> <OP> <E> <OP> <E> ⇒ x <OP> <E> <OP> <E> ⇒
⇒ x + <E> <OP> <E> ⇒ x + <ID> <OP> <E> ⇒ $x + y$ <OP> <E> ⇒
⇒ $x + y$ * <E> ⇒ $x + y$ * <ID> ⇒ $x + y * z$

<E> ⇒ <E> <OP> <E> ⇒ <ID> <OP> <E> ⇒ x <OP> <E> ⇒ x + <E> ⇒
⇒ x + <E> <OP> <E> ⇒ x + <ID> <OP> <E> ⇒ $x + y$ <OP> <E> ⇒
⇒ $x + y$ * <E> ⇒ $x + y$ * <ID> ⇒ $x + y * z$

No caso de produções duplamente recursivas, a ambiguidade pode ser eliminada retirando uma das recursividades através duma nova produção. Assim sendo, de seguida apresenta-se a gramática anterior, agora reformulada.

E → <E> <OP> <F> | <F>
F → <ID>
OP → + | - | * | /
ID → x | y | z



a) Associatividade à esquerda b) Associatividade à direita

Figura 2.5: Árvores de derivação para expressão ambígua

Conforme se pode verificar, esta gramática só permite uma árvore de derivação para cada expressão. De seguida é apresentada uma sequência de derivação usando a derivação mais à esquerda.

<E> → <E> <OP> <F> ⇒ <E> <OP> <F> <OP> <F>
→ <F> <OP> <F> <OP> <F> ⇒ <ID> <OP> <F> <OP> <F>
→ x <OP> <F> <OP> <F> ⇒ x + <F> <OP> <F> ⇒ x + <ID> <OP> <F>
→ $x + y$ <OP> <F> ⇒ $x + y$ * <F> ⇒ $x + y * z$

Na figura 2.6, é apresentada a árvore de derivação para a frase $x + y * z$, usando a nova gramática.

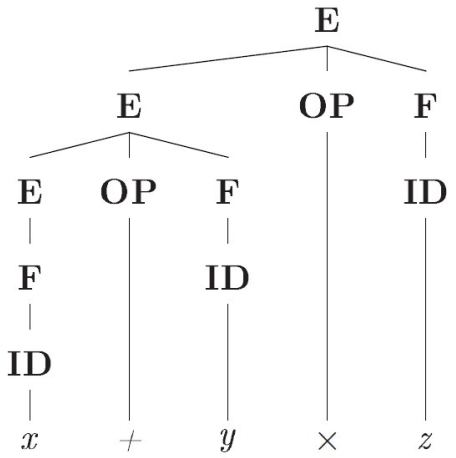


Figura 2.6: Árvore sintática para expressão sem ambiguidade

Note-se que, esta gramática ainda não considera a precedência dos operadores matemáticos. Para permitir à gramática analisar as expressões matemáticas com a precedência adequada, temos que dividir em duas produções distintas as operações aditivas (adição e subtração) e multiplicativas (multiplicação e divisão). Para conceder maior precedência às operações multiplicativas vamos tornar a expressão num conjunto de operações aditivas, em que os seus termos serão as operações multiplicativas. Obtemos assim 2 níveis, o primeiro representado pela produção **E** que define as operações aditivas e o segundo representado pela produção **T** que define as operações multiplicativas. No caso de termos outros operadores com maior precedência, seria criado um terceiro nível para esses operadores.

E → <E> <OP1> <T> | <T>
T → <T> <OP2> <F> | <F>
F → <ID> | (<E>)
OP1 → + | -
OP2 → * | /
ID → x | y | z

Para analisar a frase com esta gramática, obtemos a seguinte sequência de derivações:

<E> ⇒ <E> <OP1> <T> ⇒ <T> <OP1> <T> ⇒ <F> <OP1> <T>
⇒ <ID> <OP1> <T> ⇒ x<OP1> <T> ⇒ x + <T>
⇒ x + <T> <OP2> <F> ⇒ x+ <F> <OP2> <F> ⇒ x + <ID> <OP2> <F>
⇒ $x + y$ <OP2> <F> ⇒ $x + y$ * <F> ⇒ $x + y$ * <ID> ⇒ $x + y * z$

Esta sequência de derivação pode ser representada pela árvore de derivação da figura 2.7.

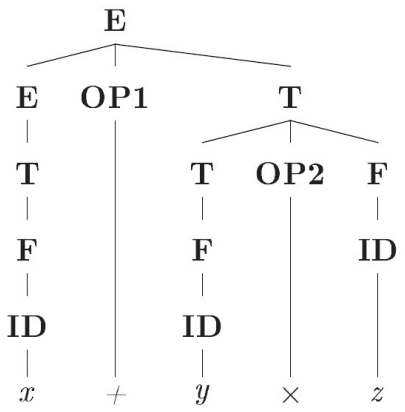


Figura 2.7: Árvore sintática para expressão com precedência

Um exemplo clássico de ambiguidade em programação, surge com a instrução *if*. No caso de dois *ifs* encadeados só com um *else* no final, não se sabe a qual *if* pertence o *else*. De seguida é apresentado um exemplo dessa gramática:

S → *if* <C> *then* <S> |
if <C> *then* <S> *else* | <S> |
x
C → 0 | 1

Conforme se pode verificar, esta gramática permite duas árvores de derivação para a frase "*if 0 then if 1 then x else x*". De seguida são apresentadas duas sequências de derivação usando a derivação mais à esquerda.

<S> ⇒ *if* <C> *then* <S> ⇒ *if 0 then* | <S>
⇒ *if 0 then if* <C> *then* <S> *else* <S> ⇒ *if 0 then if 1 then* <S> *else* <S>
⇒ *if 0 then if 1 then x else* <S> ⇒ *if 0 then if 1 then x else x*

<S> ⇒ *if* <C> *then* <S> *else* <S> ⇒ if 0 *then* <S> *else* <S>
⇒ *if 0 then if* <C> *then* <S> *else* <S> ⇒ *if 0 then if 1 then* <S> *else* <S>
⇒ *if 0 then if 1 then x else* <S> ⇒ *if 0 then if 1 then x else x*

As árvores de derivação correspondentes às sequências de derivação anteriores são apresentadas na figura 2.8.

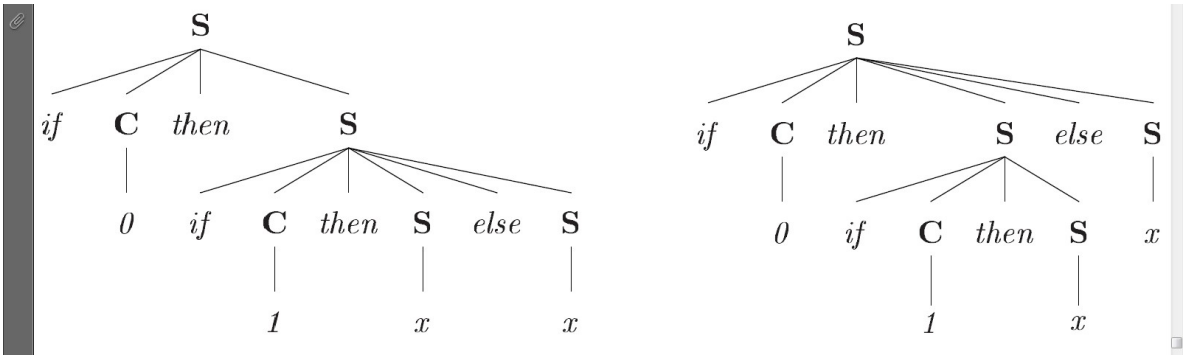


Figura 2.8: Árvores de derivação para instruções *if* ambíguas

Para resolver esta ambiguidade, será atribuído o *else* ao último *if*. Assim sendo, de seguida apresenta-se uma nova gramática para eliminar a ambiguidade. Nesta gramática, os *ifs* com *else*, nunca podem ter *ifs* sem *else* na parte do *then*, evitando-se assim a ambiguidade das frases.

S → <A> | <B>
A → *if* <C> *then* <A> *else* <A> | *x*
B → *if* <C> *then* <S> |
if <C> *then* <A> *else* <B>
C → *0* | *1*

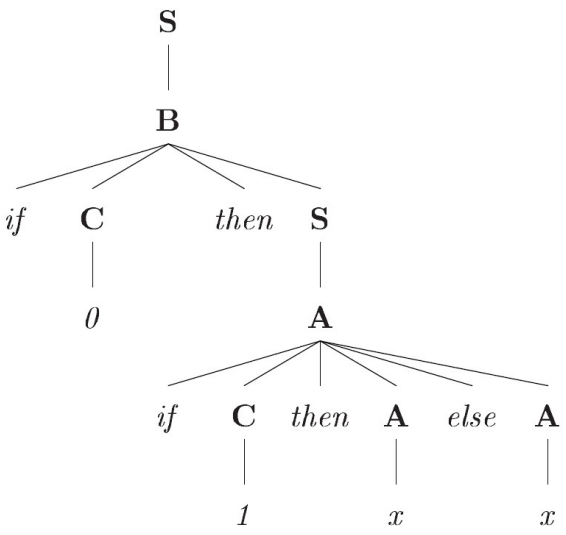


Figura 2.9: Árvore de derivação para instruções if sem ambiguidade

Usando esta gramática já é possível definir sem ambiguidade a frase referida, tal como apresentado na figura 2.9. Na prática, para resolver os problemas de ambiguidade é necessário identificar as produções que tem problemas de ambiguidade, reformulando-as para obter uma gramática sem ambiguidade.

1.12 Conversão de autómatos finitos em gramáticas

Para a conversão dum AF numa gramática é necessário criar uma produção (não terminal) para cada estado. Após o que, para cada transição é necessário criar uma alternativa na produção. Se a transição puder ter várias alternativas, estas podem ser desdobradas ou ser criada uma nova produção com essas alternativas. Todos os estados finais podem receber a cadeia vazia (ε). Considere-se o autómato finito não determinístico apresentado na figura 2.10.

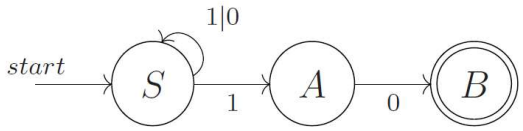


Figura 2.10: AFN a converter numa gramática regular

A gramática correspondente é a seguinte:

S → 0S | 1S | 1A
A → 0B
B → ε

Considere-se ainda o autómato finito determinístico capaz de reconhecer a mesma linguagem, apresentado na figura 2.11.

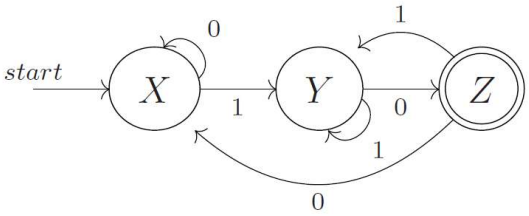


Figura 2.11: AFD a converter numa gramática regular

A gramática correspondente a este autómato é a seguinte:

X → 0X | 1Y
Y → 0Z | 1Y
Z → 0X | 1Y |ε

1.13 Conversão de gramáticas em autómatos finitos

Somente as gramáticas do tipo 3 podem ser convertidas em AF. Para a conversão a gramática deve ser linear à direita. Considere uma gramática capaz de reconhecer números binários pares:

$$L(G) = \{u \in \sum^* : u \text{ é um número par}\}$$
$$G = (V, \sum, P, S) = (\{A, B\}, \{0,1\}, P, A)$$

em que as produções (P) são:

$$A \rightarrow 1A \mid 0B$$
$$B \rightarrow 0B \mid 1A \mid \varepsilon$$

A primeira parte da conversão consiste na criação de um estado para cada não terminal, neste exemplo são necessários 2 estados, A e B. Depois é necessário substituir as produções por transições da seguinte forma:

- As produções do tipo $X \rightarrow \alpha Y$ geram no autómato a transições do tipo $\delta(X; \alpha) = Y$.
- As produções do tipo $X \rightarrow \varepsilon$ implicam que o estado correspondente à produção seja um estado final;
- As produções do tipo $X \rightarrow \alpha$, implicam a criação dum novo estado final extra, e essas produções são substituídas por transições para esse estado.

Na figura 2.12 está representado o autómato resultante da conversão desta gramática, neste caso determinístico.

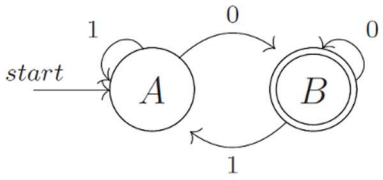


Figura 2.12: Gramática convertida numa AFD

1.14 Parsers em descida recursiva

A descida recursiva pode ser usada, para implementar *parsers* preditivos. Um *parser* preditivo é capaz de escolher a produção a aplicar simplesmente sabendo o não terminal atual e o terminal a ser processado. Este tipo de gramáticas são chamadas de LL(1), onde:

- o primeiro "L" indica que a frase é processada da esquerda para a direita;
- o segundo "L" indica que se usa primeiro a derivação mais à esquerda;
- o (1) indica o número de terminais de avanço necessários para escolher entre produções alternativas;

Todas as gramáticas do tipo 3 podem ser convertidas em LL(1). Considere a seguinte gramática capaz de reconhecer a linguagem com o alfabeto $\sum = \{a, b, c, d\}$ em que o primeiro "*b*" é precedido de um "*a*":

$$S \rightarrow aA \mid cS \mid dS$$
$$A \rightarrow aA \mid bB \mid cS \mid dS$$
$$B \rightarrow aB \mid bB \mid cB \mid dB \mid \varepsilon$$

Para construir um *parser* em descida recursiva é necessário criar uma função para processar cada um dos não terminais. De seguida é necessário para cada não terminal, processar os *starters* e chamar

as funções relativas aos não terminais que se lhes seguem. De seguida apresenta-se um *parser* em descida recursiva implementado em FLEX que reconhece a gramática apresentada.

```
%{
  enum{TOKEN_A,TOKEN_B,TOKEN_C,TOKEN_D,OUTRO,FIM};
  int token, nerros=0;
  void s(); /* protótipos */
  void a();
  void b();
%}
%%
a|A       return TOKEN_A;
b|B       return TOKEN_B;
c|C       return TOKEN_C;
d|D       return TOKEN_D;
.         return OUTRO;
\n        return FIM;
<<EOF>>   return FIM;
%%

void erro(char *s){
  nerros++;
  printf("%s<-Erro %d: %s\n",yytext,nerros,s);
  exit(1);}

void getToken(){
  printf("%s",yytext);
  token=yylex();}

void s() /* S-> aA | cS | dS */{
  switch (token) {
    case TOKEN_A : getToken(); a(); break;
    case TOKEN_C : getToken(); s(); break;
    case TOKEN_D : getToken(); s(); break;
    default   :
         erro("simbolo não reconhecido (esperava a,c ou d)");
  }}

void a() /* A-> aA | bB | cS | dS  */{
  switch (token) {
    case TOKEN_A : getToken(); a(); break;
    case TOKEN_B : getToken(); b(); break;
    case TOKEN_C : getToken(); s(); break;
    case TOKEN_D : getToken(); s(); break;
    default   :
         erro("simbolo não reconhecido (esperava a,b,c ou d)");
  }}

void b() /* B-> aB | bB | cB | dB | vazio */{
  switch (token) {
    case TOKEN_A : getToken(); b(); break;
    case TOKEN_B : getToken(); b(); break;
    case TOKEN_C : getToken(); b(); break;
    case TOKEN_D : getToken(); b(); break;
  }  /* como pode ser vazio não dá erro */}

int main(){
  token=yylex(); /* vai buscar o 1º token */
  s();           /* chama a produção inicial */

  if (nerros==0 && token==FIM)
    printf("\nExpressao válida \n");
  else
    erro("simbolo não reconhecido (esperava a,b,c,d ou fim)");
  return 0;}
```

Bibliografia:

[1]. Jeffrey D. Ullman, E. Hopcroft, Rajeev Motwani, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 2nd Edition, 2001.

[2]. Rui Gustavo Crespo, Processadores de Linguagens - da concepção à implementação, IST Press, 2001.
[3]. Alfred V. Aho, Monica S. Lam e Ravi Sethi ,Compiladores - Princípios , Técnicas e Ferramentas,Pearson, 2ª edição, 2007.
[4]. Noam Chomsky, On Certain Formal Properties of Grammars, Information and Control,Vol 2, 137-167, 1959.
[5]. Noam Chomsky, Aspects of the Theory of Syntax, Cambridge: M.I.T. Press, 1965.