UNIVERSIDADE NOVA DE LISBOA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
Departamento de Informática

LINGUAGENS FORMAIS E AUTÓMATOS

J. A. LEGATHEAUX MARTINS

LUIS MONTEIRO

1981

J. Legatheaux

LINGUAGENS FORMAIS E AUTÓMATOS

por

J. A. Legatheaux Martins
Luis Monteiro

UNL - 17/81 JULHO - 1981

Secção de
Ciência e Tecnologia da Programação
do Departamento de Informática da F.C.T. da U.N.L.

Lisboa Julho de 1981

ÍNDICE

NOTA PRÉVIA

O presente texto é o resultado da experiência acumulada por um dos autores (L.M.) no ensino das teorias das Linguagens Formais e dos Autómatos, complementada no último ano lectivo com a contribuição de J.A.L.M., na aplicação das referidas teorias à metodologia da programação pela sintaxe. Este ensino processou-se na disciplina "Linguagens Formais e Autómatos", no âmbito da Lic. em Engª Informática da F.C.T., da U.N.L.

Assim, este texto é constituído por diversos capítulos escritos em diversas épocas e integrados agora num volume único. Dado este carácter, ele tem de ser considerado uma obra em aberto visto que existem algumas incoerências de notação entre alguns capítulos assim como outros aspectos que serão numa futura reedição melhorados.

Os primeiros 4 capítulos são no essencial uma introdução à Teoria das Linguagens e Autómatos, os restantes introduzem diversas formas de aplicar esta teoria à programação assim como os complementos teóricos necessários à construção de analisadores sintácticos descendentes deterministas. Em anexo são apresentadas diversas propostas de projectos de alunos que foram por nós utilizados com sucesso.

Os exercícios propostos em cada capítulo estão eventualmente classificados. O significado das diferentes classificações é o seguinte:

(r) - exercício recomendado.

(a) - exercício avançado de enfase aplicação.

(t) - exercício avançado de enfase teórico.

Para além das presentes folhas a seguinte bibliografia complementar é recomendada:

- A. V. AHO, J. D. ULLMAN
  "The theory of parsing, Translation and Compiling"
  (2 Volumes)
  Prentice Hall, 1972.

- R. C. BACKHOUSE
  "Syntax of programming languages"
  Prentice Hall. International Series in Computer Science, 1979.

- P. CUNIN, M. GRIFFITHS, J. VOIRON
  "Apprendre La Compilation"
  Janeiro 1980. Springer Verlag.

- S. GINSBURG
  "The Mathematical Theory of Context free Languages"
  Mc Graw-hill Book Company, 1966

- J. E. HOPCROFT, D. ULLMAN
  "Formal Languages and Their relation to automata"
  A,dison Wesley, 1969.

- J. A. LEGATHEAUX MARTINS
  "A tradução como método de programação"
  Universidade Nova de Lisboa. Relatório interno, UNL - 9/81, 1981

- N. WIRTH
  "Algorithms + Data Structures = Programs"
  Prentice Hall, 1976. (Cap. 5)

Os autores agradecem a Manuel Jorge Esteves Matias que com dedicação e eficiência assegurou a edição deste texto.

Lisboa, Julho de 1981
J.A. Legatheaux Martins
Luis Monteiro

1. LINGUAGENS

1. Monoides livres

Por alfabeto entenderemos um conjunto finito qualquer, não vazio. Aos seus elementos chamaremos símbolos, caracteres ou letras.

EXEMPLOS. 1. O alfabeto do Pascal é constituído por todos os símbolos que podem ser utilizados na escrita dos programas desta linguagem, como por exemplo:

0 5 d z A M (+ / > begin if while goto

Note-se que begin, if, etc, são considerados como constituindo um único símbolo.

2. O conjunto de todas as palavras do Português pode ser encarado, em certos estudos, como um alfabeto.

Seja T um alfabeto. Justapondo, ou concatenando, zero ou mais símbolos de T obtemos cadeias ou palavras sobre T: cada palavra é uma expressão da forma:

$$a_1a_2 \ldots a_k \quad \text{em que } k \geq 0 \text{ e } a_1, a_2, \ldots, a_k \in T$$

A palavra obtida por concatenação de zero símbolos chama-se palavra vazia e será denotada λ. (Outras notações: Λ, e, 1.) O comprimento da palavra $x=a_1a_2 \ldots a_k$ define-se como sendo $|x|=k$. Tem-se $|\lambda|=0$. O conjunto de todas as palavras sobre T denota-se T^*. O conjunto de todas as palavras não vazias sobre T denota-se T^+. Assim, $T^+ = T^* - \{\lambda\}$ e $T^* = T^+ \cup \{\lambda\}$.

EXEMPLOS. 3. Se $T = \{a,b\}$ algumas palavras sobre T são a, b, aa, ab, ba, bb, aaa, aab, etc.

4. (a+Max)/5 e if (;3 são palavras sobre o alfabeto do Pascal de comprimentos 9 e 4 respectivamente.

5. A instrução vazia do Pascal é a palavra vazia.

OBSERVAÇÕES.

1. Nada obsta a que numa palavra um símbolo apareça repetido.

2. Para todo o alfabeto T, o conjunto T* é infinito. Basta notar que dado $a \in T$ todas as palavras a, aa, aaa, ..., em número infinito, estão incluídas em T*.

3. Todo o símbolo $a \in T$ pode ser entendido como uma palavra de comprimento 1 sobre T. Deste modo, $T \subset T^*$, e mesmo $T \subset T^+$.

4. Se $S \subset T$ então $S^+ \subset T^+$ e $S^* \subset T^*$.

Sejam $x=a_1 \ldots a_k$ e $y=b_1 \ldots b_n$ palavras sobre T. Põe-se $x=y$ se e só se $k=n$ e cada $a_i=b_i$.

A concatenação ou justaposição das palavras x e y é a palavra x.y que se obtem escrevendo:

$$x.y = a_1 \ldots a_k\, b_1 \ldots b_n.$$

Geralmente escreve-se xy em vez de x.y. É óbvio que se tem $|xy|=|x|+|y|$.

A concatenação satisfaz as seguintes propriedades:

i) $\lambda x = x\lambda = x$,

ii) $x(yz)=(xy)z$,

iii) $xy \neq yx$, em geral (por exemplo, se x=ab e y=ba).

As duas primeiras propriedades mostram que λ é o elemento neutro para a concatenação e que esta operação é associativa. Um conjunto onde esteja definida uma operação associativa para a qual exista um elemento neutro chama-se um monóide. Por conseguinte, T* é um monóide. Mas é mais do que isso, T* chama-se monóide livre de base T, ou gerado por T, porque satisfaz a seguinte propriedade suplementar:

iv) Todo o elemento de T* se escreve de uma única forma como uma concatenação de elementos de T.

Uma palavra x é um factor ou uma subpalavra de uma palavra y se existirem palavras v e w tais que y=vxw. Também se diz que x ocorre em y. Se $v=\lambda$ então y=xw e x é um factor esquerdo, ou um prefixo, de y; se $w=\lambda$ então y=vx e x é um factor direito, ou um sufixo, de y. Se além disso, $|x|<|y|$, diz-se próprio: factor próprio, prefixo próprio, etc.

EXEMPLO. 6. As subpalavras de abab são λ , a, b, ab, ba, aba, bab, abab (a única que não é própria). Note-se que ab é simultâneamente um sufixo e um prefixo.

Dado $x \in T^*$, poremos $x^n = \underbrace{xx \ldots x}_{n \text{ vezes}}$ se $n>0$ e $x^0 = \lambda$.

Por exemplo, se $a,b \in T$, $a^2=aa$ e $(ab)^3=ababab$. Tem-se $\lambda^n = \lambda$ para todo o $n \geq 0$.

2. Linguagens

Uma linguagem sobre um alfabeto T é um conjunto de palavras sobre T, isto é, é um subconjunto de T^*.

EXEMPLOS. 7. O FORTRN, o ALGOL e mesmo o Português estão incluídos nesta definição.

8. O conjunto vazio $\emptyset$ e $\{\lambda\}$ são duas linguagens (distintas!) sobre qualquer alfabeto. T, T^+ e T^* são linguagens sobre T.

9. Sejam T um alfabeto e $a,b \in T$. Então

$$L_1 = \{a^n : n \geq 0\}$$
$$L_2 = \{x \in T^* : |x| \leq 4\}$$
$$L_3 = \{a^n b^n : n \geq 0\}$$
$$L_4 = \{a^n b^m : n \geq 0, m \geq 0\}$$

são linguagens sobre T.

Uma linguagem diz-se finita se, como subconjunto de T^*, for um conjunto finito, isto é, se contiver apenas um número finito de palavras; de contrário diz-se infinita.

EXEMPLO. 10. As linguagens $\emptyset$, $\{\lambda\}$, T e L_2 dos exemplos 8 e 9 são finitas, ao passo que T^+, T^*, L_1, L_3 e L_4 são infinitas.

O problema da caracterização das linguagens finitas é, em princípio, simples: basta enumerar as palavras que as compõem (estamos a pôr o problema de um ponto de vista essencialmente teórico, escamoteando o facto de saber se o método propos

to tem alguma viabilidade prática). O mesmo já não se pode dizer das linguagens infinitas. Em certos casos simples, como nos do exemplo 9, podem definir-se as linguagens por meio de propriedades que as palavras que as compõem, e só elas, satisfazem. Mas em geral isso não se pode fazer. Com efeito, demonstra-se em Teoria da Computabilidade que a "maior parte" das linguagens não é passível de uma caracterização finita. (O raciocínio é essencialmente o seguinte: o conjunto de todas as linguagens sobre um determinado alfabeto T não é numerável, isto é, não pode ser posto em correspondência biunívoca com o conjunto dos inteiros positivos; por outro lado, qualquer caracterização finita de uma linguagem sobre T pode sempre ser codificada sob a forma de uma palavra de uma outra linguagem; como o conjunto de todas as palavras de uma linguagem é finito ou numerável, segue-se que nem todas as linguagens sobre T podem ser finitamente caracterizadas).

Nota.(1) Os meios mais úteis de caracterização de linguagens são as gramáticas e os autómatos, cujo estudo se iniciará no capítulo seguinte. Para já, vejamos algumas operações que se podem efectuar sobre linguagens.

3. Operações sobre linguagens

Além das operações de reunião, intersecção e complementação, herdadas da Teoria dos Conjuntos, definiremos agora outras operações sobre linguagens que se prendem directamente com a operação de concatenação de palavras.

O produto, ou concatenação, de duas linguagens L e M é a linguagem:

$$L.M = \{xy : x \in L,\ y \in M\}$$

constituída por todas as palavras que se obtêm concatenando uma palavra de L e outra de M. Também se escreve LM em vez de L.M.

OBSERVAÇÃO. 5. Os alfabetos sobre os quais estão definidas L e M podem ser diferentes. Se forem T e S respectivamente, então LM é uma linguagem sobre $T \cup S$.

(1) falar também das expressões regulares.

EXEMPLO. 11. $\{a^n b^m : n \geq 0, m \geq 0\} = \{a\}^* . \{b\}^*$.

12. Para toda a linguagem L, $L.\{\lambda\} = \{\lambda\}. L = L$.

13. Sejam $L = \{x \in T^* : |x| = n\}$ e $L' = \{x \in T^* : |x| = n+1\}$.

Então $L' = T.L = L.T$. (Prove isto!)

Dada uma linguagem L, podemos agora definir a operação de potenciação de L a partir do produto. Assim,

$$L^0 = \{\lambda\}, \qquad L^n = \underbrace{L. L \ldots L}_{n \text{ vezes}} \qquad (n > 0).$$

O fecho de L é:

$$L^* = \bigcup_{n \geq 0} L^n$$

e o fecho positivo de L é:

$$L^+ = \bigcup_{n \geq 1} L^n.$$

Cada palavra de L^* (respectivamente L^+) é então a concatenação de 0 ou mais (resp. 1 ou mais) palavras de L.

PROPOSIÇÃO.

a) $L^* = L^+ \cup \{\lambda\}$.

b) $L^+ = LL^* = L^*L$.

Demonstração: Mostraremos apenas que $L^+ = LL^*$. Se $x \in L^+$, então $x \in L^n$ para um certo $n \geq 1$, podendo escrever-se $x = x_1 x_2 \ldots x_n$, com cada $x_i \in L$. Tem-se que x é a concatenação de $x_1 \in L$ com $x_2 \ldots x_n$, a qual pertence a L^* visto que é a concatenação de $n-1 \geq 0$ palavras de L. Logo $x \in LL^*$. Inversamente, se $x \in LL^*$ pode escrever-se $x = yv$, com $y \in L$ e $v \in L^*$. Como v é uma concatenação de 0 ou mais palavras de L, $x = yv$ é uma concatenação de 1 ou mais palavras de L, logo $x \in L^+$. □

Veremos mais adiante como se pode definir, a partir destas operações sobre linguagens, uma classe de linguagens às quais se dá o nome de linguagens "regulares".

4. Exercícios

1. Dada a palavra abab, desenhe o grafo das suas subpalavras, que se define do seguinte modo:

   - os vértices são as subpalavras de abab;
   - existe um arco de x para y se x for uma subpalavra de y (não desenhe lacetes nem arcos que se obtenham por transitividade).

   Sublinhe todos os prefixos de abab e sobrelinhe todos os seus sufixos.

2.(r) Mais geralmente, ao construir-se o grafo das subpalavras de uma dada palavra x constata-se que:

   - com a excepção de um lacete em torno de cada vértice, o grafo não tem circuitos;
   - a função ordinal é tal que a ordem de cada vértice é o comprimento da palavra que o representa; em particular, a ordem do grafo é $|x|$;
   - o conjunto dos prefixos de x define um caminho do grafo, o qual não está contido em mais nenhum outro caminho; o mesmo se diz dos sufixos de x. (Ler a palavra x da esquerda para a direita, símbolo a símbolo, corresponde a percorrer o caminho dos prefixos; o prefixo em que nos encontramos em cada passo corresponde à parte já lida de x.)

   Verifique estas afirmações.

3. Sejam w, x, y, z palavras tais que wx=yz e $|w| \leq |y|$. Mostre que existe uma palavra v tal que y=wv e x=vz. (Sugestão: considere a figura

4.(r) Seja T um alfabeto. Mostre que T^n é o conjunto das palavras sobre T de comprimento n. Conclua que T*, quer considerado como o conjunto de todas as palavras sobre T quer como o fecho de T, denota o mesmo conjunto.

5.(r) Seja T={a, b, c}. Descreva as linguagens denotadas pelas seguintes expressões regulares:

- aa*
- $(a+b+c)^n$ (a+b+c)*
- a* b c*
- (a+b+c)* c (a+b+c)*
- (a+b+c)* c+c (a+b+c)*
- a (a+b+c)* b

(NOTA: $(a+b+c)^n$ não é uma expressão regular, mas sim uma abreviatura de:

(a+b+c)(a+b+c) ... (a+b+c)

n vezes.)

6.(r) Seja T={a, b, c}. Escreva expressões regulares que denotem as seguintes linguagens:

- todas as palavras com um número par de b's;
- todas as palavras com excepção de λ;
- todas as palavras com pelo menos uma ocorrência de a ou de b;
- todas as palavras em que uma ocorrência de a é imediatamente seguida de uma ocorrência de b.

7. Considere o alfabeto composto pelos simbolos

begin l s : ; end

em que 'l' e 's' são interpretados respectivamente como

uma etiqueta ("label") e uma instrução ("statement").

Construa uma expressão regular que denote todos os "programas" da forma:

begin s ; l : s ; l : ; ; l : s end

NOTAS:

1) - Há só um begin e só um end em cada programa.

2) - As instruções elementares podem ou não ser etiquetadas.

3) - Podem ocorrer instruções vazias no corpo do programa, etiquetadas ou não.

8.(r) Demonstre as seguintes igualdades entre expressões regulares:

$P + Q = Q + P$

$P + \emptyset = P$

$(P + Q) + R = P + (Q + R)$

$P + P = P$

$P.\lambda = \lambda.P = P$

$(P.Q).R = P.(Q.R)$ de onde se escreve geralmente: PQR

$P.\emptyset = \emptyset.P = \emptyset$

$P.(Q + R) = P.Q + P.R$

$(P + Q).R = P.R + Q.R$

$P^* = \lambda + PP^*$

$(P^*)^* = P^*$

9.(r) A semântica de uma estrutura de controlo pode definir-se descrevendo o conjunto de todas as execuções possíveis que essa estrutura origina. Por exemplo, dada a instrução:

while b do s

o seu comportamento descreve-se com base no alfabeto $b, \bar{b}, s$, em que b (resp. $\bar{b}$) significa que a condição b foi executada e forneceu "verdade" (resp. "falso") como resultado, e s denota a execução da instrução s. A seguinte linguagem representa o conjunto de todas as execuções possíveis da referida instrução:

$$\bar{b},\ b\ s\ \bar{b},\ b\ s\ b\ s\ \bar{b},\ \ldots\ .$$

Esta mesma linguagem é representada pela seguinte expressão regular:

$$(bs)^*\ \bar{b}$$

a) - Descreva o conjunto das execuções possíveis da instrução

repeat s until b

por meio de uma expressão regular.

b) - Mostre que

repeat s until b

e

s;
while not b do s

são sintacticamente equivalentes.

(ESCLARECIMENTOS: (1) Se p e q forem sequências de instruções cujos comportamentos são descritos por expressões regulares P e Q respectivamente, então: (a) o compor

tamento de p;q é descrito por PQ;
(b) por outro lado, p e q são sintacticamente equivalentes se e só se P = Q.

(2) Suponha que $\bar{\bar{b}} = b$.)

c) - O comportamento de

if b **then** s_1 **else** s_2

pode ser descrito pela expressão regular

$$bs_1 + \bar{b}s_2$$

A **if** b **then** s corresponde $bs + \bar{b}$. Mostre que

while b **do** s e **if** b **then**
begin s;
while b **do** s
end

são sintacticamente equivalentes.

d) - Aproveite a alínea anterior para definir a instrução **while** em termos da instrução **repeat**.

10.(r) Sejam X, A e B expressões regulares tais que

$$X = AX + B \quad (1)$$

Mostre que X = A*B satisfaz a equação (1)

(Sugestão: utilizar as igualdades referidas em 8).

2. GRAMÁTICAS

1. Definição

As gramáticas constituem provávelmente a forma mais importante de caracterização de linguagens. Essa importância advem-lhes (pelo menos parcialmente) do facto de que, além de definirem as linguagens, conferem às palavras uma estrutura útil. As gramáticas são conhecidas como "dispositivos geradores" (enquanto que os autómatos são "dispositivos reconhecedores"), visto que o processo que utilizam para definir linguagens é o da geração das suas palavras.

A definição de uma gramática envolve a especificação de símbolos não-terminais ou símbolos auxiliares ou variáveis; de símbolos terminais; de um símbolo inicial, que é um dos símbolos não-terminais, ao qual também se chama axioma; e de um conjunto finito de produções, regras de reescrita ou simplesmente regras, a serem definidas mais abaixo. Podemos então dizer que uma gramática G é um quádruplo:

$$G = (N, T, S, P)$$

em que:

N é um conjunto finito de símbolos não terminais, ou alfabeto não terminal;

T é um conjunto finito de símbolos terminais, ou alfabeto terminal;

$S \in N$ é o símbolo inicial;

P é um conjunto finito de produções; uma produção é um par ordenado (α, β) de palavras sobre o alfabeto $N \cup T$ tais que α contém pelo menos um símbolo não-terminal, isto é, $\alpha \in (N \cup T)^* N (N \cup T)^*$ e $\beta \in (N \cup T)^*$.

A produção (α, β) escreve-se mais correntemente $\alpha \rightarrow \beta$ ou, por vezes, $\alpha ::= \beta$. Se todas as produções de P cujo membro esquerdo é α forem $\alpha \rightarrow \beta_1$, $\alpha \rightarrow \beta_2$, ..., $\alpha \rightarrow \beta_n$ escreveremos $\alpha \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n$, para abreviar.

EXEMPLO. 1 A "gramática dos identificadores" em Pascal tem como símbolos não-terminais <identifier>, <letter>,

<digit>; como símbolos terminais a, b, ..., z, A, B, ..., Z, 0, 1, ..., 9; como símbolo inicial <identifier>` e como produções:

<identifier>::= <letter>|<identifier><letter>|
<identifier><digit>

<letter>::= a|b|...|z|A|B|...|Z

<digit>::= 0|1|...|9

Vejamos agora como uma gramática G gera uma linguagem. Em termos gerais eis como as coisas se passam. Cada produção $\alpha \to \beta$ é entendida como uma regra de substituição - daí o nome de regra de reescrita: em cada palavra que contenha α como subpalavra, pode substituir-se uma (e uma só!) ocorrência de α por β, obtendo-se uma nova palavra. Partindo de uma palavra x e aplicando-lhe sucessivamente 0 ou mais substituições utilizando as regras da gramática, obtem-se uma palavra y que se diz ter sido gerada por x. Pois bem, a linguagem gerada pela gramática é, por definição, o conjunto das palavras sobre o alfabeto terminal que são geradas pelo símbolo inicial.

Passemos às definições formais.

Seja G= (N, T, S, P) uma gramática. Vamos definir no conjunto $(N \cup T)^*$ uma relação binária $\overset{}{\underset{G}{\Rightarrow}}$ (que se lê deriva directamente em, e que exprime a ideia de substituição apresentada mais acima) por: se $\gamma\alpha\delta$ for uma palavra em $(N \cup T)^*$ e se $\alpha \to \beta$ for uma produção em P, então $\gamma\alpha\delta \underset{G}{\Rightarrow} \gamma\beta\delta$. Note-se que, para toda a produção $\alpha \to \beta$, se tem $\alpha \underset{G}{\Rightarrow} \beta$, o que decorre da definição fazendo $\gamma = \delta = \lambda$.

Podemos agora considerar as diversas potências, o fecho transitivo e o fecho reflexo-transitivo desta relação, cujas definições relembraremos aqui.

Dado $n \geq 0$ poremos $\alpha \overset{n}{\underset{G}{\Rightarrow}} \beta$ se existir uma sucessão de n+1 palavras $\alpha_0, \alpha_1, \ldots, \alpha_n$ tais que $\alpha_0 = \alpha$, $\alpha_n = \beta$ e $\alpha_{i-1} \underset{G}{\Rightarrow} \alpha_i$ para $1 \leq i \leq n$, o que se pode sintetizar escrevendo:

$$\alpha = \alpha_0 \underset{G}{\Rightarrow} \alpha_1 \underset{G}{\Rightarrow} \ldots \underset{G}{\Rightarrow} \alpha_n = \beta.$$

Diz-se que esta sucessão de palavras (ou, de forma mais sugestiva, a expressão anterior) é uma derivação (de comprimento n) de β a partir de α.

Põe-se $\alpha \overset{+}{\underset{G}{\Longrightarrow}} \beta$ se, para algum $n > 0$, $\alpha \overset{n}{\underset{G}{\Longrightarrow}} \beta$, e diz-se que β deriva não-trivialmente de α.

Finalmente, $\alpha \overset{*}{\underset{G}{\Longrightarrow}} \beta$ se, para algum $n \geq 0$, $\alpha \overset{n}{\underset{G}{\Longrightarrow}} \beta$, dizendo-se que β deriva de α.

OBSERVAÇÕES. 1. Tem-se $\alpha \overset{0}{\underset{G}{\Longrightarrow}} \beta$ se e só se $\alpha = \beta$.

2. Tem-se $\alpha \overset{*}{\underset{G}{\Longrightarrow}} \beta$ se e só se $\alpha \overset{+}{\underset{G}{\Longrightarrow}} \beta$ ou $\alpha = \beta$.

3. Quando α for o símbolo inicial S, omite-se a referência a α, falando-se em derivação (de comprimento n) de β, etc.

4. Quando não houver confusão possível com outras gramáticas, escreveremos simplesmente $\Longrightarrow$, $\overset{n}{\Longrightarrow}$, $\overset{+}{\Longrightarrow}$, $\overset{*}{\Longrightarrow}$, em vez dos mesmos símbolos com o G em índice.

Dado $x \in (N \cup T)^*$, denotaremos por $D(x)$ o conjunto de todas as palavras que derivam de x, isto é,

$$D(x) = \left\{ y : y \in (N \cup T)^* \text{ e } x \overset{*}{\Longrightarrow} y \right\}.$$

A linguagem gerada por x é o conjunto $L(x)$ de todas as palavras sobre o alfabeto terminal que derivam de x, isto é,

$$L(x) = \left\{ y : y \in T^* \text{ e } x \overset{*}{\Longrightarrow} y \right\}.$$

Tem-se $L(x) = D(x) \cap T^*$. Se $x \in T^*$ então $D(x) = \{x\} = L(x)$.

A linguagem gerada pela gramática G é

$$L(G) = L(S)$$

em que S é o símbolo inicial de G. Por outras palavras,

$$L(G) = \left\{ x : x \in T^* \text{ e } S \overset{*}{\Longrightarrow} x \right\}.$$

EXEMPLO. 2. Eis um exemplo de uma derivação na gramática dos identificadores:

<identifier> ⟹ <identifier><digit>

⟹ <identifier><letter><digit>

⟹ <letter><letter><digit>

⟹ d<letter><digit>

⟹ dM <digit>

⟹ dM4

Note-se que a palavra dM4 podia ter sido derivada de outra maneira, não essencialmente distinta desta, alterando a ordem de aplicação das produções. Voltaremos adiante a este ponto. Pondo T_l e T_d para os conjuntos das letras e dos digitos respectivamente, tem-se L(<letter>)=T_l, L(<digit>)=T_d e L(<identifier>)=$T_l (T_l \cup T_d)^*$.

2. Classificação das gramáticas

As gramáticas podem ser classificadas segundo as formas das produções que entram na sua definição, as quais por sua vez dependem das formas das palavras que as compõem. A classificação que apresentamos deve-se a Chomsky.

TIPO 0. As produções do tipo 0 não são sujeitas a nenhuma restrição, pelo que qualquer produção é do tipo 0.

Assim, qualquer gramática é do tipo 0, dizendo-se também que é não-restringida.

TIPO 1. - As produções do tipo 1 são da forma $\alpha A \beta \rightarrow \alpha \gamma \beta$, em que $\alpha, \beta \in (N \cup T)^*$, $A \in N$ e $\gamma \in (N \cup T)^+$ (isto é, $\gamma \neq \lambda$). Uma gramática que não tenha produções do tipo 0 diz-se do tipo 1, ou ainda dependente do contexto ("context-sensitive").

TIPO 2. - As produções do tipo 2 são da forma $A \rightarrow \alpha$, onde $A \in N$ e $\alpha \in (N \cup T)^*$. É óbvio que toda a produção do tipo 2 é também do tipo 0 (e, se $\alpha \neq \lambda$, também do tipo 1). As gramáticas do tipo 2 chamam-se gramáticas independentes do contexto ("context-free"), gramáticas algébricas ou gramáticas de Chomsky.

TIPO 3. - As produções deste tipo são da forma $A \rightarrow x$ ou $A \rightarrow xB$ com $A, B \in N$ e $x \in T^*$. É claro que as produções do tipo 3 são também do tipo 2. As respectivas gramáticas chamam-se também lineares direitas. (As gramáticas "lineares esquerdas" definem-se de modo análogo, com produções da forma $A \rightarrow Bx$ em vez de $A \rightarrow xB$. O seu estudo é decalcado do das gramáticas lineares direitas, e não será aqui abordado.)

As classes de linguagens geradas por estes tipos de gramáticas diferem entre si. Isto é, para cada i, $0 \leq i \leq 2$, é possível encontrar uma linguagem gerada por uma gramática do tipo i que não o seja por nenhuma gramática do tipo i+1. Uma linguagem diz-se do tipo i ($0 \leq i \leq 3$) se existir uma gramática do tipo i

que a gere. Além disso, empregam-se também as designações de linguagem independente do contexto, etc. Às linguagens do tipo $\emptyset$ chamaremos também linguagens recursivas, e às do tipo 3 linguagens de Kleene (ou ainda linguagens regulares, como veremos mais adiante embora estas sejam introduzidas por intermédio de uma definição diferente, que terá de ser provada equivalente à presente definição).

As linguagens mais importantes do ponto de vista das suas aplicações às linguagens de programação actuais são as linguagens algébricas (tipo 2). Serão estas as únicas linguagens que serão estudadas neste curso. As linguagens de Kleene, apesar de constituirem um caso particular deste, serão em parte estudadas independentemente.

chamar a atenção para as ≠'s convenção para descrever gramáticas. BNF, outras, as abreviaturas em q se dá nó's produção, etc.

3. Exercícios

1.(t) Sejam $G=(N,T,S,P)$ uma gramática e $A \in N$. Em que condições é que $L(A)=\emptyset$? Conclua que, extraindo A de N, e todas as produções em que A ocorre de P, se obtem uma gramática que gera a mesma linguagem que G.

Tire a mesma conclusão se A for tal que, para todo o $\gamma \in (N\ T)^*$ tal que $S \Rightarrow \gamma$, A não ocorre em γ.

2. Classifique pelo tipo as seguintes produções: A->a, B->AB, B ->xA, aAbcD->abcDbcD, AC->A, A->A, Abc->abDbc, AP->AbBc em que as maiúsculas denotam não terminais e as minúsculas terminais.

3. Classifique as seguintes gramáticas (convenções idênticas a 2 quanto às letras excepto que λ é a palavra vazia.).

a) S ->aA
A ->c|ab
B ->abc

b) S ->ASB|d
A ->aA|b
aaA ->aaBC
B ->dcb

c) S ->aX
X ->aX|λ

4.(r) a) Dada a C-Gramática S ->aaS/b, mostre que:

$L(S)=\{a^{2n}\ b:\ n \geq 0\}$

b) Dada a C - Gramática S ->aSb/λ, determinar L(S).

c) Dada a C - Gramática G = (N,T,S,P) com $N=\{S\}$

$X=\{a_1, \ldots, a_n, \vee, \wedge, \neg, (,)\}$ e produções:

$S \rightarrow a_1|\ldots|a_n|\neg S|(S \wedge S)|(S \vee S)$

determinar L(G)

5.(r) Para cada uma das gramáticas lineares

G_1: S -> aS|b

G_2: S -> a|aT
T -> bT|b

G_3: S -> aS|bT|λ
T -> bT|λ

exprimir $L(G_1)$, $L(G_2)$ e $L(G_3)$ em termos de expressões regulares.

6.(r) Escrever gramáticas que gerem as linguagens de todas as execuções possíveis das instruções while e repeat.

7.(r) Dada a seguinte gramática que gera a linguagem das sequências válidas de operações sobre ficheiros em Pascal, determine a correspondente expressão regular.

<operações-sobre-ficheiros>::=<criar-ler><operações-sobre ficheiros>|λ

<criar-ler>::=<criar><leituras>
<criar>::=rewrite <escrever>
<escrever>::= put <escrever>|λ
<leituras>::= <leitura><leituras>|λ
<leitura>::=reset <ler>
<ler>::=get <ler>|λ

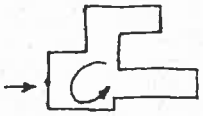
Não é uma gramática regular mas gera uma ling. regulas, chamar a atenção para em caso.

8.(a) (a) Considere a seguinte gramática sobre o alfabeto terminal {a,b}:

S ->a S b | b S a | λ

Todas as palavras geradas por esta gramática contêm igual número de a's e de b's. Que produções se devem acrescentar à gramática para que sejam geradas todas as palavras com igual nº de a's e de b's (e só elas) ?

(b) Considere o seguinte polígono



em que todos os lados são paralelos a duas direcções ortogonais dadas. Partindo de um ponto a meio de um lado, e percorrendo o polígono no sentido contrário ao dos ponteiros de um relógio e regressando ao ponto inicial, registam-se em cada vértice mudanças de sentido de $+90^{\circ}$ ou -90°. Designando essas mudanças de sentido por "a" e "b" respectivamente, obtem-se uma palavra sobre o alfabeto $\{a,b\}$. Por exemplo, com a figura anterior e para o ponto indicado, obtem-se a palavra

aabaabbaaaba

Defina uma gramática que gere todas as palavras assim obtidas para todo os possíveis polígonos do tipo indicado.

(Observação: a alínea anterior pode ser útil.)

9.(r) Uma árvore (estritamente) binária é um só nó, ou então um nó com duas sub-árvores binárias, a esquerda e a direita (o nó em questão é a "raiz" da árvore). Uma tal árvore pode ter informações associadas a cada nó, a qual estamos interessados em listar. A cada forma específica de proceder a essa listagem chamaremos "percurso" da árvore. Suponhamos que dispomos de três operações: "listar a informação contida na raiz", "aceder à sub-árvore esquerda da raiz" e "aceder à sub-árvore direita da raiz", as quais simbolizaremos por "r", "e" e "d" respectivamente. A sequência de operações que permite percorrer uma dada árvore dá origem a uma palavra sobre o alfabeto $\{r, e, d\}$. Uma gramática que gere todos os possíveis percursos prefixos de árvores binárias é a seguinte:

< percurso prefixo> :: = r | re <percurso prefixo> d
<percurso prefixo>

Defina gramáticas que geram os percursos infixos e sufixos de árvores binárias.

(Terminologia: Em Inglês, estes percursos chamam-se respectivamente "prefixed", "infixed", "postfixed".)

10. Dada a linguagem "assembler" de sua preferência, escreva a gramática correspondente a um seu sub-conjunto.

Selecione apenas 3 a 4 instruções de cada grupo e não considere pseudo-operações.

11.(r) Dada uma linguagem, sub-conjunto do Basic com as seguintes possibilidades:

1 - todas as variáveis são inteiras e os seus identificadores são constituídos por uma única letra de A a Z

2 - As constantes são só inteiros

3 - Todas as instruções têm "label"

4 - As instruções são:

variável : = expressão aritmética

goto label

if (expressão relacional) then instrução ≠ de if

input variável

output expressão aritmética

5 - Operadores unários

6 - Operadores binários

7 - Operadores relacionais

Escreva uma gramática deste mini-basic. Para a parte das expressões pode inspirar-se nas expressões do Pascal, simplificando-as.

Classifique a gramática obtida.

12.(a) Suponha que na linguagem Pascal se introduziram 2 novas instruções com a seguinte semântica:

Semântica da instrução 'loop':

O conjunto de instruções incluidas entre as palavras reservadas loop e endloop são executadas repetida e indefinidamente até ser executada a instrução exit (cuja forma de escrita é exactamente essa). Neste caso a execução continua após a palavra endloop.

Uma instrução de exit só pode figurar dentro de 1 loop e refere-se sempre ao mais próximo loop que a contém. Dentro de 1 loop podem figurar vários exit's.

a) Quais seriam as novas produções que acrescentaria à Gramática do Pascal para incluir esta nova construção de controlo.

b) Repare que todas as execuções possíveis da instrução

$$\underline{\text{loop}}\ s_1;\ \underline{\text{if}}\ b\ \underline{\text{then}}\ \underline{\text{exit}}\ \underline{\text{else}}\ s_2\ \underline{\text{endloop}}$$

usando as convenções já introduzidas atrás é representada pela expressão regular:

$$(s_1\ \bar{b}\ s_2)^*\ \ s_1 b$$

Mostre como pode realizar um while com as instruções loop e exit e demonstre a sua correção mostrando com expressões regulares a equivalência entre todas as execuções possíveis do while e da proposta que faz.

c) Identicamente para o repeat

d) Na linguagem ADA (Linguagem do 'Departement of Defense' dos E.U.A.) uma gramática simplificada da instrução loop é a seguinte:

<loop-statement>::=<iteration-specification><basic-
-loop>

<basic-loop>::=
<loop><sequência de statements><endloop>

<iteration-specification>::=
for <loop-parameter><sentido><range> |
while <condição> | λ

<loop-parameter>::=<identifier>

<sentido>::= in | reverse

<range>::=<discrete-type-constant>..<discrete-type-
-constant>

.....

<condição>::=<expressão booleana>

<discrete-type-constant>::= .....

Considerando que em ADA existe uma instrução exit semelhante à anterior, mostre como pode realizar em ADA as instruções for...do, while, repeat do Pascal e ainda qual a potência suplementar da instrução loop em ADA.

13. Dado um ficheiro de texto constituído por várias linhas em que cada linha é uma sequência de caracteres quaisquer e em que por definição a sequência de caracteres

"("< sequência de caracteres quaisquer
diferente de ")" ou "(" > ")"

é considerado um comentário. Mostre qual é a gramática que gera a linguagem que representa todos os possíveis ficheiros que respeitam esta descrição, sabendo ainda que só há 1 comentário por linha no máximo e não há comentários dentro de comentários.

Classifique a gramática obtida.

14.(r) Tome os comandos da classe PRINT, PUNCH, LOAD, COMPILE, COPY e RENAME do computador seu preferido e escreva a gramática que gera a linguagem que representa todas as sessões de trabalho ao terminal com aqueles comandos.

Classifique a gramática obtida.

Chamar a atenção para:

a) os exercícios mostram aplicação não convencional das gramáticas e das linguagens.

b) As gramáticas estruturam as linguagens ao contrário dos EP's e dos autômatos.

c) A gramática não regular $\not\Rightarrow$ linguagem não regular.

3. GRAMÁTICAS ALGÉBRICAS

1. Lema fundamental

Neste parágrafo e no próximo analisa-se mais de perto a estrutura das derivações em gramáticas algébricas. Recordemos que uma gramática $G = (N, T, S, P)$ se diz algébrica se cada produção for da forma $A \to \alpha$ com $A \in N$ e $\alpha \in (N \cup T)^*$. O resultado seguinte, que é praticamente evidente, descreve um facto essencial da estrutura das derivações.

LEMA FUNDAMENTAL. Seja $G = (N, T, S, P)$ uma gramática algébrica. Sejam $\alpha, \beta \in (N \cup T)^*$ e suponhamos que α se pode escrever na forma:

$$\alpha = x_1 \alpha_1 x_2 \alpha_2 \ldots x_n \alpha_n x_{n+1}$$

em que os x's são palavras sobre T e os α's são palavras sobre $N \cup T$. Então $\alpha \overset{*}{\Rightarrow} \beta$ se e só se existirem palavras $\beta_1, \ldots, \beta_n$ sobre $N \cup T$ tais que

$$\beta = x_1 \beta_1 x_2 \beta_2 \ldots x_n \beta_n x_{n+1}$$

e cada $\alpha_i \overset{*}{\Rightarrow} \beta_i$.

Demonstração. 1º Demonstraremos em primeiro lugar que se existirem $\beta_1, \ldots, \beta_n \in (N \cup T)^*$ tais que $\beta = x_1 \beta_1 x_2 \beta_2 \ldots x_n \beta_n x_{n+1}$ e que cada $\alpha_i \overset{*}{\Rightarrow} \beta_i$ então $\alpha \overset{*}{\Rightarrow} \beta$. Para $0 \leq k \leq n$, seja $\alpha^{(k)}$ a palavra que se obtem de α substituindo $\alpha_1, \ldots, \alpha_k$ por $\beta_1, \ldots, \beta_k$ respectivamente, de forma que $\alpha^{(0)} = \alpha$ e $\alpha^{(n)} = \beta$. Se demonstrarmos que $\alpha^{(0)} \overset{*}{\Rightarrow} \alpha^{(1)}, \alpha^{(1)} \overset{*}{\Rightarrow} \alpha^{(2)}, \ldots, \alpha^{(n-1)} \overset{*}{\Rightarrow} \alpha^{(n)}$, teremos que $\alpha = \alpha^{(0)} \overset{*}{\Rightarrow} \alpha^{(n)} = \beta$, visto que a relação $\overset{*}{\Rightarrow}$ é transitiva. Para demonstrar que $\alpha^{(k)} \overset{*}{\Rightarrow} \alpha^{(k+1)}$ escrevamos $\alpha^{(k)} = \alpha' \alpha_{k+1} \alpha''$, $\alpha^{(k+1)} = \alpha' \beta_{k+1} \alpha''$, com $\alpha' = x_1 \beta_1 \ldots \beta_k x_{k+1}$ e $\alpha'' = x_{k+2} \alpha_{k+2} \ldots \alpha_n x_{n+1}$. Dado que, por hipótese, $\alpha_{k+1} \overset{*}{\Rightarrow} \beta_{k+1}$, existe uma derivação $\alpha_{k+1} = \varepsilon_0 \Rightarrow \varepsilon_1 \Rightarrow \ldots \Rightarrow \varepsilon_r = \beta_{k+1}$ de β_{k+1}. Atendendo à definição de derivação directa, conclui-se facilmente que $\alpha' \varepsilon_0 \alpha'' \Rightarrow \alpha' \varepsilon_1 \alpha'' \Rightarrow \ldots \Rightarrow \alpha' \varepsilon_r \alpha''$ é uma derivação de $\alpha^{(k+1)} = \alpha' \beta_{k+1} \alpha'' = \alpha' \varepsilon_r \alpha''$ a partir de $\alpha^{(k)} = \alpha' \alpha_{k+1} \alpha'' = \alpha' \varepsilon_0 \alpha''$. Logo $\alpha^{(k)} \overset{*}{\Rightarrow} \alpha^{(k+1)}$, como se pretendia.

2º Suponhamos agora que $\alpha \overset{*}{\Rightarrow} \beta$ e mostremos que existem $\beta_1, \ldots, \beta_n \in (N \cup T)^*$ tais que $\beta = x_1 \beta_1 x_2 \beta_2 \ldots x_n \beta_n x_{n+1}$ e cada $\alpha_i \overset{*}{\Rightarrow} \beta_i$. A demonstração será feita por indução sobre o comprimento da derivação de β a partir de α.

Seja p o mais pequeno inteiro tal que existe uma derivação de β a partir de α de comprimento p.

Se p=0 então $\beta = \alpha$ e a propriedade é trivial.

Suponhamos a propriedade verificada por todas as palavras que derivam de α por derivações de comprimentos $\leq p-1$, e mostremos que ela também se verifica para β.

Seja $\beta^{(0)} \Rightarrow \beta^{(1)} \Rightarrow \ldots \Rightarrow \beta^{(p)}$ uma derivação de comprimento p de β a partir de α, com $\beta^{(0)} = \alpha$ e $\beta^{(p)} = \beta$. Então $\beta^{(0)} \Rightarrow \beta^{(1)} \Rightarrow \ldots \Rightarrow \beta^{(p-1)}$ é uma derivação de comprimento p-1 de $\beta^{(p-1)}$ a partir de α. Por hipótese indutiva podemos escrever

$$\beta^{(p-1)} = x_1 \beta'_1 x_2 \beta'_2 \ldots x_n \beta'_n x_{n+1}$$

com cada $\alpha_i \overset{*}{\Rightarrow} \beta'_i$, para certos $\beta'_1, \ldots, \beta'_n \in (N \cup T)^*$. Mas $\beta^{(p-1)} \Rightarrow \beta^{(p)}$, o que significa que existe um símbolo não-terminal A ocorrendo em $\beta^{(p-1)}$ e uma produção $A \rightarrow \gamma$ tal que $\beta^{(p)}$ se obtem substituindo essa ocorrência de A em $\beta^{(p-1)}$ por γ. Ora A só pode ocorrer num β'_j, visto que cada $x_i \in T^*$. A substituição de A por γ transforma então β'_j num certo β_j, isto é, $\beta'_j \Rightarrow \beta_j$. Pondo $\beta_i = \beta'_i$ para $i \neq j$, tem-se que $\beta = \beta^{(p)} = x_1 \beta_1 x_2 \beta_2 \ldots x_n \beta_n x_{n+1}$ com cada $\alpha_i \overset{*}{\Rightarrow} \beta_i$, visto que se $i \neq j$ vem por hipótese indutiva $\alpha_i \overset{*}{\Rightarrow} \beta'_i = \beta_i$, e se $i=j$, da hipótese indutiva $\alpha_j \overset{*}{\Rightarrow} \beta'_j$ e de $\beta'_j \Rightarrow \beta_j$ conclui-se que $\alpha_j \overset{*}{\Rightarrow} \beta_j$. Fica terminada a demonstração. □

OBSERVAÇÃO. 1. Aquando da aplicação deste resultado a casos concretos convem ter um certo cuidado para que não se tirem apressadamente conclusões que o lema não permite. Por exemplo, consideremos uma gramática em que A, B ∈ N, a, b ∈ T e $AB \overset{*}{\Rightarrow} ab$. Não se pode concluir que $A \overset{*}{\Rightarrow} a$ e $B \overset{*}{\Rightarrow} b$. Para aplicar o lema, escrevemos AB na forma $x_1 \alpha_1 x_2 \alpha_2 x_3$ com $x_1 = x_2 = x_3 = \lambda$ e $\alpha_1 = A$ e $\alpha_2 = B$. Existem então $\beta_1, \beta_2 \in (N \cup T)^*$ tais que $ab = x_1 \beta_1 x_2 \beta_2 x_3 = \beta_1 \beta_2$ e $A \overset{*}{\Rightarrow} \beta_1$, $B \overset{*}{\Rightarrow} \beta_2$. Porém, a equação $ab = \beta_1 \beta_2$ tem três soluções: 1) $\beta_1 = \lambda$, $\beta_2 = ab$; 2) $\beta_1 = a$, $\beta_2 = b$; 3) $\beta_1 = ab$, $\beta_2 = \lambda$. Por conseguinte, o lema só nos permite concluir que: ou 1) $A \overset{*}{\Rightarrow} \lambda$ e $B \overset{*}{\Rightarrow} ab$; ou 2) $A \overset{*}{\Rightarrow} a$ e $B \overset{*}{\Rightarrow} b$; ou 3) $A \overset{*}{\Rightarrow} ab$ e $B \overset{*}{\Rightarrow} \lambda$.

2. Árvores de derivação

Consideremos a gramática(*)

$$S \rightarrow a\,S\,b\,S \mid b\,S\,a\,S \mid \lambda$$

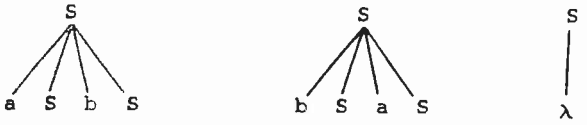
e as duas derivações seguintes da palavra a b a b:

(d_1) $S \Rightarrow a\,S\,b\,S \Rightarrow a\,b\,S\,a\,S\,b\,S \Rightarrow a\,b\,a\,S\,b\,S \Rightarrow a\,b\,a\,b\,S \Rightarrow a\,b\,a\,b$

(d_2) $S \Rightarrow a\,S\,b\,S \Rightarrow a\,S\,b \Rightarrow a\,b\,S\,a\,S\,b \Rightarrow a\,b\,S\,a\,b \Rightarrow a\,b\,a\,b.$

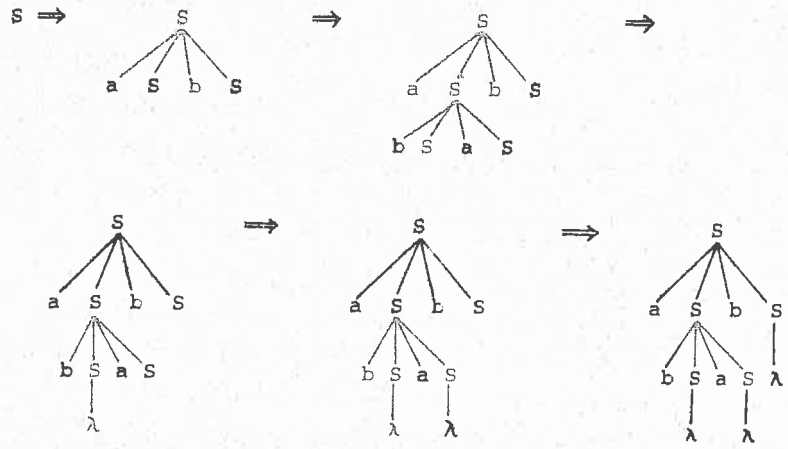
As duas derivações são diferentes porque, atendendo à definição de derivação, são constituídas por duas sucessões distintas de palavras. Contudo, as produções aplicadas numa das derivações são precisamente as mesmas que foram aplicadas na outra, embora por uma ordem diferente, e exactamente aos mesmos símbolos não-terminais. Parece então razoável considerar as duas derivações como sendo em certo sentido "equivalentes". O conceito de "árvore de derivação" permite precisar o sentido desta equivalência. Vejamos como.

Encaremos as produções da gramática como arborescências, como a seguir se indica:



(As setas, no sentido descendente, ficam implícitas.) À derivação d_1 de a b a b corresponde a seguinte sucessão de arborescências:

(*) *Só apresentamos as produções, mas está implícito que* $N=\{S\}$, $T=\{a, b\}$ *e o símbolo inicial é S. Em exemplos subsequentes utilizaremos a mesma convenção.*



A correspondência entre a derivação d_1 e esta sucessão de arborescências é a seguinte:

(i) O número de palavras de d_1 e o número de arborescências da sucessão de arborescências é o mesmo.

(ii) A i-ésima palavra de d_1 obtem-se lendo da esquerda para a direita as etiquetas das folhas da i-ésima arborescência. (Uma folha de uma arborescência é um vértice sem sucessores.)

(iii) Se na passagem da i-ésima palavra de d_1 para a (i+1)-ésima palavra se aplicou uma certa produção a uma certa ocorrência de S, aplica-se a mesma produção á correspondente folha da i-ésima arborescência para se obter a (i+1)-ésima arborescência, no seguinte sentido: substitui-se a referida folha pela arborescência correspondente à produção mencionada

Diz-se então que a última arborescência da sucessão de arborescências é uma "árvore de derivação" da palavra a b a b. (Analogamente, a i-ésima arborescência é uma árvore de derivação da i-ésima palavra de d_1.) Procedendo de modo semelhante para com a derivação d_2, verifica-se que se obtinha a mesma árvore de derivação para a b a b. A ideia intuitiva de que as duas derivações são equivalentes fica assim formalizada com esta noção: duas derivações são equivalentes se originarem a mesma árvore de derivação.

Passemos agora a uma apresentação mais formal destes conceitos. Para esse fim teremos de começar pelo conceito de "arborescência ordenada", que é o que nos vai permitir distinguir entre as arborescências

em que a ordem pela qual são escritas as folhas é importante. Por outro lado, S, a, b não são vértices destas arborescências, visto que, por exemplo, em cada uma delas o símbolo S está associado a três vértices. Estes três símbolos são "etiquetas" das arborescências, e isso vai-nos conduzir ao conceito "arborescência ordenada etiquetada". No que se segue, para simplificar a linguagem, diremos sempre "árvore" em vez de "arborescência ordenada".

Sejam $\mathbb{P}$ o conjunto dos números inteiros (estritamente) positivos, e $\mathbb{P}^*$ o conjunto de todas as sucessões finitas de elementos de $\mathbb{P}$ ("palavras" sobre $\mathbb{P}$), incluindo a sucessão nula Λ. O conjunto $\mathbb{P}^*$ é análogo a T^*, em que T é um alfabeto, com a única diferença que $\mathbb{P}$ é um conjunto infinito; a palavra vazia sobre $\mathbb{P}$ denota-se Λ porque há necessidade de a distinguir de $\lambda \in T^*$. A concatenação de $u, v \in \mathbb{P}^*$ denota-se $u.v$.

Uma <u>árvore</u> é um subconjunto A de $\mathbb{P}^*$ tal que:

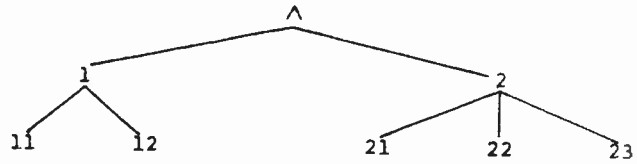
(i) $\Lambda \in A$;

(ii) se $u.n \in A$ então $u \in A$, onde $u \in \mathbb{P}^*$ e $n \in \mathbb{P}$;

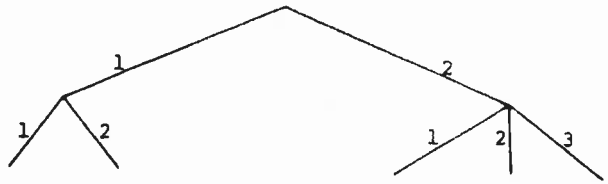
(iii) se $u.n \in A$ então $u.k \in A$ para todo o k, $1 \leq k \leq n$, onde tal como anteriormente $u \in \mathbb{P}^*$ e $n \in \mathbb{P}$.

A terminologia usual para árvores aplica-se a esta definição: os elementos da árvore são os seus <u>nós</u> ou <u>vértices</u>, o elemento Λ é a <u>raiz</u> de qualquer árvore, um <u>arco</u> é um par ordenado da forma $(u,u.n)$, sendo u o <u>predecessor</u> de $u.n$ e $u.n$ um <u>sucessor</u> de u, e uma <u>folha</u> é um vértice sem sucessores. (Note-se que uma árvore pode ter infinitos vértices.)

EXEMPLO 1. O conjunto $\{\Lambda, 1, 2, 11, 12, 21, 22, 23\}$ é uma árvore. Podemos representá-la graficamente do seguinte modo:



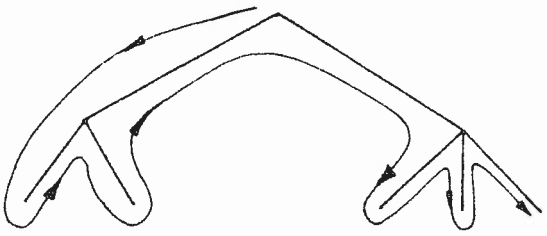
Observa-se que, em virtude da notação escolhida, cada vértice contem informação completa sobre o conjunto dos seus ascendentes: os ascendentes de u são todos os prefixos de u. Analogamente, o mais "próximo" ascendente comum a dois vértices é o seu mais longo prefixo comum. Uma forma mais cômoda de representar a árvore acima é a seguinte:



Aqui, cada arco tem uma etiquta, e cada vértice é a palavra que se obtem concatenando as etiquetas ao longo do único caminho da raiz para esse vértice. Se convencionarmos que, em arcos que partem do mesmo vértice, as etiquetas são escritas por ordem crescente da esquerda para a direita, estas podem mesmo ser eliminadas da representação.

Uma árvore está naturalmente ordenada pela ordem lexicográfica: se u e v pertencerem à árvore então u ocorre antes de v se u for um prefixo de v ou se o elemento de u que ocorre na primeira posição em que u e v diferem for inferior ao correspondente elemento de v. Por outras palavras, escrevendo $u = u_1 \ldots u_n$ e $v = v_1 \ldots v_m$, u ocorre antes de v se: (i) ou $n \leq m$ e $u_i = v_i$ para $i = 1, \ldots, n$; (ii) ou, para algum índice $i \leq \min\{n,m\}$, $u_i \neq v_i$, e se k for o menor tal índice então $u_k < v_k$.

EXEMPLO 2. Na figura seguinte mostra-se a árvore do exemplo anterior e indica-se uma forma de percorrer todos os vértices dessa árvore. Então, um vértice u ocorre antes de um vértice v se for encontrado um primeiro lugar ao longo desse percurso.



Uma árvore etiquetada, com etiquetas num conjunto X, é uma aplicação e: A → X, em que A é uma árvore. Se u ∈ A, e (u) é a etiqueta de u. Vértices distintos podem ter a mesma etiqueta. Apesar disso, confundiremos frequentemente um vértice u com a sua etiqueta e (u) (dizendo por exemplo "o vértice e (u)"), visto que isso simplifica a linguagem e não introduz ambiguidades irremediáveis.

Sejam G= (N, T, S, P) uma gramática. Uma árvore de derivação de G é uma árvore etiquetada d: A → N ∪ T ∪ {λ} tal que

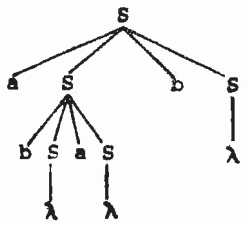
(i) d (Λ)=S;

(ii) se u ∈ A tem k sucessores então existe em P a produção d (u) → d (u.1) d (u.2) ... d (u.k).

A fronteira de uma árvore de derivação é a palavra de $(N \cup T)^*$ obtida concatenando ordenadamente as etiquetas das suas folhas, isto é, se essas folhas forem, pela ordem lexicográfica, $u_1, \ldots, u_n$, então a fronteira é d (u_1) ... d (u_n).

OBSERVAÇÃO 2. Se um vértice de uma árvore de derivação não for uma folha, então a sua etiqueta é necessariamente um símbolo não terminal da gramática. Com efeito, se esse vértice for u e se tiver k sucessores, a produção d (u) → d (u.1) ... d (u.k) só pode pertencer a P se d (u) ∈ N. Mas as etiquetas das folhas podem ser quaisquer, dentro do conjunto N ∪ T ∪ {λ}.

EXEMPLO 3. A árvore

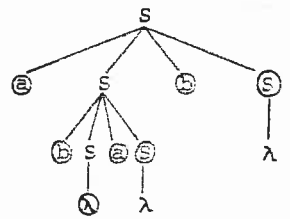


é uma árvore de derivação, com fronteira a b λ a λ b λ = a b a b, da gramática S → a S b S | b S a S | λ.

Mostraremos em seguida que uma árvore de derivação é uma forma adequada para a representação de derivações, no sentido em que a toda a derivação de uma palavra $\alpha \in (N \cup T)^*$ corresponde a uma árvore de derivação de fronteira α, e vice-versa. Para isso precisamos de introduzir dois novos termos.

Seja d uma árvore de derivação de uma gramática algébrica G= (N,T, S, P). Um corte C de d é um conjunto de vértices de d tal que toda a folha de d possui um e um só ascendente em C. Uma fronteira interior de d é a palavra que se obtem concatenando ordenadamente as etiquetas de um corte de d.

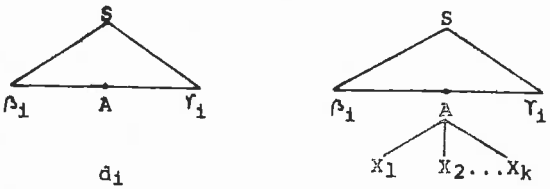
EXEMPLO 4. Na árvore de derivação seguinte (ver exemplo 3), o conjunto dos vértices rodeados por um círculo é um corte, a que corresponde a fronteira interior a b a S b S.



PROPOSIÇÃO 1. Seja G= (N, T, S, P) numa gramática algébrica. Então $S \overset{*}{\Rightarrow} \alpha$ se e só se existir uma árvore de derivação de G com fronteira α.

Demonstração. 1º Suponhamos que $S \overset{*}{\Rightarrow} \alpha$. Seja $S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n = \alpha$ uma derivação de α. Construiremos uma sucessão $d_0, d_1, \ldots, d_n$ de árvores de derivação tal que a fronteira de d_i seja α_i; d_n é então a árvore de derivação procurada. Seja d_0 a árvore de derivação consistindo num único vértice de etiqueta S; é óbvio que a fronteira de d_0 é $S = \alpha_0$. Suponhamos que d_i já foi construída e tem fronteira α_i. Dado que $\alpha_i \Rightarrow \alpha_{i+1}$, podemos escrever $\alpha_i = \beta_i A \gamma_i$, $\alpha_{i+1} = \beta_i X_1 X_2 \ldots X_k \gamma_i$, em que $A \to X_1 X_2 \ldots X_k$ é uma produção em P (cada $X_j \in N \cup T$, ou então k=1 e $X_1 \in N \cup T \cup \{\lambda\}$). Pois bem, d_{i+1} obtem-se de d_i acrescentando-lhe k sucessores ao vértice etiquetado com esta ocorrência de A e atribuindo-lhes ordenadamente as etiquetas $X_1, X_2, \ldots, X_k$. É evidente que d_{i+1} é uma árvore de derivação e a sua fronteira é α_{i+1}. Esta construção é

ilustrada na figura seguinte.



2º Seja agora d uma árvore de derivação de fronteira α. É possível construir uma sequência de cortes $C_0, C_1, \ldots, C_n$ de d tal que

(i) C_0 contem apenas a raiz de d;

(ii) C_{i+1} obtem-se a partir de C_i substituindo um vértice qualquer de C_i pelos seus sucessores;

(iii) C_n é constituído pelas folhas de d.

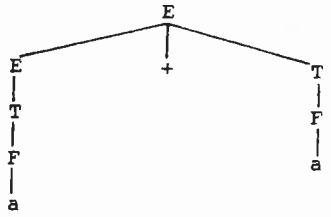
Seja α_i a fronteira interior associada a C_i. Vê-se com facilidade que $\alpha_0 = S$, $\alpha_n = \alpha$ e $\alpha_i \Rightarrow \alpha_{i+1}$. Conclui-se que $S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n = \alpha$ é uma derivação de α. □

Se na demonstração da 2ª parte da proposição C_{i+1} se obtiver de C_i substituíndo o vértice mais à esquerda de C_i que seja não-terminal pelos seus sucessores, então a derivação associada $S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n = \alpha$ é chamada derivação esquerda de α em G. Por outras palavras, $S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n = \alpha$ é uma derivação esquerda de α em G se se poder escrever cada α_i $(0 \leq i < n)$ na forma $x_i A_i \beta_i$ com $x_i \in T^*$, $A_i \in N$ e $\beta_i \in (N \cup T)^*$, e se existir uma produção $A_i \rightarrow \gamma_i$ tal que $\alpha_{i+1} = x_i \gamma_i \beta_i$, i.e., se o símbolo não-terminal de α_i que se utiliza para se obter a derivação directa $\alpha_i \Rightarrow \alpha_{i+1}$ for o que ocorrer mais à esquerda. Define-se de forma análoga derivação direita. A cada árvore de derivação está associada uma e uma só derivação esquerda, e uma e uma só derivação direita.

EXEMPLO 5. Considere-se a gramática (símbolo inicial E)

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid a. \end{aligned}$$

A árvore de derivação seguinte representa 10 derivações equivalentes da palavra a+a.



A derivação esquerda é

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + F \Rightarrow a + a$$

e a derivação direita é

$$E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + a \Rightarrow T + a \Rightarrow F + a \Rightarrow a + a$$

(Note-se que têm ambas o mesmo comprimento.)

Dada uma gramática algébrica G, pode existir uma palavra $x \in L(G)$ que tenha pelo menos duas árvores de derivação distintas. Isto equivale a dizer que existem duas ou mais derivações não equivalentes de x em G. Uma gramática em que isso acontece diz-se ambígua. Uma linguagem algébrica diz-se ambígua se toda a gramática algébrica que a gerar for ambígua.

3. Gramáticas reduzidas

Até ao fim deste capítulo definiremos algumas transformações em gramáticas algébricas que não alteram a linguagem poe elas gerada. As transformações deste parágrafo vão conduzir às chamadas "gramáticas reduzidas", que são essencialmente gramáticas donde foram excluídos todos os símbolos dos alfabetos terminal e não-terminal que, do ponto de vista da linguagem gerada, são inúteis.

Seja G= (N, T, S, P) uma gramática algébrica. Um símbolo não-terminal $A \in N$ diz-se improdutivo se L (A), a linguagem gerada por A, for o conjunto vazio $\emptyset$. Um símbolo inacessível é um símbolo $X \in N \cup T$ terminal ou não-terminal, que não ocorre em nenhuma palavra que derive do símbolo inicial S, isto é, tal que para toda a palavra $\Upsilon \in (N \cup T)^*$ com $S \overset{*}{\Rightarrow} \Upsilon$, X não ocorre em Υ. Um símbolo é dito inútil se for improdutivo ou inacessível. Uma gramática sem

símbolos inúteis diz-se reduzida.

ALGORÍTMO 1.

Dados: Gramática algébrica G= (N, T, S, P).

Determina: Os símbolos improdutivos de G.

Método: 1º Marcar todos os símbolos terminais.

2º Marcar todo o $A \in N$ para o qual exista: ou uma produção $A \rightarrow X_1 X_2 \ldots X_k$, com $X_1, X_2, \ldots, X_k \in N \cup T$, em que $X_1, X_2, \ldots, X_k$ estejam todos marcados; ou a produção $A \rightarrow \lambda$.

3º Se no passo anterior tiver sido marcado pelo menos um símbolo, repetir 2º. De contrário, passar a 4º.

4º FIM. (Os símbolos não marcados são os improdutivos.)

EXEMPLO 6. Consideremos a gramática

$$\begin{aligned} S &\rightarrow A \mid B \mid C \\ A &\rightarrow a \mid b \mid c \\ B &\rightarrow a \mid b \\ C &\rightarrow a\,C \mid a\,C\,B. \end{aligned}$$

Aplicando 1º marcam-se a, b, c. Aplicando 2º marcam-se A, B. Repetindo 2º marca-se S. Aplicando de novo 2º não resulta nenhum símbolo marcado, e termina-se. O único símbolo que ficou por marcar, C, é o único símbolo improdutivo.

Vejamos agora como eliminar os símbolos inacessíveis.

ALGORÍTMO 2.

Dados: C-gramática G= (N, T, S, P).

Determina: Os símbolos inacessíveis de G.

Método: 1º Marcar o símbolo inicial S.

2º Marcar todo o símbolo $X \in N \cup T$ para o qual exista uma produção $A \rightarrow \alpha X \beta$ em que A esteja marcado.

3º Se no passo anterior tiver sido marcado pelo menos um símbolo, repetir 2º. De contrário, passar a 4º.

4º Fim. (Os símbolos não marcados são os inacessíveis.)

Seja a gramática

$$\begin{aligned} S &\rightarrow a\ A \\ A &\rightarrow c \mid A\ b \\ B &\rightarrow a\ b\ d \end{aligned}$$

Aplicando 1º marca-se S. Aplicando 2º marcam-se a, A. Repetindo 2º marcam-se b, c. Aplicando de novo 2º não se consegue marcar mais nenhum símbolo. Os símbolos inacessíveis são portanto B, d.

ALGORÍTMO 3.

Dados: C - gramática G = (N,T,S, P).

Determina: C - gramática G' = (N', T', S', P'), reduzida e tal que L (G') = L (G).

Método: 1º Aplicar a G o algorítmo 1, obtendo N_p (conjunto dos símbolos produtivos de G). Definir $G_1 = (N_p, T, S, P_1)$, em que P_1 contem apenas as produções de P constituídas exclusivamente por símbolos de $N_p \cup T$.

2º Aplicar o algorítmo 2 a G_1, obtendo-se G' = (N', T', S, P'), com N' = símbolos de N_p não inacessíveis, T' = símbolos de T não inacessíveis, e P' = produções de P_1 envolvendo apenas símbolos de $N' \cup T'$.

PROPOSIÇÃO 2: A gramática G' do algorítmo 3 é reduzida e L (G') = = L (G).

Não faremos a demonstração deste resultado, que aliás é intuitivamente óbvio.

Consideremos a gramática G= ({S, A, B}, {a, b}, S, P), em que P consiste em

$$S \rightarrow a \mid A$$
$$A \rightarrow A\ B$$
$$B \rightarrow b.$$

Apliquemos o Algorítmo 3 a G. Aplicando o Algorítmo 1 fica-se com G_1= ({S, B}, {a, b}, S, {S → a, B → b}). Aplicando o Algorítmo 2 a G_1 vem G'= ({S}, {a}, S,{S → a}). NOTA: no Algorítmo 3 não se pode, em geral, inverter a ordem de aplicação dos Algorítmos 1 e 2. Invertendo essa ordem neste exemplo daria a gramática G_1, a qual tem símbolos inúteis.

4. Exercícios

1. Seja G definida por (S - axioma):

$$S \rightarrow AB$$

$$A \rightarrow Aa \mid bB$$

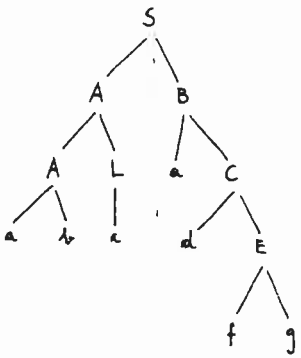
$$B \rightarrow a \mid Sb$$

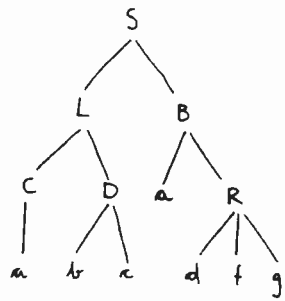
Forneça árvores de derivação para as seguintes palavras:

a) baabaab. b) bBABb. c) baSb

2. Determine uma derivação esquerda e uma derivação direita da palavra baabaab para a gramática do exercício anterior.

3.(r) Considere as seguintes árvores de derivação:





a) Invente uma gramática G tal que estas árvores sejam árvores de derivação de G.

b) Determine a fronteira dessas árvores de derivação. Que se pode concluir quanto à ambiguidade de G ?

4. Mostre que a gramática E -> E+E|E*E|(E)|a é ambígua.

5.(a) Considere as seguintes gramáticas:

G_1:
E -> E + T | T
T -> T * F | F
F -> (E) | D
D -> 0|1|2|3|4|5|6|7|8|9

G_2:
E -> E * F | F
F -> F + T | T
T -> (E) | D
D -> 0|1|2|3|4|5|6|7|8|9

a) Determine uma árvore de derivação em cada uma das gramáticas para as seguintes expressões aritméticas:

4 + 5 * 2
4 * 5 + 2
1 + 2 * 3 + 4
(1 + 2) * (3 + 4)
2 * 3 + 4 * (1 + 5)

b) Um algoritmo que, para avaliar cada uma das expressões anteriores, as analizasse em termos das árvores de derivação obtidas a partir das gramáticas G_1 e G_2, que resultados forneceria em cada um dos casos ?

c) Por analogia com as gramáticas anteriores, escreva uma gramática para definir o conjunto de todas as expressões regulares.

6.(t) Dada uma C-gramática G e w $\in$ L (G), mostre que as seguintes afirmações são equivalentes:

a) w é a fronteira de 2 árvores de derivação distintas de G.
b) w tem 2 derivações esquerdas distintas em G.
c) w tem 2 derivações direitas em G.
d) G é ambígua.

7.(r) Exercício contido na referência bibliográfica (2)

A seguinte gramática definindo assignement statement 's é uma hipotética linguagem de programação:

<assignement statement>::=<identifier>:=<expression>
<expression> ::=<term>+<expression>|<term>
-<expression>|<term>
x<expression>|<term>/<term>
<term> ::=<identifier>|<digit>|(<expression>)
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<identifier> ::= a|b|c|...|3

Um algoritmo que reconhecesse como estrutura de um programa a estrutura indicada pela árvore de derivação, que valores afectaria ao identificador em cada um dos seguintes programas gerados pela gramática anterior

a) a: = 2 + 6 x 3 x 6
b) b: = - (2 - 3 x 2)
c) c: = 2 + (3 - 2 x 6)
d) d: = 2 - 3 - 6

8. Exemplo de como a ambiguidade pode ser resolvida

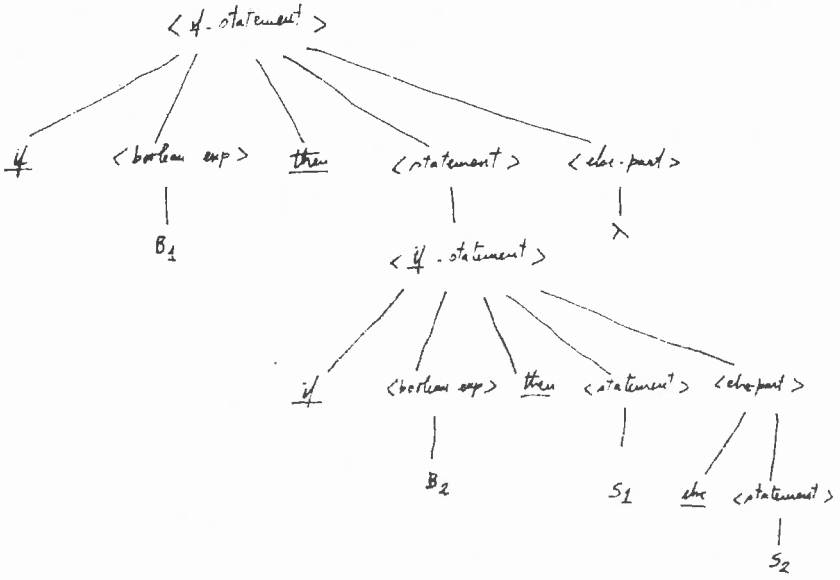
As seguintes produções (presentes por exemplo na linguagem Pascal) conduzem necessáriamente a uma ambiguidade:

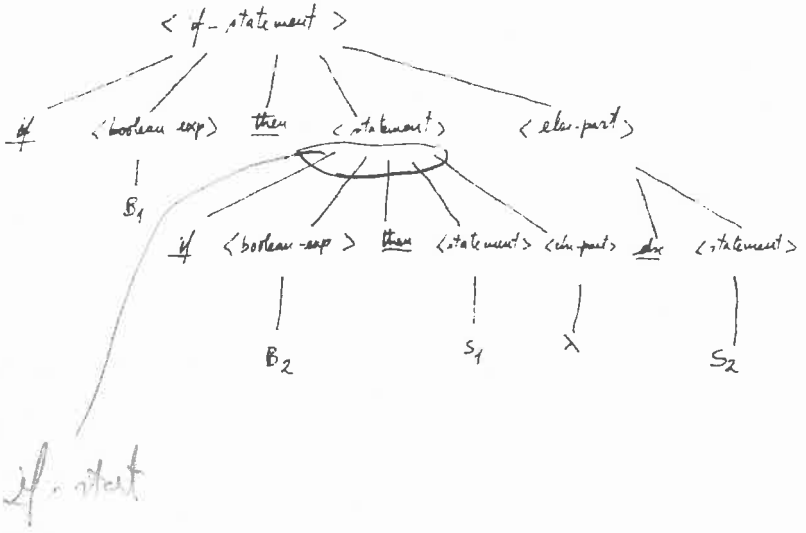
<statement>::=<s_1>|<s_2>.... <s_n>|<if-statement>...
<if-statement>::= if <boolean-exp.>then <statement>
<else-part>
<else-part>::= else <statement>|λ

pois por exemplo, a palavra:

if B_1 then if B_2 then s_1 else s_2

tem 2 árvores de derivação distintas:


<if-statement>
if
<boolean exp>
then
<statement>
<else-part>
B1
<if-statement>
λ
if
<boolean exp>
then
<statement>
<else-part>
B2
S1
else
<statement>
S2


<if-statement>
if
<boolean exp>
then
<statement>
<else-part>
B1
if
<boolean-exp>
then
<statement>
<else-part>
else
<statement>
B2
S1
λ
S2
if-stat

Esta ambiguidade é uma ambiguidade clássica, presente na primeira versão da linguagem ALGOL 60, ela foi resolvida na sintaxe, isto é, a gramática foi transformada de forma a não conter esta ambiguidade. Essencialmente a solução consiste em introduzir as produções: (Apresentadas a penas para ilustrar o processo).

<statement>::=<if-statement>|<outras>
<outras> ::=<while-stat>|<afectação>|<composta>...etc.
<if-statement>::= if <expressão>then<outras><elsepart>
< elsepart>::= else <statement>|λ

Esta consiste em forçar a utilização da instrução composta para dar significado ao else.

Assim:

if B_1 then begin if B_2 then S_1 else S_2 end

é equivalente à ~~segunda~~ primeira árvore.

Na linguagem Pascal no entanto, a ambiguidade na sintaxe permanece, mas, na semântica é dito que else se refere sempre ao último if. Logo, através da semântica a ambiguidade é resolvida, dizendo que a interpretação semântica a dar é sempre a da primeira árvore.

Uma outra solução para esta ambiguidade clássica é introduzida modernamente com as chamadas estruturas de controle sempre parentesadas.

if statement na linguagem ADA:

<if-statement>::= if <condition> then <lista de statements>
<partes else if> <parte else> end if;
<lista de statements>::= <statement>|<statement>;<lista de statements>|λ
<partes else if>::=<ramo else if> <partes else if>|λ
<ramo else if>::= else if <condition> then <lista de statements>
<parte else>::= else <lista de statements>

9. EXEMPLO RESOLVIDO:

Modelo de um Tele carregador:

Um microcomputador de 16 bits e 64 K de memória está ligado a um computador, por uma linha pela qual o micro recebe bytes enviados pelo outro computador.

Um byte especial ∝ assinala que o byte seguinte tem um código de operação. O envio do próprio byte ∝ é transformado no envio de ∝∝.

a) envio de 1 bloco de código a carregar. Para isso é necessário enviar o código de operação ∝A - início de bloco - em seguida 2 bytes com o endereço de início do carregamento - seguidos dos bytes a carregar até um código de operação ∝F - fim de bloco de carregamento.(1)

b) envio de 1 bloco com indicação de endereço de início da execução que tem a forma:

   Código de operação: ∝β, 2 bytes com o endereço de início da execução seguidos de ∝F.

O computador pode enviar vários blocos para carregar, terminados por um único bloco de lançamento da execução.

Escreva a gramática que gera todas as sequências possíveis da comunicação no sentido computador -> micro.

Classifique a gramática e linguagem obtidas.

Assegure que a gramática é "limpa".

Este é um exemplo típico que ganha em clareza se for resolvido em 2 etapas. Uma primeira gramática gera todas as sequências de bytes transmissíveis tratando apenas da sua micro estrutura:

```
<atomo-transmitido>::= <código> | <outros>
<código >::= <abrir-carregar>|<abrir-executar>|<fim>
<abrir-carregar>::=∝A
<abrir-executar>::=∝B
```

---

(1) - Os bytes a carregar são sempre em número par.

<fim> ::= ∝F

<outros>::= <atomo∝> | <simples>

<atomo∝>::= ∝∝

<simples>::= qualquer outro byte diferente de ∝ .

Repare-se que ∝ , A, B, F são designações abstractas para um determinado padrão binário de 8 bits.

Repare-se também, que a seguinte transmissão, do ponto de vista dos atomos transmitidos, não é proibida pela gramática, apesar de não ter significado enquanto transmissão:

$$A\ B\ X_1\ X_2\ X_3\ \ X_4\ \propto F\ X_5\ F\ \ldots$$

No entanto do mesmo ponto de vista

$$X_2\quad \propto K\ \propto A\ \propto B\ \propto F\quad X_1\ \ X_2 \propto F\ \ldots$$

é ilegal pois ∝K é uma sequência de bytes que não pertence à linguagem gerada pela gramática. De uma forma sistemática temos:

<axioma> - <atomo transmitido>

conjunto dos treminais = padrão de 8 bits

" de não treminais = <atomo-transmitido>, <códigos>, <simples>, <atomo ∝>, <outros>, <abrir-carregar>, <abrir-executar>, <fim>

Todas as produções são das formas:

$$A \rightarrow XB \mid X \quad \text{com} \quad X \in T^* \quad \text{e} \quad A, B \in N$$

Logo : a gramática é linear direita.

Quanto à sua "limpeza" é evidente que o é, visto que gera a mesma linguagem que:

<atomo-transmitido> ::= ∝A | ∝B | ∝F | ∝∝ | $\overline{99}$ outro ≠ ∝

que se obtem substituindo alguns não terminais na parte direita das produções sistemáticamente até só termos terminais.

<atomo-transmitido>::= <códigos>|<atomo >|<simples>

⋮

<atomo-transmitido>::= ∝A | ∝B |∝F |<atomo∝ >|<simples

⋮

O método em geral não se pode aplicar (!) senão a gramáticas muito simples.

A gramática foi apresentada com aquele maior número de produções para que seja mais "legível".

Vamos então agora construir outra gramática cujo conjunto terminal está contido no conjunto dos não terminais da primeira e que vai dar a solução final ao problema.

Gramática que denota as transmissões possíveis:

<transmissão>::= <carregamento><lançamento execução>
<carregamento>:: = <segmento><carregamento>|λ
<segmento>::= abrir-carregar outros outros
<segmento-programa> fim
<segmento-programa>::= outros <segmento-programa>|λ
<lançamento-execução>::= abrir-execução outros outros
fim

A gramática que gera por exemplo a palavra

abrir-carregar outros outros outros outros ...
outros fim

abrir-execução outros outros fim

Se outros denotar um byte simples da outra gramática (∝ ou ≠) e se substituirmos estes terminais pelos terminais respectivos

da outra gramática obtemos a seguinte sequência de bytes

$$\propto A \; X \; X \; X \; X \; X \; \ldots \; X \propto F \propto B \; \lambda \; X \propto F$$

que é uma palavra correspondente a uma transmissão correcta.

Esta 2ª gramática também denota uma linguagem regular e tem por:

axioma: <transmissão>

Não terminais: <carregamento>, <lançamento-execução>
<segmento>, <segmento programa>

Terminais: abrir-carregar, outros, abrir-execução, fim

É fácil verificar que as gramáticas estão limpas (Faça isso!).

NOTAS COMPLEMENTARES - FORA DO CONTEXTO DO CAPITULO:

Finalmente um aspecto que importa desde já chamar a atenção, é o seguinte:

Ver-se-á adiante que sempre que se está em presença de uma gramática context-free limpa(1) se podem programar directamente algorítmos que reconhecem se as palavras pertencem a linguagem gerada pela gramática e que simultâneamente reconhecem a estrutura da derivação que conduziu à mesma. Em princípio a cada regra da gramática corresponderá um procedure que trata dessa produção. Nesses procedimentos pode-se inserir o código que realiza as acções "semânticas" correspondentes. Por exemplo, o procedure correspondente a «lançamento-execução», além de reconhecer um bloco deste tipo, executaria no fim, a acção semântica Branch incondicional para o endereço denominado por <outros> <outros>.

É assim que a legibilidade de uma gramática está em profun-

---

(1)- Para o algoritmo ser simples e eficaz é também necessário garantir outras condições o que de uma forma geral se consegue fácilmente aplicando algumas transformações à gramática se isso for necessário.

da relação com a legibilidade do algoritmo, assim como a solução em "camadas" de gramáticas corresponde profundamente às soluções por "camadas" de software.

O método da programação descendente por refinamentos sucessivos, consiste em dar uma solução global a um problema, pressupondo grandes operações primitivas que de facto não existem na linguagem de programação que se usa. Na continuação do desenvolvimento do algoritmo essas operações vão sendo sucessivamente pormenorizadas em outras de mais baixo nível e assim sucessivamente. Um programa é tanto mais legível quanto a sua estrutura reflicta o traço do seu processo de desenvolvimento.

Na análise de um programa que reconhecesse o input vindo do computador começaríamos por:

```
início
   enquanto houver segmentos para carregar fazer
                                          carregar;
   lançar a execução;
fim.
```

A certa altura do processo de desenvolvimento do programa teríamos necessidade de recorrer a uma operação de leitura de um código de input. É claro que poderíamos ler directamente o byte recebido mas é muito mais interessante dispor-se de um procedimento

```
get-next-atom (var B: Byte,var tipo: integer);
```

que resolvesse todos os pequenos aspectos que têm a ver com os problemas provocados pelo ∝ como prefixo. Ora este procedimento está em profunda relação com a primeira gramática pois o trabalho que ele realiza é exactamente o de reconhecer os atomos transmitidos no input e se ele fosse desenhado sistemáticamente ele teria procedimentos internos relacionados com as produções dessa gramática.

Neste caso concrecto a gramática não se poderia usar na forma em que está. A título meramente ilustrativo apresenta-se a gramática que poderia ser usada para a programação:

<atomo-transmitido>::= ~~ff~~ byte ≠ de ∝ | ∝<resto>
<resto>::= A|B|F|∝

Assim a partição em "camadas" de gramáticas está profundamente relacionada com a partição em "camadas" de software ou com a partição de um programa em sub-problemas. Resta responder à questão: porquê começar pela gramática apresentada em 1º lugar se ela é o refinamento final do problema ? A resposta é simples: O método dos refinamentos sucessivos não é para aplicar como um "dogma" ! Para certas categorias de problemas reconhece-se logo à partida que vai ser necessário um conjunto de procedimentos para tratar outro sub-problema que se identifica fácilmente. Impor que o desenho poderia começar por aí seria puro não "academicismo". A programação exige sempre muita experiência e invenção!

Para terminar resta dizer que uma solução do problema em uma só gramática, pode ser fácilmente obtida introduzindo na 2ª gramática as produções da 1ª que descrevem os "terminais" outros, abrir-execução, abrir-carregamento e fim. O que sob o ponto de vista da programação corresponde a uma substituição de procedimentos pelas instruções que estas representam.

10.(r) EDITOR INTERACTIVO

Um editor interactivo de texto, encara o ficheiro a editar (ficheiro de caracteres) como se todas as linhas do ficheiro estivessem todas numeradas de 1 em diante ... e que o ficheiro a editar reside em memória (Vêr a analogia com as funções de edição de um interpretador de BASIC).

Os comandos possíveis são os seguintes (inclui a lógica do seu efeito).

a) especificar o ficheiro a editar:

F nome do ficheiro

A linha corrente é a primeira linha do ficheiro.

b) inserir uma linha antes da linha corrente:

I sequência de caracteres

A linha corrente não se altera.

c) Suprimir a linha corrente:

S

A linha corrente é a que a segue.

d) Saltar relativamente linhas:

$\pm$ inteiro B

+ pode ser omitido para a frente

inteiro pode ser omitido se tiver o significado de 1.

A linha corrente é a nova.

e) Saltar absoluto:

J inteiro

A linha corrente passa a ser a próxima indicada.

f) Mostrar a linha corrente ou várias linhas para a frente:

inteiro M

inteiro é opcional quando tem o significado de 1.

A linha corrente não é alterada.

g) Terminar a edição:

Sabendo que todos os comandos começam no início da linha e que terminam com o caracter "carriage-return" que pode representar por CR, construa a gramática que gera todas as sessões de trabalho "sintácticamente correctas" com este editor.

Impõe-se ainda que todas as sessões de trabalho começam com o comando F.

O conjunto dos simbolos terminais é o conjunto dos caracteres.

Classifique a gramática obtida.

11. Analizador lexicográfico da linguagem Pascal

Descreva a gramática que gera todas as sequências de componentes elementares de programas da linguagem Pascal independentemente de estarem sintácticamente correctas ou não, sob o ponto de vista de programas.

<atomo-Pascal>::= <simbolo-especial>|<identificador>|
<palavra-chave>|<valor>|<comentário>
<valor>::= <string>|<inteiro-sem-signal>|<real-sem-signal>
<simbolo-especial>::= ":="|"="|"<>"|".."|"."|">="|"+"|...
etc.
<palavra-chave>::= "array"|"end"|"begin"|.....etc
<identificador>::= <letra><resto-id>
<resto-id> ::= <letra><resto-id >|<digito><resto-id>|λ
<letra> ::= A|B|C ...|Z|a|b|c ...|
<digito> ::= 0|1|2 .... |9
<comentário> ::=<inicio-coment><meio-coment><fim-coment>
<inicio-coment> ::= "(*"|"{"
<fim-coment> ::= "*)"|"}"
<meio-coment> ::= qualquer caracter não contendo a sequência *) nem o caracter "}"
<inteiro> ::= <digito> <resto>
<resto> ::= <digito> <resto>|λ
<real-sem-sinal>::= <cabeça-real> <expoente>
<cabeça-real>::= <inteiro-sem-sinal>"."<inteiro-sem-sinal>
.....
etc.

12.(r) Ficheiro de empregados

Um programa destina-se a actualizar um ficheiro de empregados obedecendo às seguintes declarações:

```
Meses = (JAN, FEV, MAR, ABR, MAI, JUN, JUL, AGO, SET,
                                        OUT, NOV, DEZ);
Categ = (EMPREG, TECNIC, ADMINIST);
fila - empregados = file of
                        Record

                    numero: integer;
                    nome, morada: array  1..30  of clear;
                    vencimento: real;
                    caetgoria: categ;
                    dataadmissão: Record
                                    dia, ano: integer;
                                    mes: meses
                                end
                    end;
```

O programa deve ser capaz de fazer diversas espécies de alterações como supressão, inserção de um novo empregado, assim como diversos tipos de modificações.

O input para o programa deve ser extremamente legível e ser constituído por um texto de que nas linhas seguintes se dá um exemplo.

Repare-se que no exemplo a formatação, isto é, a disposição no papel é indiferente.

Exemplo de input:

```
                SUPRIMIR
            404   FIM   MODIFICAR
        502   VENCIMENTO:= 10 000.00 FIM

    INSERIR 904 NOME:= 'JOSÉ SILVA'  MORADA:=
    'RUA DO LA VAI UM'  VENCIMENTO:= 14 000.00
    CATEGORIA:= ADMINIST EM 10 AGO 79  FIM
    MODIFICAR 1034  NOME:= 'ANTONIO NOBRE'
    MORADA:= 'RUA ALTA 34'   FIM   STOP
```

Descreva uma gramática que represente todos os input's possíveis deste programa. Classifique a gramática obtida e "limpe-a".

Sugestões:

Tal como no exemplo do Tele carregador considere 2 gramáticas a 1ª das quais transforma o input em atomos elementares, por exemplo:

```
<atomo-input>::= <string>|<palavra-chave>|<valor-mês>
     <valor-categoria>|<simbomo-especial>|<inteiro>...
<string>::= "'"   sequência de caracteres sem "'"  "'"
<palavra-chave>::= "SUPRIMIR"|"FIM"|...
    :
<simbolo-especial>::= ":="
    :
```

Repare-se que os aspectos semânticos como o de os empregados estarem ordenados crescentemente ou as moradas terem menos de 30 caracteres não são tratados pela gramática.

13.(r) Árvore Geneológica

Um programa destina-se a construir e consultar uma árvore geneológica. Cada individuo é caracterizado na árvore pelo seu nome. As operações possíveis são:

```
INSERIR-ANCESTRAL   nome
NASCEU  nome  FILHO-DE  nome
   :
PAI-DE?  nome
AVO-DE?  nome
IRMAOS-DE? nome
FILHOS-DE? nome
FINALIZAR
```

Sendo todas as operações cujo identificador é terminado por? opearções de consulta.

Descreva uma gramática que representa todos os input's possíveis deste programa. Cassifique a gramática obtida e "limpe-a".

Considere as mesmas sugestões que no Ficheiro de emprega dos.

14.(a) FORMATADOR DE PROGRAMAS PASCAL

Desenhe uma gramática, cuja estrutura permita uma vez introduzidas acções semânticas nos procedimentos de reconhecimento das produções, formatar automáticamente um programa Pascal sintácticamente correcto e no qual não existem instruções case, nem labels nem comentários.

Sugestões:

Considere como conjunto terminal o conjunto de alguns não terminais do analizador lexicográfico de Pascal. As simplicações são por exemplo considerar que todas as palavras chave que não são significativas para a formatação passam a identificadores, por exemplo: **label** **case** **of**...

Depois reescreva a gramática do Pascal considerando apenas as partes que são significativas para a formatação.

Por exemplo:

```
<programa-Pascal>::= program <lista atomos>")";
                     <parte declaraçôes>
                     <bloco> "."
        :
<bloco>::=  begin <lista-instruções> end
<lista-instruções>::= <instrução><resto>
<resto>::=  ";" <instrução><resto>|λ
<instrução> ::= <if>|<outras>
        :
```

4. AUTÓMATOS FINITOS

1. Definição

Um autómato finito (AF, para abreviar) M é definido por uma lista

$$M = (Q, T, \delta, q_0, F),$$

em que

Q é um conjunto finito não vazio de estados;

T é o alfabeto de entrada;

δ é uma aplicação de QxT em Q, dita de transição de estado;

$q_0 \in Q$ é o estado inicial;

$F \subset Q$ é o conjunto dos estados finais ou de aceitação.

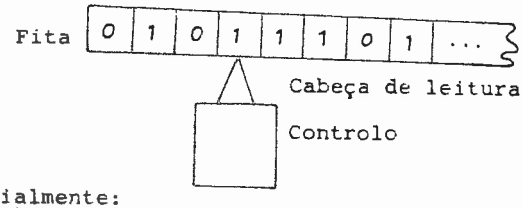
Note-se que se pode ter $F=\emptyset$ ou $F=Q$, embora ambas as situações sejam desinteressantes (como se verá mais tarde).

2. Interpretação

Vamos interpretar um AF como um "dispositivo físico", e ver como ele "funciona". A formalização desse funcionamento é feita no parágrafo 4.

Uma "fita" semi-infinita está dividida em quadrados ("células"), cada um dos quais pode conter um símbolo de T. Quando o AF está "em operação", uma palavra de T^* está inscrita na fita, ocupando as primeiras células e deixando as restantes em branco.

Existe ainda uma "cabeça de leitura", acoplada a uma "unidade (finita) de controlo". A cabeça de leitura lê os símbolos de T na fita da esquerda para a direita, lendo um símbolo da cada vez. O conjunto dos estados da unidade de controlo é Q ("estados ou configurações de memória"), a qual se encarrega de prescrever as transições (ou mudanças) de estado tendo em conta o símbolo de entrada lido pela cabeça de leitura.



Inicialmente:

a) a cabeça de leitura está colocada sobre a célula mais à esquerda da fita, lendo o primeiro símbolo da palavra de T* inscrita na fita (se essa palavra for λ , a cabeça de leitura não lê nada);

b) a unidade de controlo está no estado q_0.

Em certa etapa do funcionamento:

a) estando a unidade de controlo num estado q_0 e lendo a cabeça de leitura um símbolo de entrada a, a unidade de controlo transita para o estado $p = \delta(q_0, a)$ (se não restar nenhum símbolo para ler, a unidade de controlo permanece no mesmo estado; assim; se a palavra inicialmente inscrita na fita for λ, a unidade de controlo mantem-se no estado q_0);

b) em seguida, a fita desloca-se de uma célula para a esquerda.

Vejamos como "funciona" o AF quando na fita está inscrita a palavra $x_1\ x_2\ \ldots\ x_n$. Inicialmente o AF está no estado q_0 e lê o símbolo x_1. Muda então para o estado $q_1 = \delta(q_0, x_1)$ e a fita desloca-se de uma célula para a esquerda, passando a cabeça de leitura a ler o símbolo x_2. Transita em seguida para o estado $q_2 = \delta(q_1, x_2)$, e assim sucessivamente.

Quando a cabeça de leitura lê x_n o AF está num estado q_{n-1}, transitando para o estado $q_n = \delta(q_{n-1}, x_n)$. Dois casos se podem dar: se $q_n \in F$, diz-se que o AF aceitou ou reconheceu a palavra $x_1\ x_2\ \ldots\ x_n$; 2) se $q_n \notin F$, o AF rejeitou a dita palavra. O conjunto das palavras de T* reconhecidas por M é a linguagem reconhecida por M, que se denota L (M). Como já ficou dito, a formalização destes conceitos é feita no parágrafo 4.

3. Representações

Tabelas de estado

Seja o AF $M = (Q, T, \delta, q_0, F)$ dado por:

$Q = \{q_0, q_1, q_2, q_3\}$

$T = \{0, 1\}$

δ: $(q_0, 0) \mapsto q_2$ $\quad (q_0, 1) \mapsto q_1$

$(q_1, 0) \mapsto q_3$ $\quad (q_1, 1) \mapsto q_0$

$(q_2, 0) \mapsto q_0$ $\quad (q_2, 1) \mapsto q_3$

$(q_3, 0) \mapsto q_0$ $\quad (q_3, 1) \mapsto q_2$

q_0 - estado inicial

$F = \{q_0, q_2\}$.

Podemos representá-lo pela seguinte tabela de estado:

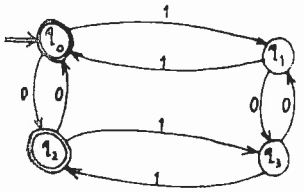
		0	1
$\Rightarrow$	q_0	q_2	q_1
	q_1	q_3	q_0
	q_2	q_0	q_3
	q_3	q_1	q_2

A seta dupla $\Rightarrow$ representa o estado inicial, e os estados rodeados por um círculo são os estados finais.

Diagramas ou grafos de estado

São grafos cujos vértices correspondem aos estados, existindo um arco de q para p com a etiqueta a se $\delta(q, a) = p$. O estado inicial é indicado por uma seta dupla $\Rightarrow$, e os estados finais representando os respectivos vértices por 2 círculos. Para

o exemplo anterior tem-se:



Existe uma correspondência biunívoca entre as palavras não vazias de T^* e os caminhos do grafo de estado que partem do estado inicial. O percurso desse caminho corresponde ao funcionamento do modelo físico do AF. Uma palavra não vazia de T^* é aceite pelo autómato se e só se o caminho que lhe corresponde terminar num estado final. A palavra vazia é aceite se e só se o estado inicial q_0 for também final, como sucede neste exemplo.

4. Extensão de δ

Define-se recursivamente $\delta^*: Q \times T^* \rightarrow Q$ por:

$$\delta^*(q, \lambda) = q$$

$$\delta^*(q, a\,x) = \delta^*(\delta(q,a), x),$$

quaisquer que sejam $q \in Q$, $a \in T$ e $x \in T^*$. Para o exemplo dado no parágrafo 3, tomemos $01001 \in \{0,1\}^*$ e calculemos $\delta^*(q_0, 01001)$:

$$\begin{aligned}
\delta^*(q_0, 01001) &= \delta^*(\delta(q_0, 0), 1001) = \delta^*(q_2, 1001) \\
&= \delta^*(\delta(q_2, 1), 001) = \delta^*(q_3, 001) \\
&= \delta^*(\delta(q_3, 0), 01) = \delta^*(q_1, 01) \\
&= \delta^*(\delta(q_1, 0), 1) = \delta^*(q_3, 1) = \delta^*(q_3, 1\lambda) \\
&= \delta^*(\delta(q_3, 1), \lambda) = \delta^*(q_2, \lambda) \\
&= q_2.
\end{aligned}$$

O cálculo de δ^* é a formalização do funcionamento do "modelo físico" do AF, bem como do percurso do caminho no grafo de estado. Note-se que a sucessão de estados $q_0, q_2, q_3, q_1, q_3, q_2$ é o caminho do grafo de estado determinado pela palavra 01001.

Seja $w \in T^*$; $\delta^*(q, w)$ é o estado para o qual o AF transita após a leitura de todos os símbolos de w. Se $w = \lambda$, não há símbolos para ler, e o AF permanece no estado q; se $w = a\,x$ ($a \in T$, $x \in T^*$), vê-se primeiro o estado para o qual o AF transita

após a leitura de a, e em seguida o estado para o qual ele transita após a leitura de x.

Note-se que para todo o $q \in Q$ e todo o $a \in T$ se tem $\delta^*(q, a) = \delta(q, a)$. Com efeito, se $p = \delta(q, a)$, $\delta^*(q, a) = \delta^*(q, a\lambda) = \delta^*(\delta(q, a), \lambda) = \delta^*(p, \lambda) = p$.

Por isso se diz que δ^* é uma extensão de δ. Daqui por diante representaremos δ e δ^* pelo mesmo símbolo 'δ', para simplificar a escrita.

Diz-se que:

M aceita ou reconhece $x \in T^*$ se $\delta(q_0, x) \in F$;

M rejeita $x \in T^*$ se $\delta(q_0, x) \notin F$.

(Assim, a palavra 01001 no exemplo mais acima é aceite pelo AF em questão.)

A linguagem aceite ou reconhecida por M é:

$$L(M) = \{ x \in T^* : \delta(q_0, x) \in F \},$$

isto é, é o conjunto das palavras reconhecidas por M. Vê-se que se $F = \emptyset$ se tem $L(M) = \emptyset$, se $F = Q$ então $L(M) = T^*$, e $\lambda \in L(M)$ sse $q_0 \in F$.

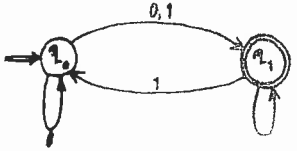
5. Autómato finito não determinista

Um autómato finito não determinista (AFND, para abreviar) define-se tal como um AF,

$$M = (Q, T, \delta, q_0, F),$$

com a única diferença que δ aplica $Q \times T$ em 2^Q (conjunto dos subconjuntos de Q). O "funcionamento" de um AFND não tem uma descrição tão intuitiva. O mais simples é supor que, dados $q \in Q$ e $a \in T$, ele transita "simultaneamente" de q para todos os estados de $\delta(q, a)$ (se $\delta(q, a) = \emptyset$, o que é uma possibilidade, não há nenhuma transição).

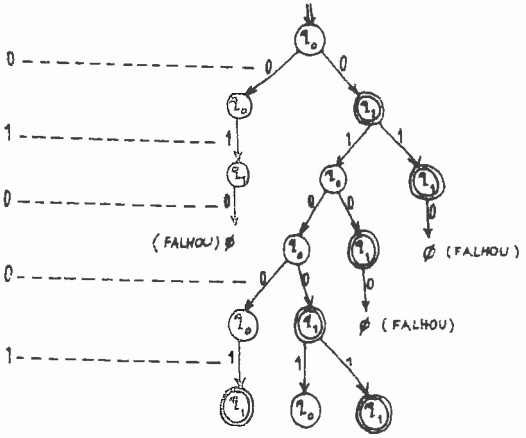
Como exemplo, consideremos o AFND seguinte:



Tem-se

$$\delta(q_0, 0) = \{q_0, q_1\} \qquad \delta(q_0, 1) = \{q_1\}$$
$$\delta(q_1, 0) = \emptyset \qquad \delta(q_1, 1) = \{q_0, q_1\}.$$

Consideremos o seu "funcionamento" para a palavra de entrada 01001. Em vez de ser dado por um "grafo linear" (constituído por uma sucessão de estados), tem-se a seguinte arborescência:



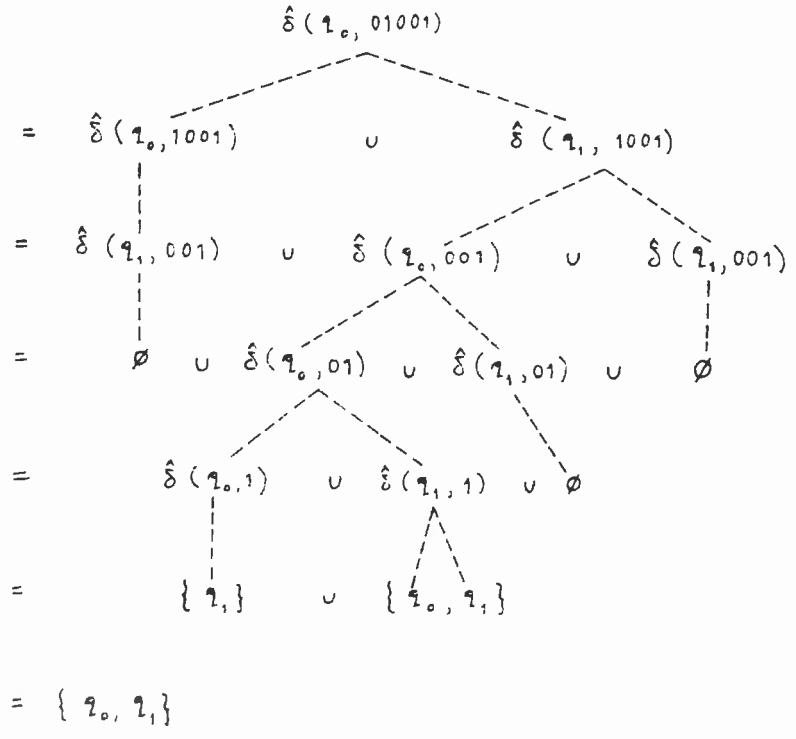
Os vértices do último nível da arborescência representam os estados para os quais o AFND transita após ter lido a palavra 01001, e que são q_0 e q_1. Como um destes estados é final (o q_1) diz-se que o AFND aceitou ou reconheceu a palavra 01001.

Para definir formalmente estas ideias, é preciso estender a definição de δ, tal como foi feito para os AF's.

Define-se $\hat{\delta} : Q \times T^* \rightarrow 2^Q$ recursivamente por:

$$\hat{\delta}(q, \lambda) = \{q\}$$
$$\hat{\delta}(q, a\,x) = \bigcup_{p \in \delta(q,a)} \hat{\delta}(p, x)$$

quaisquer que sejam $q \in Q$, $a \in T$ e $x \in T^*$. Para o exemplo anterior tem-se (comparar com a arborescência):



Como de costume, escrevemos δ em vez de $\hat{\delta}$. Uma palavra $x \in T^*$ é aceite ou reconhecida por um AFND se

$$\delta(q_0, x) \cap F \neq \emptyset.$$

A linguagem reconhecida por um AFND M é

$$L(M) = \{ x \in T^* : \delta(q_0, x) \cap F \neq \emptyset \}.$$

Um AF é um caso particular de um AFND. Dado um AF $M = (Q, T, \delta, q_0, F)$ define-se um AFND $M' = (Q', T', \delta', q_0', F')$ tal que $Q' = Q$, $T' = T$, $q_0' = q_0$, $F' = F$ e $\delta'(q, a) = \{p\}$ se $\delta(q, a) = p$. Vê-se facilmente que $L(M) = L(M')$. Isto mostra que para todo o AF existe um AFND que reconhece a mesma linguagem que ele. No sentido inverso, o resultado seguinte mostra que toda a linguagem reconhecida por um AFND também pode ser reconhecida por um AF. Por conseguinte, a introdução dos AFND's não traz nada de novo de um ponto de vista teórico, cifrando-se porém numa maior comodidade em certos casos.

TEOREMA 1. Se $L \subset T^*$ é aceite por um AFND, então L é aceite por um AF.

Demonstração . Seja L aceite por um AFND $M = (Q, T, \delta, q_0, F)$, isto é, $L = L(M)$. Defina-se um AF $M' = (Q', T', \delta', q_0', F')$ por

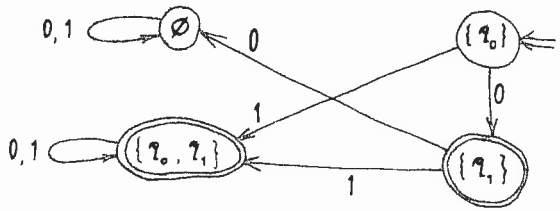
$$Q' = 2^Q \qquad T' = T$$

$$\delta'(K,a) = \bigcup_{q \in K} \delta(q, a)$$

$$q'_0 = \{q_0\} \qquad F' = \{K \in 2^Q : K \cap F \neq \emptyset\}.$$

Vê-se com facilidade que $\delta'(q'_0, x) = \delta(q_0, x)$ para todo o $x \in T^*$. Logo $x \in L(M')$ sse $\delta'(q'_0, x) \in F'$ sse $\delta'(q'_0, x) \cap F \neq \emptyset$ sse $\delta(q_0, x) \cap F \neq \emptyset$ sse $x \in L(M)$. □

Para o AFND dos exemplos anteriores, o AF correspondente é o seguinte:



6. Autómato minimal

Dada uma linguagem $L \subset T^*$ reconhecida por um AF, pretende-se determinar neste parágrafo um AF com um número mínimo de estados que reconheça L. Tal AF existe sempre e chama-se autómato minimal. (Este estudo restringe-se a AF's deterministas.)

Partindo de um AF M, determinaremos um AF minimal M' tal que $L(M) = L(M')$, em duas etapas:

1º Determina-se o autómato "monogénico" $M(q_0)$ de M, em que q_0 é o estado inicial de M.

2º Determina-se o autómato reduzido de $M(q_0)$.

Autômato monogênico

O autômato monogênico de $M = (Q, T, \delta, q_0, F)$ é o autômato $M(q_0) = (Q_0, T_0, \delta_0, q_0, F_0)$, em que:

$Q_0 = \{ q \in Q : \exists\, x \in T^* \text{ tal que } q = \delta(q_0, x) \}$ é o conjunto dos estados de Q "atingíveis" a partir de q_0;

$T_0 = T$;

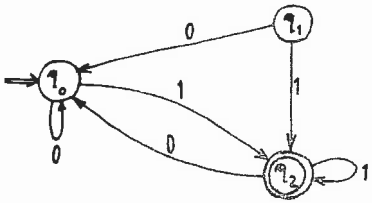
$\delta_0(q, a) = \delta(q, a)$ para todo o $q \in Q_0$ e todo o $a \in T$;

$q_0 = q_0$;

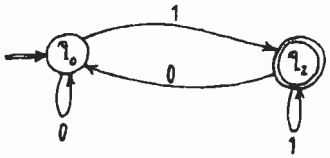
$F_0 = F \cap Q_0$.

Vê-se fácilmente que $L(M(q_0)) = L(M)$, isto é, $M(q_0)$ e M reconheceu a mesma linguagem. (Pense nas representações de M e $M(q_0)$ em grafos de estados.)

EXEMPLO. O autômato monogênico de



é



Ou seja com o autômato monogênico retiram-se os estados inacessíveis.

Autômato reduzido

Seja $M = (Q, T, \delta, q_0, F)$ um AF. Sejam $q, q' \in Q$. Ponhamos $q \equiv q'$ e $q \equiv_k q'$ para "q é equivalente a q' " e "q é k-equivalente a q' " respectivamente. As definições são:

$q \equiv q'$ sse $\forall_{x \in T^*}$ $\delta(q, x)$ e $\delta(q', x)$ pertencem ambos a F ou ambos a Q-F;

$q \equiv_k q'$ sse $\forall_{x \in T^*}$ $|x| \leq k \Rightarrow$ $\delta(q, x)$ e $\delta(q', x)$ pertencem ambos a F ou ambos a Q-F.

Assim, $q \equiv q'$ significa que é indiferente em qual destes estados M se encontra, visto que o seu comportamento a partir de qualquer deles é o mesmo do ponto de vista da linguagem que M reconhece. Se $q \equiv_k q'$, porém, só se garante o mesmo comportamento nas próximas k etapas.

É óbvio que

$$q \equiv q' \text{ sse } \forall_{k \geq 0} \; q \equiv_k q'.$$

Estas duas relações são de equivalência, e portanto induzem partições em Q. Sejam π e π_k as partições induzidas por $\equiv$ e $\equiv_k$ respectivamente. O nosso objectivo é calcular π; se mencionamos os π_k é porque estes vão auxiliar a calcular π, do seguinte modo:

Calculam-se π_0, π_1, π_2, ... até que para um certo k_0 se tenha $\pi_{k0} = \pi_{k0+1}$. Nessa altura $\pi = \pi_{k0}$.

O facto de que existe k_0 tal que $\pi_{k0} = \pi_{k0+1}$ resulta de que:

1º $\pi_0 = \{F, Q-F\}$ tem 2 blocos (estamos a supor que $F \neq \emptyset$ e $K-F \neq \emptyset$; a não ser assim todos os estados seriam equivalentes entre si e ter-se-ia $\pi = \{Q\}$).

2º É fácil de ver que $q \equiv_{k+1} q' \Rightarrow q \equiv_k q'$. Assim, os blocos de π_{k+1} estão contidos em blocos de π_k. Se $\pi_{k+1} \neq \pi_k$, os blocos de π_{k+1} são mais numerosos que os de π_k. Como este número não pode aumentar indefinidamente (o nº máximo de blocos é n - nº de estados do AF - quando cada bloco tem 1 estado), para um certo k_0 virá $\pi_{k0+1} = \pi_{k0}$. (Tem-se $k_0 \leq n-2$ visto que, na pior das hipóteses, parte-se de π_0 com 2 blocos, π_{k0} tem n blocos, e na passagem de cada π_k para π_{k+1} aumenta-se em uma só unidade o nº de blocos.).

Para calcular sucessivamente π_0, π_1, ..., é mais conveniente uma definição diferente (mais equivalente) de $\equiv_k$, por indução em k:

$q \equiv_0 q' \Longleftrightarrow$ ($q \in F$ e $q' \in F$) ou ($q \notin F$ e $q' \notin F$)

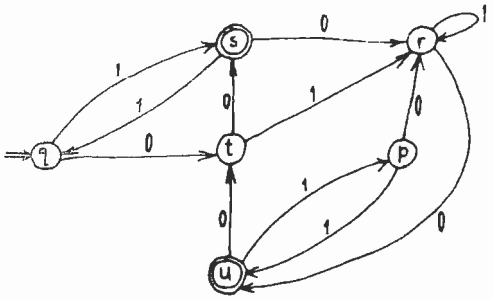
$q \equiv_{k+1} q' \Longleftrightarrow q \equiv_k q'$ e $\delta(q, a) \equiv_k \delta(q', a)$ $\forall a \in T$.

Demonstra-se por indução em k que as duas definições são equivalentes. Assim:

1º Faz-se $\pi_0 = \{F, Q-F\}$.

2º Calcula-se π_{k+1} a partir de π_k do seguinte modo: para cada bloco B de π_k, tomam-se todos os pares de estados $q, q' \in B$; se, para todo o $a \in T$, $\delta(q, a)$ e $\delta(q', a)$ pertencerem a um mesmo bloco de π_k, então q e q' pertencem a um mesmo bloco de π_{k+1}; se não, q e q' pertencem a diferentes blocos de π_{k+1}.

EXEMPLO. Seja o AF



O seguinte quadro mostra os vários π_k:

π_0	π_1	π_2
p	p	p
q	q	q
r	r	r
t	t	t
s	s	s
u	u	u

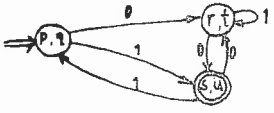
Os blocos das diversas partições estão indicadas por meio de separações por traços horizontais. Como $\pi_1 = \pi_2$ tem-se $\pi = \pi_1$. Por exemplo, q e r, que estão no mesmo bloco de π_0, fi

caram separados em π_1 porque t= δ(q, 0) e u= δ(r, 0) não estão no mesmo bloco de π_0; já p e q estão no mesmo bloco de π_2 visto que r= δ(p, 0) e t= δ(q, 0) estão no mesmo bloco de π_1, assim como u= δ(p, 1) e s= δ(q, 1).

Denotemos por [q] o bloco de π que contém q. O autômato reduzido de M é $M_r = (Q_r, T_r, \delta_r, q_{0r}, F_r)$ em que

$$Q_r = \pi$$
$$T_r = T$$
$$\delta_r = ([q], a) = [\delta(q, a)]$$
$$q_{0r} = [q_0]$$
$$F_r = \{[q]: q \in F\}.$$

EXEMPLO. O autômato reduzido do AF anterior é:



7. Autômatos e gramáticas lineares

TEOREMA 2. Seja G = (N, T, S, P) uma gramática linear. Então existe um autômato finito M = (Q, T, δ, q_0, F) tal que L (M)= L (G).

Demonstração. Define-se um AFND M tal que L (M) = L (G) do seguinte modo:

Q = N$\cup${A}, em que A é um novo símbolo que não pertence a N;

T;

δ (A, a) = $\emptyset$ ($\forall$ a $\in$ T);

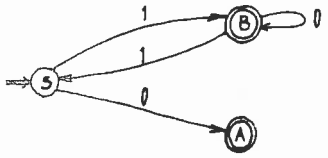
$$\delta(B, a) = \begin{cases} \{C: B \longrightarrow a\ C \in P\} \cup \{A\} & \text{se } B \longrightarrow a \in P \\ \{C: B \longrightarrow a\ C \in P\} & \text{se } B \longrightarrow a \notin P; \end{cases}$$
$$(\forall B \in N, a \in T)$$

q_0 = S;

F = {A}$\cup${B: B $\longrightarrow \lambda \in$ P}.

EXEMPLO. Seja a gramática S $\longrightarrow$ 1B | 0, B $\longrightarrow$ 0B | 1S | λ. O AFND correspondente é:

É aqui que se deve introduzir as aplicações dos autómatos ?



TEOREMA 3. Seja $M = (Q, T, \delta, q_0, F)$ um autómato finito. Existe uma gramática linear $G = (N, T, S, P)$ tal que $L(G) = L(M)$.

Demonstração. Podemos supor, pelo teorema 1, que M é determinista. G define-se do seguinte modo:

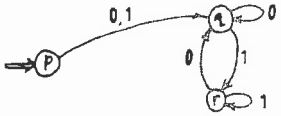
$$N = Q;$$

$$T;$$

$$S = q_0;$$

$$P = \{p \to a\,q : \delta(p,a) = q\} \cup \{p \to a : \delta(p,a) \in F\}$$
$$\cup \{p \to \lambda : p \in F \text{ e } p \text{ inicial}\}.$$

EXEMPLO. Dado o AF



a gramática correspondente é a seguinte:

$$N = \{p, q, r\}, \quad T = \{0, 1\}, \quad S = p$$

$$P : p \to q0 \mid 1_q$$
$$\to 0q \mid 1r \mid 1$$
$$r \to 0q \mid 1r \mid 1.$$

8. Expressões regulares

Seja T um alfabeto. Uma "expressão regular" sobre T é uma expressão que se obtem com o auxílio dos símbolos $\emptyset$ e λ, dos símbolos de T e dos operadores +, * e de concatenação. Cada expressão regular denota um subconjunto de T^* (ou seja, uma lingua gem sobre T).

As definições são as seguintes:

1) $\emptyset$ é uma expressão regular (ER) que denota o subconjunto vazio de T^*.

2) λ é uma ER que denota $\{\lambda\}$.

3) Para todo o $a \in T$, a é uma ER denotando $\{a\}$.

4) Seja p, q ER's denotando respectivamente $P \subset T^*$ e $Q \subset T^*$. Então:

   a) p q é uma ER e denota PQ.
   b) p+q é uma ER e denota $P \cup Q$.
   c) p* é uma ER e denota P^*.

5) Uma ER é qualquer expressão que se obtenha por um nº finito de aplicações de 1, 2, 3 e 4.

EXEMPLOS DE ER's

i) a* denota $\{a\}^* \subset T^*$, por 3 e 4-c).

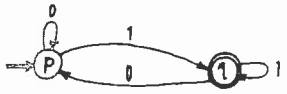
ii) a+b denota $\{a,b\} = \{a\} \cup \{b\}$, por 3 e 4-b)

iii) a* (a+b) denota $\{a\}^* \{a,b\}$, por i), ii) e 4-a)

TEOREMA 4. Seja M um AF. Então L (M) pode ser denotado por uma ER.

Demonstração. Associa-se um sistema de equações ao AP e resolve-se por eliminação usando o facto de que a solução de $X = X\alpha + \beta$ é $X = \beta\alpha^*$.

EXEMPLO



$$L_p = \{ x \in T^* : \delta(p, x) = p \}$$

$$L_q = \{ x \in T^* : \delta(p, x) = q \} = L(M).$$

$$\text{Sistema} \begin{cases} L_p = L_p 0 + L_q 0 + \lambda \\ L_q = L_p 1 + L_q 1 \end{cases}$$

Diremos que 2 ER's são iguais se denotam o mesmo conjunto. Tem-se as igualdades, sendo x, y, z ER's

$$\emptyset\, x = x\, \emptyset = \emptyset \qquad\qquad x + x = x$$

$\emptyset + x = x + \emptyset = x$

$\lambda x = x \lambda = x$

$x (y + z) = x y + x z$

$(x + y) z = x z + y z$

Com isto o sistema anterior resolve-se do seguinte modo:

$L_p = (Lq0 + \lambda) o^* = Lq00^* + 0^*$

$Lq = Lq 00^*1 + 0^*1 + Lq1 = Lq(00^*1+1) + 0^*1$

$\Rightarrow Lq = 0^*1 (00^*1+1)^*$

Donde L (M) é denotado por $Lq = 0^*1 (00^*1+1)^*$.

9. Exercícios

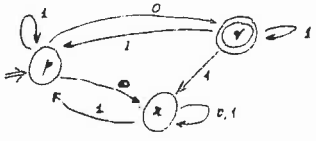
1. Considere as seguintes gramáticas:

$$G_1: S \to a \mid a\ T \qquad G_2: S \to a\ S \mid bT \mid \lambda$$
$$T \to b\ T \mid b \qquad\qquad T \to b\ T \mid \lambda\ .$$

   a) Determine $L(G_1)$ e $L(G_2)$.

   b) Determine autômatos finitos (deterministas) M_1 e M_2 tais que $L(M_1) = L(G_1)$ e $L(M_2) = L(G_2)$.

2. Determine uma expressão regular para a linguagem reconhecida pelo seguinte autômato finito (não determinista):



3. Determine um autômato finito determinista que reconheça a linguagem denotada pela seguinte expressão regular:

$$[(\lambda + x)\ (xy + y)^*]^*.$$

4. Determine a gramática limpa da seguinte gramática:

$S \to AF$, $A \to BA \mid cd$, $B \to SB \mid a$, $C \to CD$, $D \to eE \mid Fb$, $E \to e\ F \mid bFSe$, $F \to AS \mid B$.

5. Dada a gramática:

$$P \to bP \mid aR \mid \lambda$$
$$Q \to aP \mid aQ \mid bR$$
$$R \to bQ \mid a$$

Determine uma expressão regular que denote a mesma linguagem que a gramática gera. Em seguida determine autômato finito que reconheça aquela linguagem. Determi-

ne também um autômato finito determinista e a partir deste um autômato reduzido que reconheça aquela linguagem.

5. PROGRAMAÇÃO PELA SINTAXE UTILIZANDO LINGUAGENS REGULARES

1. A programação de um autómato finito determinista

Para programar um algoritmo que "lê" uma palavra reconhecida por um autómato finito determinista e a percorre, simulando o processo de leitura do autómato, deve-se proceder da seguinte forma:

a. Introduzir um tipo que represente o conjunto dos terminais.

```
type   conjunto - terminal = ...;
```

por exemplo:

```
type   conjunto - terminal = char;
```

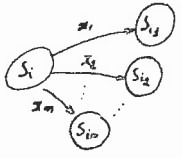
e uma variável global t deste tipo.

b. Introduzir uma operação que permite ler um terminal da entrada ("input"):

```
procedure  ler-terminal (var t: conjunto-terminal);

begin ... end;
```

c. Para cada estado Si com a forma:



em que Sij denota o estado que se segue a Si se na entrada estiver x_j, introduzir um procedimento com a forma:

```
procedure  Si;
   begin
       if t = x1 then
              ler-terminal (t);
              S1
       else if t = x2  then
               ler-terminal (t);
               Si2
       else if ...
          :
          :
       else ERRO
       end if
    end (* Si *);
```

NOTA: O ramo else ERRO será explicado a seguir e numa primeira aproximação pode ignorar-se

d. A estrutura do algoritmo é:

```
procedure  automato-finito-determinista;
    type  conjunto-terminal = ...;
    var t: conjunto-terminal; (*simbolo corrente*)
    procedure ler-terminal (var t: conjunto-terminal);
            begin ... end;
    procedure  S1;  (* estado inicial*)
            begin ... end;
           :
    procedure  Sn;
            begin ... end;
    begin
        ler-terminal (t);
        S1
    end (* automato-finito-determinista*);
```

2. Considerações sobre o algoritmo

a. Conjunto terminal distinto de um tipo pré-definido da linguagem de implementação

Muitas vezes sucede que o conjunto terminal é distinto de um tipo pré-definido na linguagem. Por exemplo: conjuntos dos não terminais definidos por:

```
<identificador> ≡ letra (digito|letra)*
<número> ≡ digito (digito)*
<string> ≡ "'" caracter* "'"
    ⋮
```

Neste caso colocam-se os seguintes sub-problemas:

a.1 forma do conjunto-terminal ...
a.2 forma de if t = x_i ...
a.3 forma do procedure ler-terminal (...);

a.1. Uma solução possível é introduzir o tipo:

```
type  classe-terminal = (identificador,numero,string,
                                                ...);
      conjunto-terminal = Record
                          classe: classe-terminal;
                          valor: ...
                          end;
```

O campo valor é introduzido apenas quando o procedimento ler-terminal, pode desde logo, por uma questão semântica, calcular um valor eventualmente associado ao terminal. Por exemplo, o "valor" de <numero>.

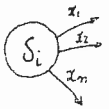
a.2. As expressões t = x_i podem tomar a forma:

```
if t. classe = identificador ...
```

a.3. Finalmente, o procedimento tem a função anterior, com a complexidade suplementar de determinar a classe a que o terminal pertence e eventualmente o seu valor.

b. Considerações sobre a cadeia IF ... ELSE IF...

Para um estado do automato com a forma:



várias situações podem suceder:

b.1. O conjunto $x_1, x_2, \ldots, x_n$ é um sub-conjunto estricto do conjunto-terminal.

Neste caso acontece que nem todos os símbolos que aparecem na entrada, são legais. Esta situação particular pode ser tratada das seguintes formas:

Introdução de um estado suplementar (ERRO), que aparece em todos os ramos ELSE de todos os estados e que é o estado que trata o problema do erro. Este estado, também pode ser especificado ao nível do automato, neste caso força-se a igualdade entre $x_1, x_2, \ldots, x_n$ e o conjunto-terminal, logo não existe ramo ELSE na cadeia e esta transforma-se fácilmente num CASE.

Na outra hipótese, para não sobrecarregar o automato, é a nível do algoritmo que este novo estado e respectivo procedimento é introduzido, sendo este sempre chamado no ramo ELSE.

Em qualquer dos casos, ERRO é chamado sem a acção prévia: ler-terminal. ERRO é um estado de excepção.

Nas linguagens em que o case statement só é bem definido se existem ramos para todos os valores do tipo da expressão (caso do Pascal standard) a cadeia IF ... ELSE IF não pode, ser transformada num CASE, de uma forma imediata.

b.2. Por outro lado, a presença de um estado Sij=Si, (presença de um lacete) introduz uma chamada recursiva de Si. Esta pode ser fácilmente transformada em iterativa, pois devido ao contexto onde é feita não existe qualquer necessidade de memorização do contexto, a não ser o de estado seguinte que é constante. A transformação assume a forma:

```
while t = xj do ler-terminal (t);
```

Estes ramos iterativos vêm à frente da cadeia IF ELSE IF , a qual, por razões idênticas às anteriores, deve permanecer em Pascal standard.

b.3. Finalmente, uma optimização evidente é introduzida no caso em que vários Sij coincidem entre si. Os ramos assumem então as formas:

```
ELSE IF (t = xi) or (t = xj) or ... THEN
  ou
ELSE IF  t  in [ xi, xj, ... ] THEN,
```

errado!

```
WHILE  (t = xi) or (t = xj) or ... THEN
  ou
WHILE  t  in [xi, xj, ...] THEN,
```

c. Terminação do algoritmo

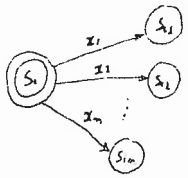
Consideremos 2 situações distintas:

c.1. Os estados finais não têm sucessore e podem por tanto ser reduzidos a um só, que se traduz no procedimento:

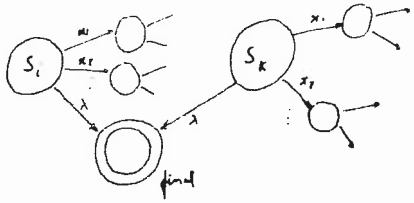
```
procedure final;
    begin
    end (*final*);
```

E a análise pára. Os estados que chamam final não avançam a entrada antes desta chamada.

c.2. Existem vários estados finais em que alguns têm sucessores. Neste caso os estados finais têm a forma geral



e podem ser transformados no anterior introduzindo não determinismo:



Na prática este não determinismo pode ser transformado em determinista introduzindo um valor suplementar em conjunto-terminal, valor que representa o fim do input e a que chamaremos Fim-input. Fim-input é o terminal devolvido por ler-terminal quando o input terminou.

Os estados S_i e S_k com esta forma, testam o ter

minal corrente t, se o seu valor for Fim-input chamam o estado final sem avançar prèviamente a entrada.

Um exemplo pode ser a expressão

ELSE IF t = Fim-input, tomar a forma

ELSE IF t = ∅

quando se usa um ∅ para assinalar o fim de uma sequência de inteiros.

d. Tratamento de erros

Um determinado problema pode, por definição, não considerar a hipótese de surgirem erros no processo de análise. Neste caso, o critério segurança do algoritmo e o provérbio:

"Mesmo quando foi provado que nada pode correr mal, alguma coisa vai correr mal"

recomendam a introdução do procedimento ERRO que a seguir se apresenta e cuja execução é desencadeada no ramo ELSE de todas as cadeias IF ... ELSE IF de todos os estados.

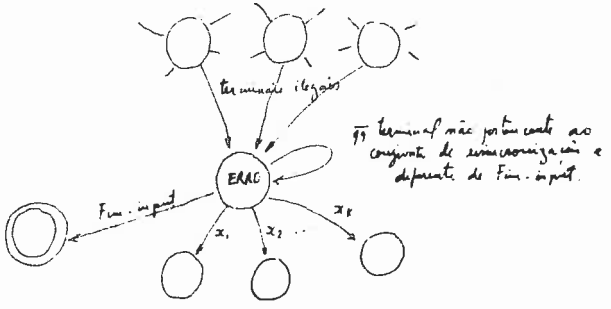
```
procedure ERRO;
    begin
        writeln ('terminal inesperado
                no input', t)
        {abortar programa}
    end (*ERRO*);
```

Assim, quer no caso de o enunciado do problema não estar correcto, quer no caso de o programa conter erros, o algoritmo tem sempre resultados correctos ou explicitamente errados mas, nunca imprivisíveis.

Noutros casos, o problema exige tratamento de erros. A solução a adoptar é extretamente dependente do mesmo, da sua semântica e do contexto da análise.

Uma solução possível consiste em, ao determinar que o terminal na entrada não é legal, imprimir uma mensagem de erro a notificar esse facto, de seguida os terminais da entrada são desprezados até um terminal que permita a resincronização da entrada com o processo de análise.

Este solução consiste em introduzir um estado suplementar no automato, ERRO, com a forma:



Este novo estado pode potencialmente conduzir directamente ao estado final pois, uma das formas possíveis de erro é a terminação abrupta do input[1] Por isso, uma chamada de ERRO também não deve avançar o input. Uma chamada típica será:

```
ELSE IF t = x   then
  :
else assinalar-erro (N, t);
     ERRO
end if;
```

(1) - Na verdade Fim-input tem sempre de pertencer a qualquer conjunto de resincronização.

assinalar-erro (N, t) é um procedimento exterterno ao processo da análise sintáctica e que permite, se tal for desejável, assinalar exter namente os erros. Adiante se verá que este pro cedimento é já uma função semântica.

O procedimento correspondente ao estado ERRO toma então a forma:

```
procedure ERRO;
  begin
    while (not t in conjunto-resincronização)
          and (t<> Fim-input) do ler-ter
          minal (t);
    if t = Fim-input then
       assinalar-erro (100, t)   (1)
    else if t = x1 then
       ler-terminal (t);
       S1
    else if .
            .
            .
    else if t = xk then
       ler-terminal (t);
       Sk
    else (* nada *)

    end  if

  end  (* ERRO *);
```

Uma solução mais geral consiste em introduzir tantos estados de erro quantos os conjuntos de resincronização que o problema e a sua semântica recomendam.

---

(1) - mensagem 100 = 'fim do input inesperado'

Com o comportamento apresentado é evidente que o comportamento do automato é bem definido seja qual fôr a entrada.(1) O facto de esta não ser legal é assinalado pelas passagens pelos estados de erro com eventual terminação em final. Cada passagem por caminhos não de acordo com o reconhecimento da linguagem que o automato inicial reconhece, são assinaladas pela função semântica:

assinalar-erro (N, t)

Na prática começa-se por obter o automato que reconhece a linguagem que nos interessa. Se o problema da terminação se puser, então introduz-se Fim-input. Finalmente em função do contexto do problema concreto introduz-se um tratamento de erros adequado, o qual pode ser mais ou menos complexo em função das necessidades do mesmo. De qualquer forma o automato a implementar deve ter um comportamento bem especificado para qualquer palavra de input pertencente a (conjunto-terminal)*.

d. Transformação de todo o algoritmo num algoritmo iterativo.

Analisando o algoritmo precedente, verificamos que não é só no caso da presença do lacete que se introduz recursividade. Sempre que no automato existir qualquer ciclo, (associado necessàriamente a um operador * numa expressão regular que denote a mesma linguagem que o automato) a chamada de um procedimento, pode eventualmente desencadear um conjunto de chamadas que por sua vez termine nele próprio.

---

(1) - Isto é, este automato reconhece: (conjunto terminal)*.

Uma análise mais detalhada, conduz-nos a verificar que o algoritmo é potencialmente altamente recursivo. Em si, este aspecto não coloca nenhum problema, pois, como os procedimentos envolvidos não têm parâmetros, nem variáveis locais, a sua execução nem degrada a velocidade de execução, nem ocupa demasiada memória. No entanto, como estes algoritmos poderão ter necessidade de ser programadas em linguagens sem recursividade(1), atendendo a que a sua passagem a iterativos não introduz nenhuma degradação da legibilidade e como essa passagem é fácil, vale a pena analisar-mo-la.

---

(1) - Apesar de o FORTRAN, COBOL ou ASSEMBLER não permitirem recursividade, a verdade é que, a entrada numa sub-rotina e o seu retorno, é muitas vezes implementado nestas linguagens empilhando o endereço. No algoritmo que vimos, dado que não são utilizadas variáveis locais ou parametros, poderiam portanto ser usadas aquelas linguagens.

Em qualquer caso, estamos a trabalhar numa zona não standard da linguagem o que é extremamente perigoso (Isto na hipótese de o compilador ser tão mau que deixasse passar!).

No assembler, dado que se tem acesso directo à forma como as "calling sequences" se processam, é uma hipótese a encarar.

Para passarmos o algoritmo a iterativo, verificamos que o carácter determinista do automato, implica que apenas se tem que conhecer em cada passo, o próximo procedimento a executar.[1]

Esse valor pode ser guardado numa única variável global.

Introduz-se então o tipo:

```
type conjunto-estados = (inicial, ..., ...,
                                  erro, final);
```

e a variável

```
var estado: conjunto-estados;
```

Inicialmente, estado é inicializada com o valor inicial. O algoritmo toma então a forma:

```
procedure automato-finito-determinista;
    type conjunto-estados = (inicial, ...,
                                  final, erro);
        conjunto-terminal = ...;
    var t: conjunto-terminal;
        estado: conjunto-estados;
    procedure ler-terminal (...); begin
                                  ... end;
    begin
        estado: = inicial;
        ler-terminal (t);
        while estado <> final do
```

(1) - A execução do algoritmo anterior, traduz-se num contínuo empilhar do estado seguinte até que finalmente um estado final é atingido. Dá-se então um desempilhamento abrupto dos retornos.

```
case estado of
  inicial:
    if t = x1 then
      ler-terminal (t);
      estado: = S1
    else if t = x2 then
      ler-terminal (t);
      estado: = S2
    else if ...
      :
      :
    else if t = Fim-input then
      estado: = final
    else  assinalar - erro (N);
          estado: = erro;
    end if;
  :
  :
  ERRO: ...
  final:;
end case
end (* automato-finito-determinista*);
```

NOTA: final:; foi introduzido por uma questão de completude.

O algoritmo é então construído de forma análoga ao anterior, sendo cada ramo do "case" global, um bloco construído exactamente da mesma forma que o corpo dos procedimentos anteriores, com cada chamada de procedimento substituído por uma afectação à variável estado do valor do estado seguinte.

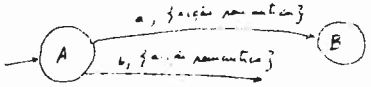
Todas as observações válidas naquele contexto, quer para a cadeia de IF'S, quer para o estado final, quer para o tratamento de ERROS são absolutamente válidas no novo algoritmo.

3. Automatos com funções semanticas

Um automato, em si, é uma máquina meramente reconhecedora da sintaxe. Como tal, o seu interesse é muito reduzido. As suas principais aplicações são feitas introduzindo acções semânticas.

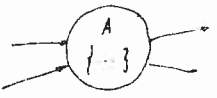
Assim, dado o estado corrente e o terminal presente no input é determinada a estrutura da palavra já reconhecida e a partir da mesma podem-se executar acções que, para o problema dado, se associam ao automato.

Uma notação possível para assinalar essas acções semânticas no automato, consiste em introduzir a seguinte representação para os arcos

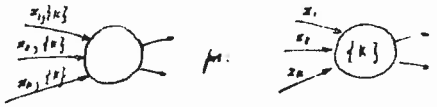


que se pode ler: Se o automato estiver no estado A e na entrada encontrar o símbolo a, então executar a acção semantica e depois passar ao estado B.

Outra notação possível é inserir a acção semantica no próprio estado



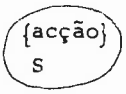
que se pode ler: ao entrar no estado A, executar a acção semantica ... e continuar a análise. Trata-se de uma forma de abreviar a situação:



As acções semanticas devem ser inseridas nos algoritmos anteriores, exactamente nas posições correspondentes.

a, {acção} ⟶

A acção é inserida antes de avançar a entrada e executar a passagem ao estado seguinte.



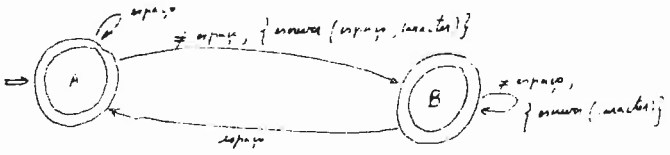
A acção é inserida à cabeça do procedimento S ou a entrada do ramo do case global.(1)

As invocações de assinalar-erro () são um exemplo de inserção de acções semanticas.

4. Pequeno exemplo

Um programa tem por entrada 1 linha de texto e por saída a mesma linha com todas as sequências de espaços repetidos transformados num só espaço.

Automato com funções semanticas:



type conjunto-terminal = char; (*acrescido por fim-input*)

ambos os estados são finais e em todos os estados

$\{x_1, x_2, \ldots, x_n\}$ = Conjunto-terminal

---

(1) - Repare-se que neste caso t já não contém o terminal que conduzia ao estado S!

```
programa suprimir-brancos (input,output);
   const Fim-input = char (13); (*código interno de
                                  carriage return*)
     var t: char;
     procedure ler-terminal (var t: char);
         begin
            if eoln (input) then t: = Fim-input else
                                              read (t)
         end;
     procedure A;
         begin
              if t = '␣' then begin
                                  ler-terminal (t);
                                  A
                              end
              else if t = Fim-input then fim
              else begin
                        write('␣'; T);
                        ler-terminal (t);
                        B
                   end
         end;
     procedure B;
         begin
              if t = '␣' then begin
                                  ler-terminal (t);
                                  A
                              end
              else if t = Fim-input then fim
              else begin
                        write (t);
                        ler-terminal (t);
                        B
                   end
         end;
```

```
procedure fim; begin end;
begin
     ler-terminal (t);
     A;
     writeln
end (*suprimir-brancos*).
```

Repare-se que se se introduziu o procedimento ler-terminal ao invés de se usar directamente read (t) e eoln (input). Este aspecto deve-se ao facto de em Pascal uma linha se percorrer por:

```
(*reset (input) - implícito*)
while not eoln (input) do
    begin
         read (t);
         (*tratamento de t*)
    end;
```

enquanto todos os algoritmos dos automatos estão escritos pressupondo que esse percurso se faria por:

```
ler-terminal (t)
while t <> Fim-input do
    begin
         (*tratamento de t*)
         ler-terminal (t)
    end
```

ou seja t = Fim-input só é verdade quando se tentou ler o fim da linha, o que não é verdade em Pascal usando read (t), pois quando se "lê" o último caracter da linha antes do seu fim, eoln passa a verdade.

Utilizando, no entanto, directamente o input↑ como t, get (input) como ler-terminal (t) e eoln (input) como t = Fim-input e tomando em consideração o programa anterior obtem-se uma nova versão mais compacta:

```
procedure A;
        begin
             if eoln (input) then fim
             else if input↑ = '␣' then begin get
                                       (input); A end
             else begin
                       write ('␣', input↑);
                       get (input);
                       B
                  end
        end;

procedure B;
        begin
             if eoln (input) then Fim
             else if input↑ = '␣' then begin
                                       get (input);
                                       A
                                       end
             else begin
                       write (input↑);
                       get (input);
                       B
                  end
        end;
```

```
    procedure fim; begin end;
    begin
        A;
        writeln
    end (*suprimir-brancos*).
```

Nas cadeias IF ... ELSE IF a ordem dos testes não é indiferente!

Uma versão iterativa seria:

```
program suprimir-brancos (input, output);
  const Fim-input = chr (13);
   type conjunto-estados = (A, B, fim);
    var t: char; S: conjunto-estados;
    procedure ler-terminal (var t: char); begin
                                          ... end;
    begin
        ler-terminal (t); S: = A;
        while S <> fim do
         case S of
            A:
              if t = '␣' then begin
                                  ler-terminal (t);
                                  S: = A
                              end
              else if t = Fim-input then S: = fim
              else begin
                       write ('␣', t);
                             ler-terminal (t);
                             S: = B
                   end;
```

```
            B: .... ;
            fim:;
        end (*case*);
        writeln
    end (* suprimir-brancos *).
```

O leitor atento reparará que todas as soluções introduzem 1 espaço suplementar se a linha começar por um caracter diferente de espaço.

5. Problema do Telecarregador - algoritmos

(Ver problema 9 do capítulo 3)

A gramática de baixo nível definia o conjunto dos terminais da gramática de alto nível. ESte conjunto é o sub-conjunto do conjunto dos não terminais da gramática de baixo nível:

$$\{\langle\text{abrir-carregar}\rangle, \langle\text{abrir-executar}\rangle, \langle\text{fim}\rangle, \langle\text{outros}\rangle\}$$

A gramática de baixo nível é:

$\langle$átomo$\rangle$::=$\langle$outros$\rangle$ | $\langle$fim$\rangle$ | $\langle$abrir-executar$\rangle$ | $\langle$abrir-carregar$\rangle$

$\langle$abrir-carregar$\rangle$::= $\propto$ A

$\langle$abrir-executar$\rangle$::= $\propto$ B

$\langle$fim$\rangle$::= $\propto$ F

$\langle$outros$\rangle$::= $\propto\propto$ | $\overline{qq}$ byte $\neq$ de $\propto$.

O procedimento ler-terminal, da gramática de alto nível, é um automato que lendo bytes do input de baixo nível atinge um estado final sempre que reconhece um daqueles átomos.

Para obtermos esse automato transformamos a gramática até obtermos a expressão regular que denota a mesma lin-

guagem:

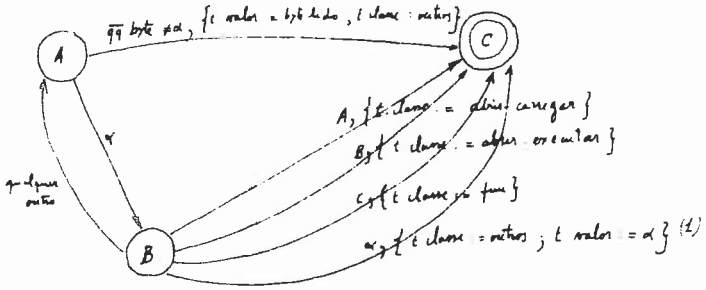
$$\alpha\,(A + B + F + \neg\alpha) + \overline{qq}\ \text{byte} \neq \alpha$$

Assim a gramática de alto nível trabalhará sobre

```
type classe-terminal = (outros, abrir-carregar,
                        abrir-executar, fim);
     conjunto-terminal = Record
                          classe: classe-terminal;
                          valor: byte
                         end;
```

O procedimento ler-terminal dessa gramática é modeliza do pelo seguinto automato:

```
procedure ler-terminal (var t: conjunto-terminal)
```



(1) porque a semantica de 2 α's seguidos é que o valor transmitido é α.

No exemplo introduzimos um tratamento de erros que consiste em desprezar o caracter errado.

Este automato tem um único estado final e seja qual for o estado em que está, existe um estado seguinte para qualquer que seja o byte no input. Em cada activação de ler-terminal o automato arranca no estado A e quando terminar em C colocou em t o terminal colectado do input.

```
procedure ler-terminal (var t: conjunto-terminal);
    const ' = ...;
          A = ...;
          B = ...;
          F = ...;
     type conjunto-estados = (stA, stB, stC);
      var tt: byte; s: conjunto-estados;
    begin
        s: = stA;
        ler (tt);
        while s <>stC do
           case s of

                StA: if tt<>' then begin
                                     t.valor:=tt;
                                     t.classe:=outros;
                                     s:=stC
                                   end
                               else begin
                                     ler (tt);
                                     s:=stB
                                   end;

                StB:
                    if tt = A then begin
                                     t.classe:=abrir-carregar;
                                     ler (tt);
                                     S:=StC
                                     end
```

```
                else if tt = B then ...
                else if tt = C then ...
                else if tt = α then ...
                else begin
                          ler (tt);
                          S:=sta
                     end;

          StC:;

        end (* case *)
      end (* ler-terminal *);
```

Analisemos agora o problema da gramática de alto nível.

Em primeiro lugar constatamos que não é uma gramática regular. No entanto se a transformarmos sucessivamente, quer por substituições, quer transformando a recursividade pela introdução do operador *, obtem-se a expressão regular:

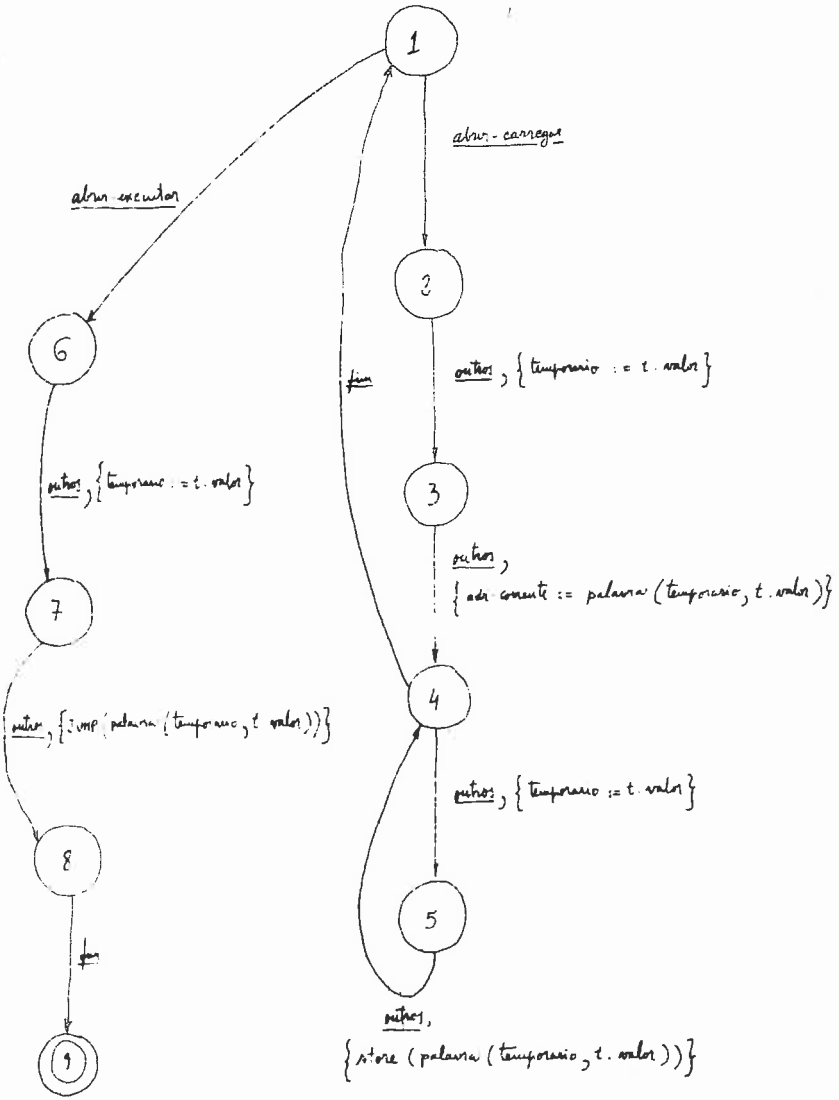
<transmissão> ≡
(abrir-carregar outros outros (outros outros)*
fim)* (abrir-executar outros outros fim)

Logo estamos perante uma linguagem regular para a qual é possível determinar um automato finito determinista que a reconhece.

No que se segue vamos admitir que não há erros.

Para explicarmos sintéticamente a semântica do problema vamos introduzir as seguintes definições:

```
type word = ... (*palavra de memória*)
     byte = 0 ..  256;
var temporário: byte;
    adr-corrente: word;
    t: terminal;
function palavra (b1, b2: byte): word;
    begin
        palavra: = b2 "concatenado com" b1
    end;
procedure store (P: word);
    begin
        memory (adr-corrente): = p;
        adr-corrente: = adr-corrente + 1
    end;
procedure JUMP (adr: word);
    (* lança a execução em adr *)
```


1
abrir executor
abrir - carregar
6
2
fim
outros, { temporario := t.valor }
outros, { temporario := t.valor }
3
7
outros,
{ add comente := palavra (temporario, t.valor)}
4
outros, { temporario := t.valor }
outros, {jump (palavra (temporario, t.valor))}
8
5
fim
outros,
{store (palavra (temporario, t.valor))}
9

Há laia de conclusão, podemos vêr como as gramáticas, expressões regulares e automatos, foram usados para analisar e obter um algoritmo, que é um modelo de uma solução para o problema colocado inicialmente.

Quais as questões a pôr em evidência:

a. As gramáticas ajudaram à clarificação da especificação do problema.

b. Ajudaram a conduzir a análise do mesmo.

c. Permitiram após manipulações adequadas das mesmas, obter algoritmos mais eficientes para implementar uma sua solução.

   Assim, a análise do algoritmo foi feita, de forma interposta, na transformação das gramáticas.

d. Os automatos poseram claramente em evidência os pressupostos que se tiveram que fazer, no que toca ao tratamento de erros.

NOTA COMPLEMENTAR:

A solução apresentada não é realista no que toca à utilização do procedimento ler-terminal. Um Tele-carregador destina-se geralmente a ser instalado numa máquina nua, sendo a sua função o carregamento do sistema de arranque (boodstrap loader). Todo o algoritmo desenhado é um modelo fácil de traduzir em código assembler, com excepção exactamente de ler-terminal, que exige uma possível "bufferização" da leitura de bytes, ou qualquer outro processo de bloquear o algoritmo até um byte chegar pela linha.

Uma outra solução seria fundir as 2 gramáticas e obter um único autómato. A iniciativa deveria então pertencer ao "handler" da linha de input, que, sempre que recebe um byte da linha, executa o case statement global do autómato.

Este por sua vez deixa de estar envolvido no while.

```
procedure  I/O trap (B: byte (*byte recebido*));
    begin
        case estado of
            1:
            2:
            3:
            .
            .
            .
            n: ...
                JUMP (palavra (temporário, B))
        end;
    end (* I/O trap *);
```

Todas as varáveis (estado, temporário, etc.) seriam globais.

6. Um editor interactivo - exemplo reduzido

Especificação reduzida.

Um editor interactivo, após o lançamento da execução, lê para memória central um ficheiro de texto, passado em input.

Os seguintes comandos permitem modificar esse texto (Inicialmente a linha corrente é a linha imediatamente anterior à primeira do ficheiro):

a. Inserir uma linha antes da linha corrente:
   Formato: I sequência de caracteres (carriage return)
   A linha corrente não se altera.

b. Suprimir a linha corrente:
   Formato: S (CR)
   A linha corrente é a que se segue.

c. Saltar relativamente linhas:
Formato: ± inteiro B (CR)
O sinal + pode ser omitido.

d. Mostrar a linha corrente:
Formato: M (CR)

e. Fim da edição:
Formato: F (CR)

Uma expressão regular que denota todas as sessões de trabalho sintácticamente correctas é a seguinte:

("I" "C" * "CR" + "S" "CR" + ("+" + "-" +λ) "d" "d" *
"B" "CR" + "M" "CR")* "F" "CR"

em que os terminais aparecem entre "'s.
"C" significa qualquer caracter
"CR" carriage return
"d" digito

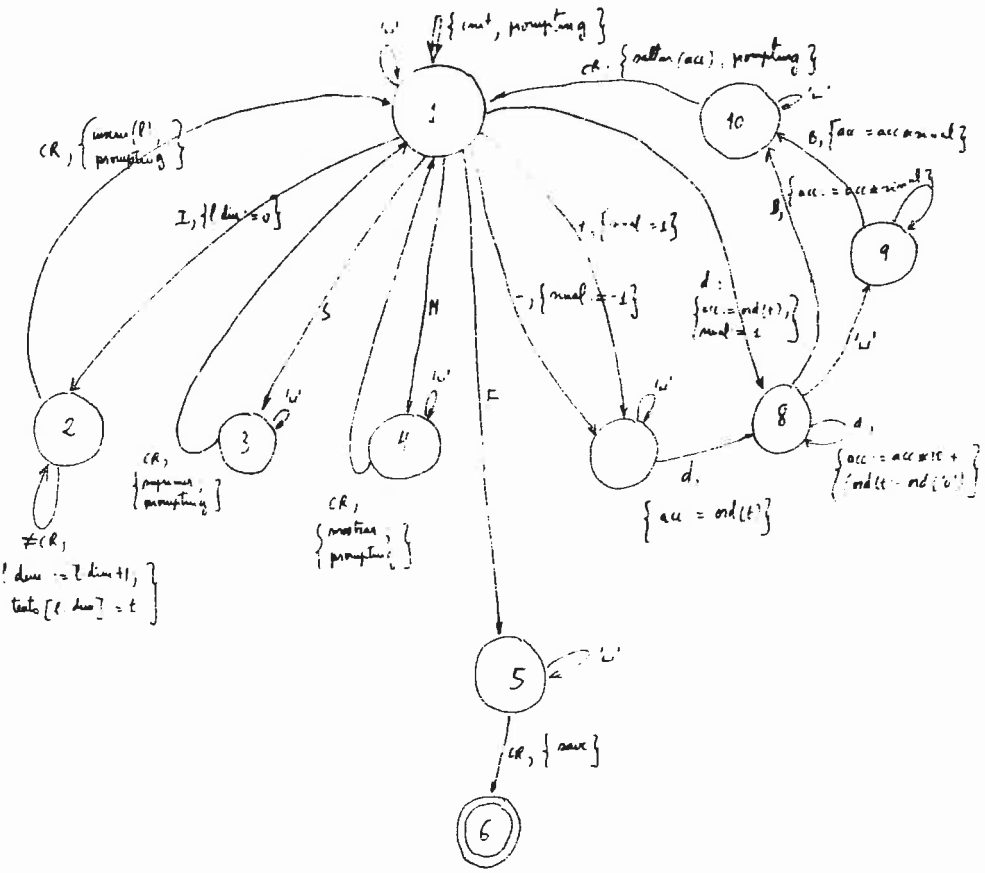
Para explicarmos a semântica e fixar rigorosamente o funcionamento do autómato, tomemos as definições:

```
const CR = chr (13); (*código interno de carriage return*)
conjunto-terminal = char
var t: char; (* caracter corrente lido *)
procedure ler-terminal (var t: char);
(*devolve o caracter lido da consola. Se se atingiu
 o fim da linha devolve t = CR*)
type linha = Record
               Texto: array [1...70] of char;
                 dim: ∅..70
             end
var l: linha;
var acc, sinal: integer; (*permitem calcular o inteiro*)
```

Existe também uma estrutura de dados abstracta manipulada pelas operações:

```
procedure init;
    (* lê do input o ficheiro a editar e inicializa
    a estrutura em memória.
    Linha corrente da estrutura toma o valor adequa
    do *);
procedure inserir (l: linha);
    (* insere l antes da linha corrente na estrutu-
    ra em memória.
    A linha corrente não é alterada *);
procedure mostrar;
    (* imprime na consola a linha corrente. Se a es
    trutura está vazia, ou a linha corrente ultrapas
    sa o fim, assinala o facto *);
procedure suprimir;
    (* suprime a linha corrente. A linha corrente
    passa à seguinte. Se não existe assinala esse fa
    cto *);
procedure saltar (n: integer);
    (* modifica a linha corrente em função de n.
    Se ultrapassa o início do ficheiro, ou o seu fim,
    assinala esse facto *);
procedure save;
    (* o conteúdo da estrutura em memória é escrito
    no ficheiro de output *);
```

Tomando em consideração as definições dadas, um primeiro autómato com funções semânticas seria o seguinte:



NOTA: o estado 9 e os lacetes com label '␣' destinam-se a possibilitar que espaços suplementares não desencadeiem imediatamente um erro, dando uma liberdade suplementar ao utilizador, o que quer dizer que a expressão regular que denota a linguagem do editor é ligeiramente diferente da apresentada inicialemente.

Para implementar o automato, por exemplo com case's, tinha-se ainda que definir:

```
type estado = 1 .. 10;
var S: estado;
```

A sua implementação seria trivial.

As acções semânticas repetidas poderiam ser transformadas em procedimentos.

O único aspecto delicado seria o do tratamento dos erros.

Uma forma simples de os tratar seria a introdução de um estado suplementar, 11, que seria o estado erro, assim como um procedimento assinalar-erro (N) que escreveria as mensagens de erro e que era invocado antes da afectação S: = 11;

Assim, em todos os ramos IF ... ELSE IF de cada ramo do case global, haveria sempre um ELSE

```
begin
    assinalar-erro (i);
    S: = 11
end
```

A técnica de tratamento de erros pode ser, por exemplo, desprezar toda a linha corrente até ao CR (caracter de resincronização) e passar ao estado 1.

Um aspecto complementar e interessante de focar é o seguinte:

E se o utilizador dá uma linha para inserir com mais de 70 caracteres ?

Existem 2 formas de resolver o problema:

a) Desprezá-los sem avisar o utilizador passando a acção semantica de memorização no estado 2 à forma:

```
if l. dim <= 69 then
  begin
       l. dim: = l. dim + 1;
       l. texto [ l. dim ] : = t
  end
```

ou então introduzir um estado 2 com a seguinte estrutura:

```
2:
  if t = CR then begin
                      inserir (l);
                      prompting;
                      ler-terminal (t)
                      S: =1
                 end

  else if l. dim <69 then
                 begin
                      l. dim: = l. dim + 1;
                      l. texto [ l. dim ] : = t;
                      ler-terminal (t);
                      S: = 2
                 end

  else
      begin
           assinalar-erro (...);
           S: = 11
      end;
```

Isto é, aproveitar a sintaxe para tratar os erros semanticos.