Processamento de Linguagens
Reconhecedores Sintáticos

José João Almeida
José Bernardo Barros

8 de Maio de 2020

Conteúdo

1 Introdução

Uma linguagem tem como objectivo a caraterização de um conjunto de objectos. É constituida por três componentes:

alfabeto é um conjunto finito de símbolos cujas combinações formam os elementos da linguagem.

sintaxe diz quais, de todas as combinações dos símbolos do alfabeto, pertencem à linguagem.

semântica descreve o significado dos elementos da linguagem.

Dado um alfabeto T, uma sequência $s = a_1a_2...a_n$, em que $\forall_i, (a_i \in T)$, diz-se uma string sobre T.

À string sem nenhuma componente chama-se *string nula* e denota-se por ε.

Uma operação fundamental entre strings é a *concatenação.*

Dadas $u = a_1a_2...a_n$ e $v = b_1b_2...b_m$, chama-se *u concatenado com v* à string

$$\text{u.v} = a_1a_2...a_nb_1b_2...b_m$$

Ao conjunto de todas as strings sobre um alfabeto T chama-se T^*

Sejam L_1 e L_2 subconjuntos de T^*. Define-se a concatenação de duas linguagens

$$\text{L} = \text{L}_1 \ . \ \text{L}_2 = \{\mu \in T^* | \mu = \mu_1.\mu_2 \wedge \mu_1 \in L_1 \wedge \mu_2 \in L_2\}$$

Ist é, é o conjunto das strings que se obtêm concatenando strings de L_1 com strings de L_2.

1.1 Expressões Regulares (ER)

As expressões regulares são uma forma de caraterizar a sintaxe de linguagens.

Uma expressão regular pode ser definida da seguinte maneira:

Sejam **a** um símbolo do alfabeto e **p** e **q** expressões regulares que caraterizam as linguagens L_p e L_q respectivamente. Então

1. ϕ carateriza a linguagem {} (linguagem vazia - a que não pertence qualquer string).
2. ε carateriza a linguagem $\{\varepsilon\}$ (linguagem cuja única string é a string vazia. É de notar a diferença com a anterior!).
3. **a** carateriza a linguagem {a}
4. **p + q** carateriza a linguagem $L_p \cup L_q$
5. **p q** carateriza a linguagem $L_p.L_q$
6. **(p)** carateriza a linguagem L_p
7. $\mathbf{p}^*$ carateriza a linguagem L_p^*
8. $\mathbf{p}^+$ carateriza a linguagem L_p^+

Exercício 1.1 *Escreva expressões regulares para especificar a sintaxe das seguintes linguagens:*

(a) strings constituidas por um ou mais ***a****'s*

(b) strings constituidas por um ou mais ***a****'s seguidas de zero ou mais* ***b****'s*

(c) strings que representam os números inteiros

(d) strings que representam os números reais (em Pascal, i.e., com parte inteira, parte decimal e expoente opcionais mas tem de existir parte inteira ou parte decimal)

(a) e = (a)$^+$

(b) e = (a)$^+$ (b)*

(c) e = ('+' + '−' + ε) ('0' + '1' + ...+ '8' + '9')$^+$

(d) e = ('+'+'−+ε).('0'+'1'+...+'9')*.(('0'+...+'9')'.'+'.'('0+...+'9')+ε).
(ε+'E'.('+'+'−+ε)..('0'+...+'9')$^+$)

Em casos em que a expressão regular é muito extensa, é costume escrevê-la de uma forma mais estruturada. Assim podíamos ter feito

e= sinal . mantissa . expoente
sinal = ('+' + '−' + ε)
mantissa = digitos . (digito . '.' + '.' . digito + ε) . digitos
expoente = ε+ ('E' . sinal . digito . digitos)
digitos = digito*
digito = ('0' + '1' + ... + '9')

Álgebra das expressões regulares
Sejam α, β e γ expressões regulares. Então é válido afirmar que:

1. $(\alpha + \beta) + \gamma = \alpha + (\beta + \gamma)$
2. $\alpha + \beta = \beta + \alpha$
3. $\alpha + \phi = \phi + \alpha = \alpha$
4. $\alpha + \alpha = \alpha$
5. $(\alpha \,.\, \beta) \,.\, \gamma = \alpha \,.\, (\beta \,.\, \gamma)$
6. $\alpha \,.\, \varepsilon = \varepsilon \,.\, \alpha = \alpha$
7. $\alpha \,.\, (\beta + \gamma) = \alpha \,.\, \beta + \alpha \,.\, \gamma$
8. $(\beta + \gamma) \,.\, \alpha = \beta \,.\, \alpha + \gamma \,.\, \alpha$
9. $\alpha^+ = \alpha \,.\, \alpha^* = \alpha^* \,.\, \alpha$
10. $\alpha^* = \varepsilon + \alpha^+$
11. $(\alpha + \varepsilon)^+ = (\alpha + \varepsilon)^* = \alpha^*$
12. **se** X = $\beta + \alpha$. X **então** X = α^* . β
13. **se** X = β + X . α **então** X = β . α^*

Exercício 1.2 *Considere que*

- *a execução da tarefa* ***A*** *seguida da execução da tarefa* ***B*** *se representa pela expressão regular* ***A.B***
- *a execução da tarefa* ***se c então A senão B*** *se representa pela expressão regular* ***c.A + ¬c.B***

- *a execução da tarefa **skip** se representa pela expressão regular ε.*
- *a execução da tarefa **abort** se representa pela expressão regular ϕ.*

(a) *Determine expressões regulares que representem a execução das seguintes tarefas:*

(i) ***enquanto c fazer A***

(ii) ***repetir A até c***

(b) *Usando as propriedades de expressões regulares (e o facto de $\neg\neg c = c$) mostre que:*

(i) *(**enquanto c fazer A**) = (**Se c então (repetir A até $\neg$ c) senão skip**)*

(ii) *(**repetir A até c**) = (**A; enquanto $\neg$c fazer A**)*

(c) *Considere o procedimento P definido como*

P = se c então A senão (B;P)

(i) *Determine uma expressão regular que caraterize a sua execução.*

(ii) *Usando as propriedades das expressões regulares, determine uma versão iterativa (usando **enquanto**) do procedimento **P**.*

(a)

(i) Analizemos as execuções da ação **enquanto c fazer A**:

A primeira ação a executar é o cálculo da condição **c**; se esta for falsa não se executa mais nada; senão executa-se a ação A e depois voltamos ao mesmo ponto. Isto é,

$$\begin{aligned}\textbf{enquanto c fazer A} &= \neg c.\varepsilon + c.A.(\neg c.\varepsilon + c.A.(...)) \\ &= \neg c + c.A.\neg c + c.A.c.A.\neg c + ... \\ &= (c.A)^*.\neg c\end{aligned}$$

(ii) Ao analizarmos a execução de **repetir A até c** vemos que:

A primeira ação a executar é A seguida do cálculo da condição **c**; se esta for verdadeira não se executa mais nada; senão voltamos ao mesmo ponto. Isto é,

$$\begin{aligned}\textbf{repetir A até c} &= A\ .\ (c.\varepsilon + \neg c.(A.(c.\varepsilon + \neg c.A.(...)) \\ &= A.c + A.\neg c.A.c + A.\neg c.A\neg c.A.c + ... \\ &= (A.\neg c)^*.A.c\end{aligned}$$

(b) Usando os reultados da alínea anterior teremos:

(i) **se c então (repetir A até ¬ c) senão skip** $= \mathrm{c}.(\ (\mathrm{A}.\neg\neg\mathrm{c})^* + \neg\mathrm{c}.\varepsilon$
$= \mathrm{c}.(\mathrm{A}.\mathrm{c})^*.\mathrm{A}.(\mathrm{c}.\mathrm{A})^*.\mathrm{c}.\mathrm{A}.\neg\mathrm{c} + \neg\mathrm{c}$
$= ((\mathrm{c}.\mathrm{A})^*.\mathrm{c}.\mathrm{A} + \varepsilon).\neg\mathrm{c}$
$= ((\mathrm{c}.\mathrm{A})^+ + \varepsilon).\neg\mathrm{c}$
$= (\mathrm{c}.\mathrm{A})^*.\neg\mathrm{c}$
$=$ **enquanto c fazer A**

(ii) **A; enquanto ¬c fazer A** $= \mathrm{A}.(\neg\mathrm{c}.\mathrm{A})^*.\neg\neg\mathrm{c}$
$= \mathrm{A}.(\neg\mathrm{c}.\mathrm{A})^*.\mathrm{c}$
$= (\mathrm{A}.\neg\mathrm{c})^*.\mathrm{A}.\mathrm{c}$
$=$ **repetir A até c**

(c)

(i) Considere-se que a expressão regular **p** traduz a execução do procedimento **P**. Então

$$\mathrm{p} = \mathrm{c}.\mathrm{A} + \neg\mathrm{c}.\mathrm{B}.\mathrm{p}$$

Esta equação recursiva tem a seguinte solução

$$\mathrm{p} = (\neg\mathrm{c}.\mathrm{B})^*.\mathrm{c}.\mathrm{A}$$

(ii) $\mathrm{p} = (\neg\mathrm{c}.\mathrm{B})^*.\mathrm{c}.\mathrm{A} =$
$= ((\neg\mathrm{c}.\mathrm{B})^*.\mathrm{c}).\mathrm{A}$
$=$ **enquanto ¬c fazer B);A**

1.2 Gramáticas

Uma das formas de caraterizar a sintaxe de uma linguagem é através do uso de **gramáticas**.

Uma gramática **G** é um objecto com quatro componentes:

$$\mathrm{G} = (\ \mathrm{T}\ ,\ \mathrm{N}\ ,\ \mathrm{S}\ ,\ \mathrm{P}\ )$$

em que:

T é o alfabeto da linguagem - conjunto finito de símbolos terminais.

N é o conjunto dos símbolos não terminais.

S é um símbolo não terminal especial chamado símbolo inicial.

P é um conjunto de produções.

Uma produção é um par $(\alpha\ ,\ \beta)$, e escrevemos $\alpha \to \beta$, em que α e β são sequências finitas de símbolos terminais e não terminais. Dada uma produção $\alpha \to \beta$ dizemos que α é o lado direito e que β é o lado esquerdo da produção.

Apresentam-se de seguida, algumas definições relacionadas com gramáticas:

- Sejam $\alpha \in (T \cup N)^+$ e $\beta \in (T \cup N)^*$

  Dizemos que α *deriva imediatamente em* β e notamos $\alpha \Longrightarrow \beta$ **sse**

$$\exists_{\Phi,\Psi\in(N\cup T)^*,(\delta_1\to\delta_2)\in P}\ .\ \alpha = \Phi\delta_1\Psi \wedge \beta = \Phi\delta_2\Psi$$

Isto é, existe uma produção em que

  - o lado esquerdo é uma sub-string de α
  - β pode ser obtida substituindo δ_1 por δ_2 em α.

- Dizemos que α *deriva num ou mais passos em* β e notamos $\alpha \Longrightarrow^+ \beta$ **sse**

$$\alpha \Longrightarrow \beta \quad \vee \quad \exists_{\delta \in (N \cup T)^*} \, . \, \alpha \Longrightarrow \delta \wedge \delta \Longrightarrow^+ \beta$$

  Ist é, se α deriva imediatamente em β ou numa outra string (δ) que deriva num ou mais passos em β.

- Dizemos que α *deriva em zero ou mais passos em* β e notamos $\alpha \Longrightarrow^* \beta$ **sse**

$$\alpha = \beta \quad \vee \quad \alpha \Longrightarrow^+ \beta$$

- Dada uma gramática G=(T,N,S,P), a linguagem gerada por essa gramática é:

$$L = \{\mu \in T^* \, . \, S \Longrightarrow^* \mu\}$$

  Isto é, é o conjunto das strings que se podem derivar do símbolo inicial S.

- Uma gramática diz-se *independente do contexto* **sse**

$$\forall_{\alpha \to \beta \, \in \, P} \, . \, \alpha \; \in \; N$$

  Isto é, em que os lados esquerdos das produções têm, apenas um símbolo e não terminal.

- Seja X $\in (T \cup N)$. Define-se *árvore de derivação* do símbolo X como uma árvore cujos nodos são marcados com símbolos terminais e não terminais, tais que

  - a raiz é marcada com o símbolo X;
  - as sub-árvores são ainda árvores de derivação associadas aos símbolos $Y_1, Y_2, ..., Y_n$, tais que existe uma produção X $\to Y_1 Y_2 ... Y_n$

  Note-se que associada a um símbolo terminal existe apenas uma árvore de derivação (apenas com raiz) enquanto que associadas a um símbolo não terminal podem existir várias árvores de derivação.

  Seja X $\in (T \cup N)$ para o qual existe a derivação X $\Longrightarrow \alpha_1 \Longrightarrow ... \Longrightarrow \alpha_n \Longrightarrow \mu$. Se construirmos a árvore de derivação associada a esse símbolo correspondente a essa derivação, a string resultante da concatenação das folhas da esquerda para a direita (fronteira da árvore) coincide com μ.

- Uma gramática diz-se *ambígua* **sse** a um mesmo símbolo existirem associadas mais do que uma árvore de derivação diferentes com a mesma fronteira.

Exercício 1.3 *Escreva gramáticas para especificar a sintaxe das seguintes linguagens:*

(a) strings constituidas por um ou mais ***a****'s*

(b) strings constituidas por um ou mais ***a****'s seguidas de zero ou mais* ***b****'s*

(c) strings que representam os números inteiros

(d) strings que representam os números reais (em Pascal, i.e., com parte inteira, parte decimal e expoente opcionais mas tem de existir parte inteira ou parte decimal)

(a) Produções da gramática
S → a
S → a S

ou escrevendo de outra forma equivalente
S → a | a S

(b) Produções da gramática:
S → A B
A → a | a A
B → ε | b B

(c) Produções da gramática:
S → A B
A → ε | '+' | '-'
B → D | D B
D → '0' | '1' | ...| '8' | '9'

(d) Produções da gramática:
S → Sinal Mantissa Expoente
Sinal → '+' | '−' | ε
Mantissa → Dígitos Meio Dígitos
Dígitos → ε| Dígito Dígitos
Meio → Dígito '.' | '.' Dígito
Dígito → '0' | '1' | '2' | ... | '9'
Expoente → ε| 'E' Sinal Dígito Dígitos

Uma das ideias chaves na escrita de gramáticas é identificar os blocos constituintes das strings a caraterizar e associar-lhes um símbolo não terminal.

A última alínea do exercício anterior é bem prova desta estratégia.

Exercício 1.4 *Escreva gramáticas para especificar a sintaxe das seguintes linguagens:*

*(a) strings constituidas por zero ou mais **a**'s seguidas de zero ou mais **b**'s sendo o número de **a**'s igual ao número de **b**'s.*

*(b) strings constituidas por zero ou mais **a**'s seguidas de zero ou mais **b**'s seguidas de zero ou mais **c**'s sendo o número de **a**'s mais o número de **b**'s igual ao número de **c**'s.*

*(c) strings constituidas por **a**'s e **b**'s sendo o número de **a**'s igual ao número de **b**'s.*

(a) Produções da gramática:
S → ε | a S b

(b) Produções da gramática:
S → A | a S c
A → ε | b A c

Considerou-se que o símbolo A representa strings com tantos **b**'s como **c**'s.

(c) Seja A um símbolo que representa strings que têm mais um **a** do que **b**'s, e B representa strings com um **b** a mais do que **a**'s.

Produções da gramática:
S → ε | a B | b A
A → a S | b A A
B → a B B | b S

Exercício 1.5 *Considere o conjunto L das strings sobre o alfabeto T = { −,M,I }, definido recursivamente por:*

(i) $\forall_{x \in (-)^*}$, $xM{-}Ix{-} \in L$

(ii) $\forall_{x,y,z \in (-)^*}$, **se** $xMyIz \in L$ ***então*** $xMy{-}Iz{-} \in L$

Escreva uma gramática que caraterize a sintaxe de L.

Na resolução deste exercício, convém constatar que a linguagem a definir é bastante semelhante à da alinea (b) do exercício anterior. De facto as strings que pretendemos caraterizar são compostas por três sequências de hifens (–) separadas pelos símbolos ≪M≫ e ≪I≫ em que o número de hifens da terceira sequência é igual ùsoma dos números de hifens das duas primeiras sequências. Assim sendo,
S → − S − | M X
X → − X − | − I −

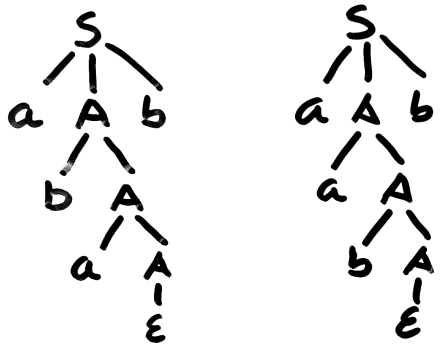
Exercício 1.6 *Considere uma gramática com as seguintes produções:*

- $S \rightarrow a\ A\ b$
- $A \rightarrow \varepsilon \mid a\ A \mid b\ A$

Construa árvores de derivação associadas ao símbolo S cujas fronteiras sejam:

a) $\alpha = abab$

b) $\beta = aabb$



2 Linguagens Regulares

De uma forma simplista, podemos afirmar que uma linguagem cuja sintaxe pode ser caraterizada por uma expressão regular é uma linguagem regular.

Outra das formas de caraterizar a sintaxe de linguagens regulares é através de gramáticas regulares ou autómatos.

Veremos nesta secção definições de gramáticas regulares e de autómatos, bem como processos de conversão entre as várias caraterizações de linguagens regulares.

Finalmente serão apresentados algoritmos de reconhecimento de linguagens regulares.

2.1 Gramáticas Regulares (GR)

Uma gramática independente do contexto G=(T,N,S,P) diz-se **regular**

à direita quando todas as produções são da forma:

$$A \rightarrow \mu$$

ou $A \rightarrow \mu\ B$

e em que A,B $\in$ N e $\mu \in T^*$

Isto é, todas as produções têm do lado direito, no máximo **um** símbolo não terminal (e este é o último símbolo do lado direito).

à esquerda quando todas as produções são da forma:

$$A \rightarrow \mu$$

ou $A \rightarrow B\ \mu$

e em que A,B $\in$ N e $\mu \in T^*$

Isto é, têm do lado direito, no máximo **um** símbolo não terminal (e este é o primeiro símbolo do lado direito).

Exercício 2.1 *Escreva gramáticas regulares à esquerda e à direita que caraterizem a sintaxe das seguintes linguagens:*

*(a) strings constituidas por um ou mais **a**'s seguidas de zero ou mais **b**'s*

(b) strings que representam os números inteiros

(a) Produções da gramática regular à direita:

- S → a A
- A → B | a A
- B → ε | b B

Produções da gramática regular à esquerda:

- S → A | S b
- A → a | A a

(b) Produções da gramática regular à direita:

- S → '+' N | '-' N | N
- N → '0' | ...| '8' | '9'
- N → '0' N | '1' N | ...| '8' N | '9' N

Produções da gramática regular à esquerda:

- S → A '0' | A '1' | ...| A '8' | A '9'
- S → S '0' | S '1' | ...| S '8' | S '9'
- A → '+' | '-' | ε

Exercício 2.2 *Considere a linguagem das strings que representam números com parte inteira e (opcionalmente) parte decimal (considere que* ***d*** *- dígito é um símbolo terminal).*

(a) Escreva uma gramática regular à direita para caraterizar a sintaxe dessa linguagem.

(b) Escreva uma gramática regular à esquerda para caraterizar a sintaxe dessa linguagem.

(a) S → I | E
I → A | '+' A | '-' A
A → d | d A
E → '+' F | '-' F | F
F → d F | d '.' A

(b) S → A d
A → ε | '+' | '-' | A d | F d '.'
F → ε | '+' | '-' | F d

2.2 Conversão de Gramáticas em Expressões Regulares

Na conversão de gramáticas em ER segue-se o seguinte método:

1. Para cada símbolo não terminal A cujas produções em que ele aparece no lado esquerdo são

$$A \rightarrow \alpha_1|\alpha_2|...|\alpha_n$$

escreve-se uma equação da forma

$$A = \Phi_1 + \Phi_2 + ... + \Phi_n$$

onde Φ_i é a expressão regular correspondente a α_i

Obtem-se assim um sistema de *equações*.

2. Resolve-se este sistema em ordem a S obtendo então a expressão regular pretendida

O sistema de equações obtido pode não ter solução o que significa que a linguagem caraterizada por tal gramática não pode ser expressa por uma expressão regular. Se a gramática for regular, o sistema tem sempre solução.

Exercício 2.3 *Considere a seguinte gramática:*

- *S* → *Sinal RealSemSinal*
- *Sinal* → $\varepsilon | + | -$
- *RealSemSinal* → *Inteiro ParteDecimal*
- *Inteiro* → *digito* | *digito Inteiro*
- *ParteDecimal* → $\varepsilon |$ *'.' Inteiro*

Determine uma expressão regular equivalente.

Equações :

- S = Sinal RealSemSinal
- Sinal = ε + '+' + '-'
- RealSemSinal = Inteiro ParteDecimal
- Inteiro = digito + digito Inteiro
- ParteDecimal = ε + '.' Inteiro

Solução: S = (ε + '+' + '-') (digito)$^+$ (ε + '.' (digito)$^+$)

2.2.1 Gramáticas Simplesmente Regulares (GSR)

Uma gramática G=(T,N,S,P) diz-se **simplesmente regular**

à direita quando todas as produções são da forma:

$$A \rightarrow a\ B$$

em que A,B ∈ N e **a** ∈ T, com excepção para a produção

$$Z \rightarrow \varepsilon$$

à esquerda quando todas as produções são da forma:

$$A \rightarrow B\ a$$

em que A,B ∈ N e a ∈ T*, com excepção para a produção

$$Z \rightarrow \varepsilon$$

Ao símbolo Z chama-se o **símbolo final**.

2.2.2 Conversão de GR em GSR

Seja $G_1 = (T_1,N_1,S_1,P_1)$ uma gramática regular à direita. Então é possível escrever uma gramática $G_2 = (T_2,N_2,S_2,P_2)$ simplesmente regular à direita tal que a linguagem gerada é a mesma.

Um processo standard para atingir este objectivo é o seguinte:

- $S_2 = S_1$
- $T_2 = T_1$
- $N_2 = N_1 \cup \{X_1, X_2, ..., X_n\} \cup Z$
- $Z \rightarrow \varepsilon \ \in P_2$
- **se** $A \rightarrow B \in P_1$ **então** $A \rightarrow B \in P_2$
- **se** $A \rightarrow a\ B \in P_1$ **então** $A \rightarrow a\ B \in P_2$
- **se** $A \rightarrow a \in P_1$ **então** $A \rightarrow a\ Z \in P_2$
- **se** $A \rightarrow a_1a_2...a_mB \in P_1$ **então**
  - $A \rightarrow a_1X_1 \in P_2$
  - $X_1 \rightarrow a_2\ X_2 \in P_2$
  - .....
  - $X_{m-2} \rightarrow a_{m-1}\ X_{m-1} \in P_2$
  - $X_{m-1} \rightarrow a_mB \in P_2$
- **se** $A \rightarrow a_1a_2...a_m \in P_1$ **então**
  - $A \rightarrow a_1X_1 \in P_2$
  - $X_1 \rightarrow a_2\ X_2 \in P_2$
  - .....
  - $X_{m-2} \rightarrow a_{m-1}\ X_{m-1} \in P_2$
  - $X_{m-1} \rightarrow a_mZ \in P_2$

É possível definir um processo semelhante para caraterizar a conversão de gramáticas regulares à esquerda em gramáticas simplesmente regulares à esquerda.

Exercício 2.4 *A partir das gramáticas obtidas no exercício anterior, obtenha gramáticas simplesmente regulares.*

(a)

S → I | E
I → A | '+' A | '-' A
A → d Z | d A
E → '+' F | '-' F | F
F → d F | d X_1
X_1 → '.' A
Z → ε

(b)

S → A d
A → Z | Z '+' | Z '-' | A d | X_1 '.'
X_1 → F d
F → Z | Z '+' | Z '-' | F d
Z → ε

2.3 Autómatos

Para além das gramáticas regulares e expressões regulares a sintaxe de uma linguagem regular pode ser caraterizada por autómatos.

Um autómato **A** é um objecto com 5 componentes:

$$A = (T , Q , S , Z , \delta)$$

onde

T é o alfabeto

Q é um conjunto de estados

S $\in$ Q é um estado especial chamado **estado inicial**

Z $\in$ Q é um conjunto de estados chamados **estados finais**

δ é uma função de transição de estados

Representamos graficamente um autómato da seguinte forma:

- a cada estado fazemos corresponder um círculo com o identificador de estado;
- os estados finais são assinalados fazendo-lhes corresponder um círculo diferente;
- sempre que para um dado estado **q**, existe um símbolo **a** do alfabeto tal que δ(q,a) = q', então marcamos uma seta do círculo correspondente ao estado **q** para o símbolo **q'** com a etiqueta **a**
- o estado inicial é marcado fazendo-lhe chegar uma seta sem origem.

Dado um autómato, γ é uma string da linguagem caraterizada por esse autómato **sse** existir um caminho do estado inicial para um estado final tal que a concatenação das etiquetas de todos os seus ramos é igual a γ

2.3.1 Autómatos Não Determinísticos (AND)

Um autómato não determinístico **A** é um autómato em que a função de transição de estado

δ: Q $\times$ (T $\cup\{\varepsilon\}$) $\rightarrow$ P(Q) $\cup\{\perp\}$

é uma função que dado um estado e um símbolo do alfabeto dá como resultado um conjunto de estados. Esta função é parcial, i.e., existem pares de valores (q,s) tais que δ(q,s) = $\perp$.

2.3.2 Conversão de GSR em AND

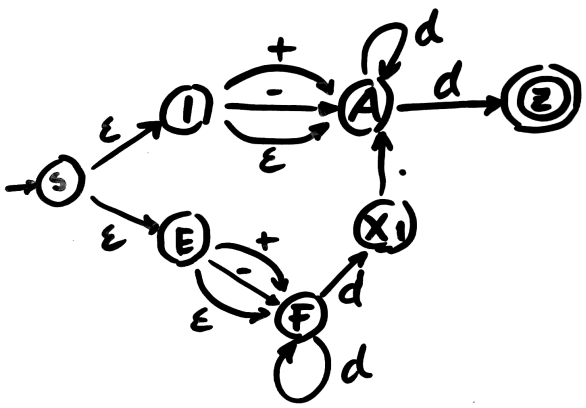
Dada uma gramática simplesmente regular à direita o autómato não deterministico correspondente pode ser obtido da seguinte forma:

- a cada símbolo não terminal corresponde um estado.
- o estado a que corresponde o símbolo S (símbolo inicial da gramática) é o estado inicial do autómato.
- o estado correspondente ao símbolo Z é o estado final do autómato.
- para cada produção da forma (A $\rightarrow$ B) $\in$ P coloca-se uma seta do estado correspondente a A para o estado correspondente a B com etiqueta ε.
- para cada produção da forma (A $\rightarrow$ a A) $\in$ P coloca-se uma seta do estado correspondente a A para o estado correspondente a B com etiqueta **a**.

Exercício 2.5 *Considere a seguinte gramática simplesmente regular à direita:*

- $S \rightarrow I \mid E$
- $I \rightarrow A \mid$ '+' $A \mid$ '-' A
- $A \rightarrow d\ Z \mid d\ A$
- $E \rightarrow$ '+' $F \mid$ '-' $F \mid F$
- $F \rightarrow d\ F \mid d\ X_1$
- $X_1 \rightarrow$ '.' A
- $Z \rightarrow \varepsilon$

Determine o AND correspondente a esta gramática.



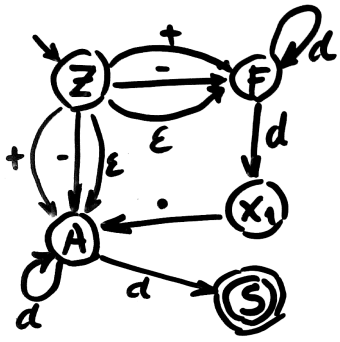
Dada uma gramática simplesmente regular à esquerda, o autómato não deterministico correspondente pode ser obtido da seguinte forma:

- a cada símbolo não terminal corresponde um estado.
- o estado a que corresponde o símbolo S (símbolo inicial da gramática) é o estado final do autómato.
- o estado correspondente ao símbolo Z é o estado inicial do autómato.
- para cada produção da forma (A $\rightarrow$ B) $\in$ P coloca-se uma seta do estado correspondente a B para o estado correspondente a A com etiqueta ε.
- para cada produção da forma (A $\rightarrow$ a A) $\in$ P coloca-se uma seta do estado correspondente a B para o estado correspondente a A com etiqueta **a**.

Exercício 2.6 *Considere a seguinte gramática simplesmente regular à esquerda:*

- $S \rightarrow A\ d$
- $A \rightarrow Z \mid Z$ '+' $\mid Z$ '-' $\mid A\ d \mid X_1$ '.'
- $X_1 \rightarrow F\ d$
- $F \rightarrow Z \mid Z$ '+' $\mid Z$ '-' $\mid F\ d$
- $Z \rightarrow \varepsilon$

Determine o AND correspondente a esta gramática.



Exercício 2.7 *Determine um processo sistemático de dada uma gramática simplesmente regular à direita obter uma gramática simplesmente regular à esquerda que caraterize a sintaxe da mesma linguagem.*

2.3.3 Autómatos Determinísticos (AD)

Um autómato determinístico **A** é um autómato em que a função de transição de estado

δ: Q $\times$ T $\rightarrow$ Q $\cup\{\perp\}$

é uma função que dado um estado e um símbolo do alfabeto dá como resultado um estado. Esta função é parcial, i.e., existem pares de valores (q,s) tais que δ(q,s) = $\perp$.

2.3.4 Conversão de AND em AD

Seja N = (T,Q,S,Z,δ) um autómato determinístico.

Considere-se a seguinte função:

fecho : P(Q) $\rightarrow$ P(Q)

fecho (X) = X $\cup \bigcup_{x \in X} fecho(\delta(x, \varepsilon))$

isto é, o fecho de um conjunto de estados X é o conjunto de estados a que se pode chegar dos estados de X através de transições por ε.

Para a partir de N = (T,Q,S,Z,δ) se obter o autómato D = (T',Q',S',Z',δ') correspondente procede-se da seguinte forma:

- T' = T
- S' = fecho ({S})
- δ' (X,t) = fecho $(\cup_{q \in X}, (\delta(q, t)))$
- Q' define-se recursivamente da seguinte forma:

  (i) fecho({S}) $\in$ Q'

  (ii) **se** q $\in$ Q' **então** δ' (q,t) $\in$ Q'

Exercício 2.8 *Considere a seguinte gramática:*

- $S \rightarrow A \mid a\ S \mid b\ A$
- $A \rightarrow a\ b \mid a\ b\ A \mid b\ a\ S$

a) Converta-a numa gramática simplesmente regular à direita.

b) Converta esta última num AND.

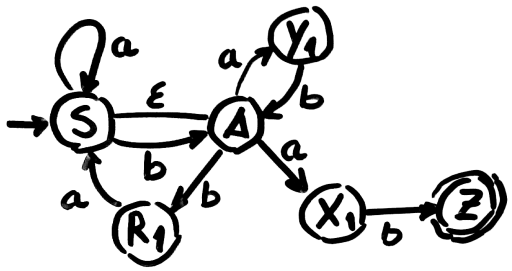
c) Converta o AND obtido na alínea b) num AD.

d) Converta o AND obtido na alínea b) numa gramática simplesmente regular à esquerda.

a)

S → A | a S | b A
A → a X_1| a Y_1| b R_1
X_1 → b Z
Y_1 → b A
R_1 → a S
Z → ε

b)



c) Um método fiável de conversão de AND em AD é o seguinte:

1. começa-se por construir uma tabela com uma primeira coluna para os estados e com uma coluna para cada símbolo terminal

Estado	a	b

2. A primeira linha corresponderá ao estado **S'**, isto é, ao $fecho(S)$ (no caso S, A).

   Na coluna correspondente ao símbolo **a** colocaremos todos os estados a que podemos chegar, partindo de S ou de A por ramos de peso **a**.

   Na coluna correspondente ao símbolo **b** colocaremos todos os estados a que podemos chegar, partindo de S ou de A por ramos de peso **b**.

Estado	a	b
S,A	S,A,X_1,Y_1	A, R_1

3. Para cada novo estado encontrado (conjunto de símbolos não terminais colocados na tabela) procederemos de igual forma.

Estado	a	b
S,A	S,A,X_1,Y_1	A, R_1
S,A,X_1,Y_1	S,A,X_1,Y_1	A,R_1,Z

4.

Estado	a	b
S,A	S,A,X_1,Y_1	A, R_1
S,A,X_1,Y_1	S,A,X_1,Y_1	A,R_1,Z
A,R_1	X_1,Y_1,S,A	R_1

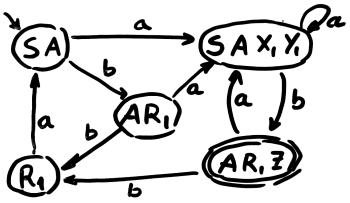
5.

Estado	a	b
S,A	S,A,X_1,Y_1	A, R_1
S,A,X_1,Y_1	S,A,X_1,Y_1	A,R_1,Z
A,R_1	X_1,Y_1,S,A	R_1
A,R_1,Z	X_1,Y_1,S,A	R_1

6.

Estado	a	b
S,A	S,A,X_1,Y_1	A, R_1
S,A,X_1,Y_1	S,A,X_1,Y_1	A,R_1,Z
A,R_1	X_1,Y_1,S,A	R_1
A,R_1,Z	X_1,Y_1,S,A	R_1
R_1	S,A	

O AD será então:



2.3.5 Conversão de ER em AND

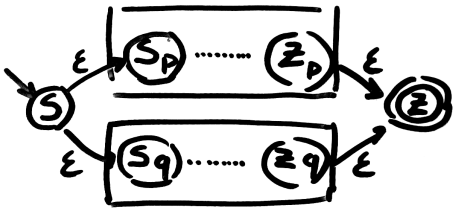
A conversão de ER em AND pode ser feita usando as seguintes regras:

1. $\mathbf{e} = \phi$

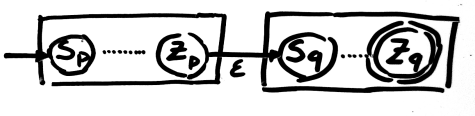
2. $\mathbf{e} = \varepsilon$

3. $\mathbf{e} = \mathbf{a}$

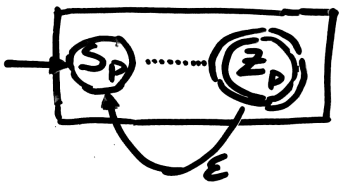
4. $\mathbf{e} = \mathbf{p} + \mathbf{q}$ sendo **p** e **q** expressões regulares



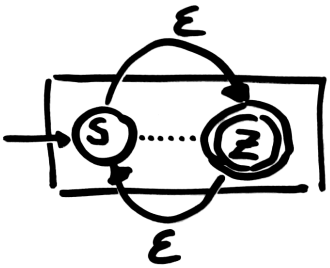
5. $\mathbf{e} = \mathbf{p} \cdot \mathbf{q}$ sendo **p** e **q** expressões regulares



6. $\mathbf{e} = \mathbf{p}^+$ sendo $\mathbf{p}$ uma expressão regular



7. $\mathbf{e} = \mathbf{p}^*$ sendo $\mathbf{p}$ uma expressão regular

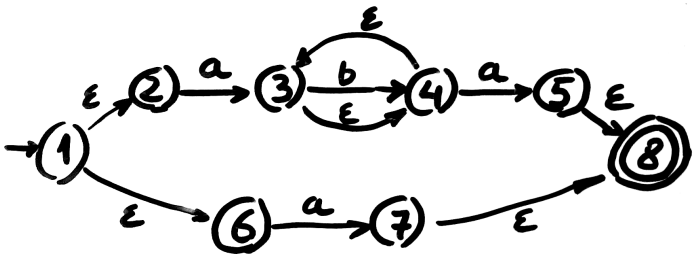


Exercício 2.9 *Considere a seguinte expressão regular:*

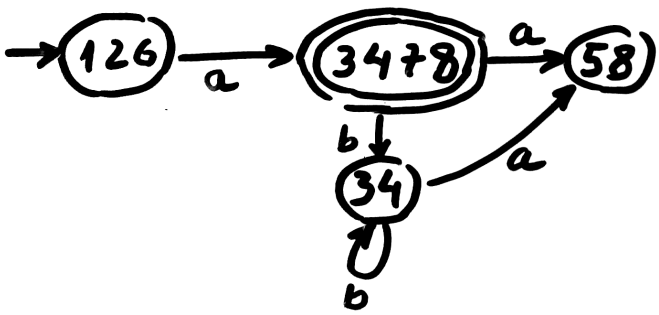
$$e = (a\ b^*\ a) + a$$

(a) Determine um AND correspondente à expressão regular.

(b) Converta-o para AD.



(a)



(b)

2.4 Reconhecedores baseados em AD

A estrutura geral de um reconhecedor baseado em autómatos determinísticos é a seguinte:

proc *reconhece* (**in** γ : *string*)

$$\left\{\begin{array}{l} estado \leftarrow estadoinicial; \\ \textbf{enquanto}\ \ (\neg((estado \in estadosFinais) \wedge (\gamma\uparrow = \$))) \longrightarrow \\ \quad \left\{\begin{array}{l} proximoEstado \leftarrow oraculo(estado, \gamma\uparrow); \\ \textbf{se}\ (proximoEstado \neq erro) \longrightarrow \\ \quad \left\{\begin{array}{l} executaAccao(estado, \gamma\uparrow); \\ estado \leftarrow proximoEstado; \\ avanca(\gamma) \end{array}\right. \end{array}\right. \end{array}\right.$$

A quarta e sexta linhas deste algoritmo devem ser refinadas. Como dado um estado e um símbolo, o próximo estado fica deterministicamente determinado, podemos refinar a instrução

$$estado \leftarrow oraculo(estado, \gamma\uparrow)$$

usando uma tabela onde será armazenado o autómato.

Teremos então uma tabela *oraculo* declarada da seguinte forma:

oraculo : array [estado, simbolo] of estado;

E em que a componente **[s,c]** significa o estado para que se transita se no estado **s** aparecer o símbolo **c**.

As ações semânticas nestes reconhecedores estão normalmente associadas às transições. Assim devemos ter uma outra tabela *accoes* que para cada par (estado,simbolo) contenha uma referência à ação associada a essa transição.

accoes : array [estado, simbolo] of accao;

Desta forma a sexta linha do algoritmo pode ser reescrita da seguinte forma:

caso accoes [estado,$\gamma\uparrow$] seja
1 : executa accao correspondente
2 : ...
...
n : ...

As duas tabelas referidas podem ser compactadas numa só em que cada componente é um *record* com 2 campos: o próximo estado e a ação a executar.

2.5 Análise de um Exemplo

Vamos tentar escrever um editor de texto muito simples que permita:

- abrir um ficheiro **A *ficheiro***
- posicionar numa determinada linha **P *linha***
- inserir texto terminado por ESC **i *texto* esc**
- apagar a linha atual **d**
- apagar as linhas desde uma posição a outra **d** $\boldsymbol{linha_1}$ $\boldsymbol{linha_2}$
- escrever o ficheiro **s**
- terminar a edição **f**

Todos os comandos (excepto o de inserção de texto) devem ser terminados com RETURN.

2.5.1 Caraterização da Linguagem

A gramática que se apresenta, representa as sequências de carateres válidas para o editor de texto.

S	→ Comando S | Fim
Comando	→ ComAbr | ComPos | ComIns | ComApa | ComEsc
ComAbr	→ A nome RETURN
ComPos	→ P inteiro RETURN
ComIns	→ i texto ESC
ComApa	→ d RETURN | d inteiro ',' inteiro RETURN
ComEsc	→ s RETURN
Fim	→ f RETURN
nome	→ alfabético | nome alfanumérico
inteiro	→ digito | digito inteiro
texto	→ ε| linha texto
linha	→ RETURN | char linha
numérico	→'0' | '1' | ... | '9'
alfanumérico	→ numérico | alfabético
alfabético	→ 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'
char	→ qualquer carater diferente de ESC e de RETURN

Esta gramática não põe em evidência o facto de que o primeiro comando a ser executado deve ser um comando de *abrir um ficheiro*. Para isso bastava modificar as produções associadas ao símbolo **S** para

S	→ ComAbr SeqCom Fim
SeqCom	→ ε| Comando SeqCom

Esta gramática, embora caraterizando a linguagem pretendida tem uma grande limitação - **não é regular**.

Para resolver este problema podemos seguir várias estratégias

- determinação de uma gramática regular equivalete
- determinação de uma expressão regular equivalente

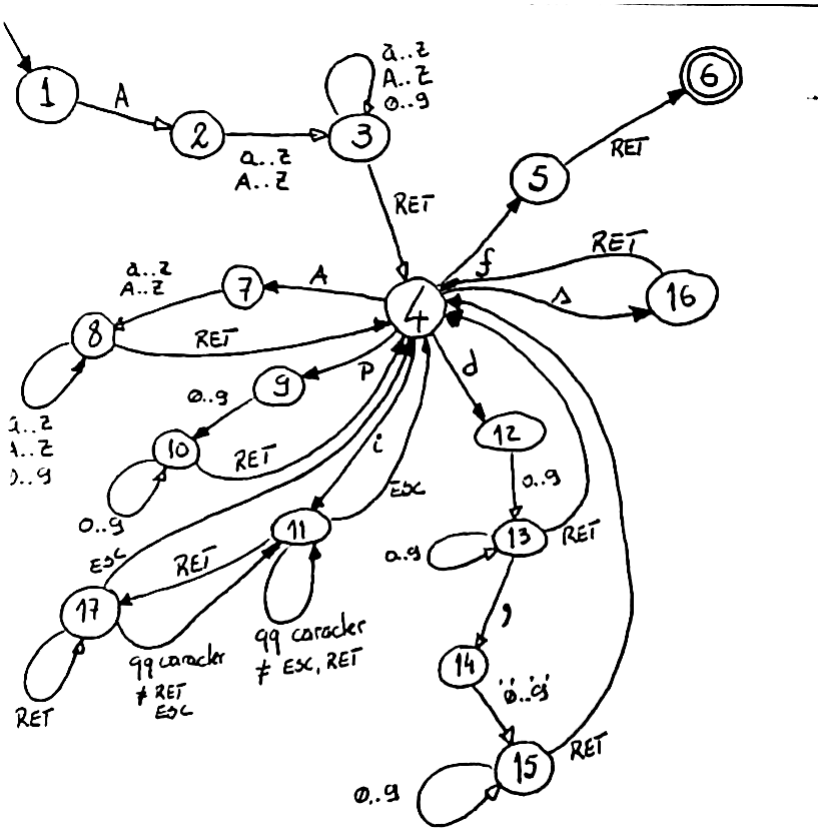
Note-se que o nosso objectivo é determinar um autómato que caraterize a linguagem.

Vamos optar pela escrita de uma expressão regular (escrita de uma forma mais ou menos estruturada)

S	= ComAbr . SeqCom . Fim
SeqCom	= (Comando)*
Comando	= ComAbr + ComPos + ComIns + ComApa + ComEsc
ComAbr	= A . nome . RETURN
ComPos	= P . inteiro . RETURN
ComIns	= i . texto . ESC
ComApa	= d . RETURN + d . inteiro . ',' . inteiro RETURN
ComEsc	= s . RETURN
Fim	= f . RETURN
nome	= alfabético . (alfanumérico)*
inteiro	= (digito)$^+$
texto	= (linha)*
linha	= (char)* . return
numérico	= '0' + '1' + ... + '9'

alfanumérico = numérico + alfabético
alfabético = 'a' + 'b' + ... + 'z' + 'A' + 'B' + ... + 'Z'
char qualquer carater diferente de ESC e de RETURN

O autómato equivalente a esta expressão é:



Podemos constatar o determínismo deste autómato.

Escrito sob a forma de uma tabela teremos

$Est \backslash Simb$	...	RET	...	ESC	...	A	...	P	...	Z	...	,	...	a	...
1						2									
2						3	3	3	3	3				3	3
3		4				3	3	3	3	3				3	3
4						7		9							
5		6													
6															
7						8	8	8	8	8				8	8
8		4				8	8	8	8	8				8	8
9															
10		4													
11	11	17	11	4	11	11	11	11	11	11	11	11	11	11	11
12															
13		4										14			
14															
15		4													
16		4													
17	11	17	11	4	11	11	11	11	11	11	11	11	11	11	11

$Est \backslash Simb$	...	d	...	f	...	i	...	s	...	z	...	0	...	9	...
1															
2	3	3	3	3	3	3	3	3	3	3					
3	3	3	3	3	3	3	3	3	3	3		3	3	3	
4		12		5		11		16							
5															
6															
7	8	8	8	8	8	8	8	8	8	8					
8	8	8	8	8	8	8	8	8	8	8		8	8	8	
9												10	10	10	
10												10	10	10	
11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
12												13	13	13	
13												13	13	13	
14												15	15	15	
15												15	15	15	
16															
17	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11

2.5.2 Ações Semânticas

Como dissemos as ações semânticas estão associadas às várias transições de estado.

Para as escrever vamos supor que temos disponível um módulo **ficheiro** que tem as seguintes operações:

carrega (*nome*: string) que carrega o ficheiro *nome* para memória.

guarda() que escreve o ficheiro que se encontra em memória principal em memória secundária.

posiciona (*n* : int) faz com que a linha corrente seja *n*.

apaga (*l*$_1$,*l*$_2$: int) que apaga as linhas compreendidas entre l_1 e l_2.

insere (*buffer* : seq (string)) que insere as linhas de *buffer* após a linha atual.

mostra () que escreve no ecran as linhas entre a (linha atual - 10) e a (linha atual + 10).

max () que dá como resultado o número de linhas do ficheiro atualmente em memória.

mensagem (*s*: string) escreve em local apropriado a mensagem *s*.

Vamos agora descrever as várias ações associadas às transições. Como existem ações comuns a várias transições, para cada ação indicam-se também as transições a que estão associadas

1. $\boxed{1} \to \boxed{2}$
   $\boxed{4} \to \boxed{7}$

   $\left\{ \; i \leftarrow 1 \right.$

2. $\boxed{2} \to \boxed{3}$
   $\boxed{3} \to \boxed{3}$
   $\boxed{7} \to \boxed{8}$
   $\boxed{8} \to \boxed{8}$

   $\left\{ \begin{array}{l} nome[i] \leftarrow simbolo; \\ i \leftarrow i + 1 \end{array} \right.$

3. $\boxed{3} \to \boxed{4}$
   $\boxed{8} \to \boxed{4}$

   $\left\{ \; carrega(nome) \right.$

4. $\boxed{4} \to \boxed{9}$
   $\boxed{4} \to \boxed{12}$

   $\left\{ \; numero \leftarrow 0 \right.$

5. $\boxed{9} \to \boxed{10}$
   $\boxed{10} \to \boxed{10}$
   $\boxed{12} \to \boxed{13}$
   $\boxed{13} \to \boxed{13}$
   $\boxed{14} \to \boxed{15}$
   $\boxed{15} \to \boxed{15}$

   $\left\{ \; numero \leftarrow (numero * 10) + (ord(simbolo) - ord('0')) \right.$

6. $\boxed{10} \to \boxed{4}$

   $\left\{ \begin{array}{ll} \textbf{se} & (numero > max()) \longrightarrow \left\{ \begin{array}{l} numero \leftarrow max() \end{array} \right. \\ & (numero = 0) \longrightarrow \left\{ \begin{array}{l} numero \leftarrow 1 \end{array} \right. \\ posiciona(numero) & \end{array} \right.$

7. $\boxed{4} \to \boxed{11}$

   $\left\{ \begin{array}{l} linhas \leftarrow <>; \\ i \leftarrow 1 \end{array} \right.$

8. $\boxed{11} \rightarrow \boxed{11}$
   $\boxed{17} \rightarrow \boxed{11}$

$$\left\{ \begin{array}{l} linha[i] \leftarrow simbolo; \\ i \leftarrow i + 1 \end{array} \right.$$

9. $\boxed{11} \rightarrow \boxed{17}$
   $\boxed{17} \rightarrow \boxed{17}$

$$\left\{ \begin{array}{l} linha[i] \leftarrow simbolo; \\ linhas \leftarrow linhas \frown < linha >; \\ i \leftarrow 1 \end{array} \right.$$

10. $\boxed{11} \rightarrow \boxed{\ 4\ }$
    $\boxed{17} \rightarrow \boxed{\ 4\ }$

$$\left\{ \ insere(linhas)$$

11. $\boxed{13} \rightarrow \boxed{14}$

$$\left\{ \begin{array}{l} \textbf{se} \quad (numero > max()) \longrightarrow \left\{ \begin{array}{l} numero \leftarrow max() \\ numero \leftarrow 1 \end{array} \right. \\ \qquad\ \ numero = 0) \longrightarrow \\ linha_1 \leftarrow numero; \\ numero \leftarrow 0 \end{array} \right.$$

12. $\boxed{13} \rightarrow \boxed{\ 4\ }$

$$\left\{ \begin{array}{l} \textbf{se} \quad (numero > max()) \longrightarrow \left\{ \begin{array}{l} numero \leftarrow max() \\ numero \leftarrow 1 \end{array} \right. \\ \qquad\ \ numero = 0) \longrightarrow \\ apaga(numero, numero) \end{array} \right.$$

13. $\boxed{15} \rightarrow \boxed{\ 4\ }$

$$\left\{ \begin{array}{l} \textbf{se} \quad (numero > max()) \longrightarrow \left\{ \begin{array}{l} numero \leftarrow max() \\ numero \leftarrow 1 \end{array} \right. \\ \qquad\ \ numero = 0) \longrightarrow \\ apaga(linha_1, numero) \end{array} \right.$$

14. $\boxed{\ 4\ } \rightarrow \boxed{16}$

$$\left\{ \ mensagem(\text{RETURN para guardar})$$

15. $\boxed{16} \rightarrow \boxed{\ 4\ }$

$$\left\{ \ guarda()$$

16. $\boxed{\ 4\ } \rightarrow \boxed{\ 5\ }$

$$\left\{ \ mensagem(\text{RETURN para sair})$$

17. $\boxed{5} \rightarrow \boxed{6}$

$\left\{ \; skip \right.$

3 Métodos de Reconhecimento - introdução

No capítulo anterior analisou-se a escrita de reconhecedores com base em autómatos determinísticos.

Esses reconhecedores cobrem apenas as situações de linguagens regulares e são de delicada utilização para gramáticas/linguagens regulares complexas já que a associação de acções semânticas é feita com base nas transições de estado do autómato.

Isso corresponde a um nível por vezes demasiado "baixo", também se tornando difícil o desenho de autómatos com um número elevado de estados.

Nesta secção pretende-se introduzir outros métodos de reconhecimento sintático mais directamante baseados em gramáticas, incluindo um leque mais vasto de linguagens e com associação de acções semânticas directamente às produções.

Estes métodos têm associadas teorias que são aqui expostas de um modo superficial e muito incompleto.

Numa secção final será analisado um exemplo concreto. O exemplo escolhido foi o de uma **calculadora** de expressões.

Por último será feita uma breve alusão a ferramentas de software para geração automática de analisadores sintáticos

De um modo genérico, os reconhecedores podem ser classificados como:

- TOP-DOWN - (cuja estratégia consiste em tentar reconhecer o axioma da gramática como objectivo inicial e conforme os símbolos terminais que surgirem na string de entrada, escolheremos um caminho, i.e. decidiremos qual a nova sequência de ojectivos a reconhecer). Nesta situação diremos que a árvore de derivação é construída da raiz (o axioma) para as folhas (os símbolos terminais da linguagem).
- BOTTOM-UP - (neste caso examinaremos os terminais na tentativa de os agrupar em símbolos não terminais e assim sucessivamente na tentativa de conseguirmos agrupar tudo no símbolo axioma). Nesta situação diremos que a árvore de derivação é construída das folhas para a raiz.

Dentro cada uma destas classes haverá várias variantes conforme seja ou não dirigido por tabela, conforme seja ou não um algoritmo recursivo.

Para a descrição que se segue, utilizaremos a seguinte gramática, correspondente a expressões numéricas LISP muito simplificadas:

```
0  S → Exp '.'

1  Exp → int
2      |  '(' Funcao ')'

3  Funcao → '+' Lista
4         |  '*' Lista

5  Lista → Exp Lista
6        |                        /* Vazio */
```

Incluindo portanto expressões como:

```
( * 2 3 ( + 1 1 ) )                    (daria 12)
( + 1 2 ( * 1 2 3 ) ( + 11 11 ) )      (daria 31)
```

4 Reconhecedores Top-Down

4.1 Reconhecedor recursivo descendente

Este reconhecedor baseia-se na escrita de um procedimento para cada símbolo (terminal ou não) da gramática.

No caso dos símbolos terminais, o seu reconhecimento corresponde a verificar se o símbolo seguinte da string a reconhecer é igual ao símbolo terminal que se pretende reconhecer. Para permitir mais fácil descrição das acções semânticas, cada símbolo poderá ter a ele associado um valor que será retornado:

função $recT\ (\ t : T) : tipoValT$

$$\begin{cases} \textbf{se}\quad t = simboloseg \longrightarrow \begin{cases} val \leftarrow simboloseg.valor \\ ABORT \end{cases} \\ \qquad \textbf{senão} \longrightarrow \\ \text{return } val \end{cases}$$

Para cada símbolo não termininal A com produções da forma

$$A \rightarrow x1\ x2\ ...\ xn|...|y1\ y2\ ..\ ym$$

haverá que decidir qual produção deverá ser seguida.

Para a escolha da produção a ser usada aquando do reconhecimento de um não terminal, torna-se necessário o cálculo dos conjuntos de terminais que correspondem a inícios de caminhos possíveis de derivação para cada produção.

Esses conjuntos chamam-se **lookahead** de uma produção e serão mais exactamente definidos na secção seguinte, bem como o algoritmo de cálculo respectivo[1].

A escolha será feita com base nos **lookahead**: só se poderá seguir por uma determinada produção se o símbolo seguinte da string a reconhecer pertencer ao lookahead dessa produção.

função $recA\ (\ ) : tipoValA$

$$\begin{cases} \textbf{se}\ simboloseg \in lookahead(A \rightarrow x1x2...xn) \longrightarrow \\ \quad \begin{cases} val1 \leftarrow recx1; \\ val2 \leftarrow recx2; \\ ... \\ valn \leftarrow recxn; \\ valA \leftarrow f(val1, val2, ...valn) \end{cases} \\ \quad simboloseg \in lookahead(A \rightarrow ...) \longrightarrow \\ \quad \begin{cases} ... \end{cases} \\ \quad simboloseg \in lookahead(A \rightarrow y1y2...ym) \longrightarrow \\ \quad \begin{cases} val1 \leftarrow recy1; \\ val2 \leftarrow recy2; \\ ... \\ valm \leftarrow recym; \\ valA \leftarrow g(val1, val2, ...valm) \end{cases} \\ \quad \textbf{senão} \longrightarrow \\ \quad \begin{cases} [erro] \end{cases} \\ \text{return } valA \end{cases}$$

As variáveis "valX"vão armazenar os valores associados a cada símbolo do lado direito da produção seguida e são usados para facilitar a descrição da semântica associada a cada produção.

[1] Na secção seguinte é feita a analise do cálculo dos lokahead1 embora o problema possa ser facilmente generalizavel para lookahead$_k$

Saliente-se que a única parte que vai diferir de caso para caso, é:

- a escrita da gramática,
- a escolha dos tipos a associar a cada símbolo e
- a escrita das expressões f() e g() para cálculo dos valores a associar aos NT.

Será portanto de prever a possibilidade de existência de ferramentas que, a partir da gramática + Tipos de valores de cada símbolo + expressões de cálculo dos referidos valores, gerem o reconhecedor (ver YACC, LALR, ELL, ...).

Para além das funções $g(), f()$ que implementam o cálculo do valor a associar ao símbolo não terminal (acções de síntese) há necessidade de executar outras (acções de reacção) para executar algo com esses valores (ex. escrevê-los no ecran).

proc *rec_S* ( )

$$\begin{cases} val1 \leftarrow recExp() \\ recT('.') \\ escreve(val1) \end{cases}$$

função *rec_Exp* ( ) : *real*

$$\begin{cases} \textbf{se} \quad simbseg = int \longrightarrow \begin{cases} val \leftarrow recT(int) \end{cases} \\ \qquad\quad simbseg =' (' \longrightarrow \begin{cases} recT('(') \\ val2 \leftarrow rec\_Funcao() \\ recT(')') \\ val \leftarrow val2 \end{cases} \\ \text{return } val \end{cases}$$

função *rec_funcao* ( ) : *real*

$$\begin{cases} \textbf{se} \quad simbseg =' +' \longrightarrow \begin{cases} recT('+') \\ val2 \leftarrow rec\_Lista() \\ val \leftarrow soma(val2) \end{cases} \\ \qquad\quad simbseg =' *' \longrightarrow \begin{cases} recT('*') \\ val2 \leftarrow rec\_Lista() \\ val \leftarrow mult(val2) \end{cases} \\ \text{return } val \end{cases}$$

função *rec_Lista* ( ) : $real - seq$

$$\begin{cases} \textbf{se} \quad simbseg =')' \longrightarrow \begin{cases} val \leftarrow <> \end{cases} \\ \qquad\quad simbseg \in \{'(', int\} \longrightarrow \begin{cases} val1 \leftarrow rec\_Exp() \\ val2 \leftarrow rec\_Lista() \\ val \leftarrow cons(val1, val2) \end{cases} \\ \text{return } val \end{cases}$$

4.2 Lookahead1, first1 follow1

Para o cálculo dos lookahead é necessário a definição de duas outras funções auxiliares: **First e Follow**.

4.2.1 First

O cálculo de First1 ou início corresponde a determinar o conjunto de terminais que são inícios válidos da linguagem associada a um símbolo não terminal (**first_nt**), ou a uma sequência de símbolos terminais ou não (**first_str**).

função *first_nt* ($G : gramatica, A : NT) : T - set$

$$\left\{ \begin{array}{l} \textbf{para } (A \rightarrow rhs) \in Producoes(G) \textbf{ fazer} \\ \quad \left\{ \; fir \leftarrow fir \cup first\_str(rhs) \right. \\ \text{return } fir \end{array} \right.$$

função *first_str* ($G : gramatica : s : (T|NT) - seq) : T - set$

$$\left\{ \begin{array}{ll} \textbf{se} \quad s = <> \longrightarrow & \left\{ \; fir \leftarrow \{\} \right. \\ \quad s = <h.t> \wedge h \in T \longrightarrow & \left\{ \; fir \leftarrow h \right. \\ \quad s = <h.t> \wedge h \in NTanulaveis \longrightarrow & \left\{ \; fir \leftarrow first\_nt(h) \cup first\_str(t) \right. \\ \quad s = <h.t> \wedge h \notin NTanulaveis \longrightarrow & \left\{ \; fir \leftarrow first\_nt(h) \right. \\ \text{return } fir & \end{array} \right.$$

[2]

Exercício 4.1 *Constate que:*

- *NTanulaveis* = { *Lista* }
- *first(S) = first(Exp) = first(Lista)* = { *int, '(' }*
- *first(Funcao)* = { *'*', '+'* }

4.2.2 Follow

Follow1 ou seguinte de um símbolo não terminal **A**, corresponde a determinar o conjunto de símbolos terminais que se podem seguir àquilo em que **A** derive. O cálculo do Follow(A) vai depender do contexto em que **A** possa estar inserido. Esses contextos vão ser lidos nas produções em que **A** aparece no lado direito. Nomeadamente se existe uma produção

$$B \rightarrow ... \; A \; cont$$

então o Follow1(A) vai incluir o início de **cont** ou seja $first\_str(cont)$; Quando **cont** for anulável, então aquilo que se puder seguir a **B** também pertencerá a Follow1(A).

função *follow* ($G : gramatica, A : NT) : T - set$

$$\left\{ \begin{array}{l} \textbf{para } (B \rightarrow \; ... \; A \; cont) \in Producoes(G) \textbf{ fazer} \\ \quad \left\{ \begin{array}{l} \textbf{se } [\text{cont é anulável}] \longrightarrow \\ \quad \left\{ \; fol \leftarrow fol \cup first\_str(G, cont) \cup follow(G, B) \right. \\ \quad \textbf{senão} \longrightarrow \\ \quad \left\{ \; fol \leftarrow fol \cup first\_str(G, cont) \right. \end{array} \right. \\ \text{return } fol \end{array} \right.$$

Exercício 4.2 *Constate que:*

[2] $s =< h.t >$ ... deve ser interpretado como:
$h \leftarrow head(s)$
$t \leftarrow tail(s)$

- *follow(S) = {}*
- *follow(Exp) = { int '(' ')' '.' }*
- *follow(Funcao)= { ')' }*
- *follow(Lista)= { ')' }*

4.2.3 Lookahead

Para determinar o **lookahead** de uma produção usaremos a seguinte definição:

função *lookahead* ($G : gramatica, P : producao)) : T - set$

$$\left\{ \begin{array}{l} \textbf{se } [\text{rhs(P) é anulável}] \longrightarrow \\ \quad \left\{ \; loo \leftarrow first\_str(G, rhs(P)) \cup follow(G, lhs(P)) \right. \\ \quad \textbf{senão} \longrightarrow \\ \quad \left\{ \; loo \leftarrow first\_str(G, rhs(P)) \right. \\ \text{return } loo \end{array} \right.$$

ou seja, o conjunto de símbolos terminais que se podem seguir a uma situação em que possa ser usada uma produção $A \rightarrow rhs$, é determinada pelo $first\_str(rhs)$ sendo ainda necessário analisar o que se segue a **A** se **rhs** for anulável.

Exercício 4.3 *Constate que:*

- *lookahead(S → Exp '.') = {int,(}*
- *lookahead(Exp → int) = {int }*
- *lookahead(Exp → '(' Funcao ')') = {(}*
- *lookahead(Funcao → '+' Lista) = {+ }*
- *lookahead(Funcao → '*' Lista) = { * }*
- *lookahead(Lista → Exp Lista) = {int,(}*
- *lookahead(Lista →) = {) }*

4.3 Top-Down dirigido por tabela

4.3.1 Funcionamento Geral

Este algoritmo corresponde a uma versão não recursiva do método recursivo descendente, anteriormente analisado.

O funcionamento geral do reconhecedor vai ser descrito com base numa tabela em que descreveremos a evolução do algoritmo através da indicação de:

- lista de objectivos a reconhecer; cada objectivo será um símbolo pertencente aos terminais ou não terminais da gramática,
- string de entrada a ser reconhecida
- árvore de derivação até ao momento
- acção a realizar:
  - "prod n- Ao pretender reconhecer um símbolo não terminal "A"e com base no(s) símbolo(s) terminal seguinte(s) **x**, conclui-se que deve ser usada a produção "n". O objectivo actual vai ser substituído pelo lado direito da produção.

– "avança- acção correspondente a ter como objectivo o reconhecimento de um terminal igual ao proximo símbolo da string a reconhecer.
– "reconhecimento- acção correspondente a ter tido sucesso no reconhecimento
– "erro- acção correspondente a impossibilidade de reconhecimento.

Objectivo	string	acção	arvore de derivação
Exp .	(+ 3 4) .	Prod 2	Exp
(Fun) .	(+ 3 4) .	Avanca	(Fun)
Fun) .	+ 3 4) .	Prod 3	
+ Lis) .	+ 3 4) .	Avanca	+ Lis
Lis) .	3 4) .	Prod 5	
Exp Lis) .	3 4) .	Prod 1	Exp Lis
int Lis) .	3 4) .	Avanca	int
Lis) .	4) .	Prod 5	
Exp Lis) .	4) .	Prod 1	Exp Lis
int Lis) .	4) .	Avanca	int
Lis) .	) .	Prod 6	
) .	) .	Avanca	vazio
.	.	Reconhe	

4.3.2 Algoritmo Top-down dirigido por tabela

Da tabela anterior haverá que extrair os objectos a empregar no algoritmo.

Objectivo: de uma análise da tabela anterior concluiremos que é um objecto de características sequenciais e que alterado por:

- Avança : retirar o primeiro elemento
- Prod n : retirar o primeiro elemento e introduzir uma sequência de outros elementos (o lado direito da produção n) no local dele.

Este objecto poderá ser implementado como:

- tipo: **stack(T|NT)**.
- funções:
  - PushO: T|NT $\rightarrow$
  - PopO: $\rightarrow$
  - TopoO: $\rightarrow$ T|NT
  - IniciaO: $\rightarrow$

Para as acções Pord n torna-se necessária a existência de um objecto *produções* que guarde as produções da gramática.

$$prod = array[1..maxprod]de\ producao$$

$$producao = \begin{array}{l} esq : NT \\ dir : (T|NT) - seq \end{array}$$

Outro problema consiste em decidir que acção deve ser tomada em cada situação. Para tal, começaremos por construir uma tabela que armazenará qual acção a efectuar quando se pretenda reconhecer um determinado símbolo (T ou NT) (eixo horizontal), em função do símbolo seguinte da entrada (eixo vertical).

As acções poderão ser:

- Av – Avançar na string de entrada
- Pn – seguir pela produção **n**
- Rec – Reconhecer
- (vazia) – Erro

	int	(	)	+	*	.		Exp	Fun	Lis
int	Av							P1		P5
(		Av						P1		P5
)			Av							P6
+				Av					P3	
*					Av				P4	
.						Rec				

Construção da tabela:

Esta tabela (normalmente chamada **Oráculo**) é composta de duas partes. A primeira corresponde ao reconhecimento de símbolos terminais e é facilmente preenchida através de colocar a acção **Av** na diagonal principal e Erro em todos os outros pontos. (Na prática esta primeira parte não precisa sequer de ser armazenada; Porquê?)

$$\left\{ \begin{array}{l} \textbf{para } t1, t2 \in T^2 \textbf{ fazer} \\ \quad \left\{ \begin{array}{lll} \textbf{se } & t1 = t2 \neq \text{"."} \longrightarrow & \left\{ oraculo[t1, t2] \leftarrow AV \right. \\ & t1 = t2 = \text{"."} \longrightarrow & \left\{ oraculo[t1, t2] \leftarrow Rec \right. \\ & t1 \neq t2 \longrightarrow & \left\{ oraculo[t1, t2] \leftarrow Erro \right. \end{array} \right. \end{array} \right.$$

A segunda parte da tabela corresponde às acções a efectuar quando se pretende reconhecer símbolos não terminais. Essas acções serão no sentido de decidir o caminho a seguir (ou seja escolher uma produção). Essa escolha será feita em função dos lookahead das produções:

$$\left\{ \begin{array}{l} \text{[inicializar esta parte da matriz a "erro"]} \\ \textbf{para } (A \rightarrow rhs) \in Producoes \textbf{ fazer} \\ \quad \left\{ \begin{array}{l} \textbf{para } t \in Lookahead(A \rightarrow rhs) \textbf{ fazer} \\ \quad \left\{ oraculo[t, A] \leftarrow P_{(A \rightarrow rhs)} \right. \end{array} \right. \end{array} \right.$$

Vamos ainda supor que existem as seguintes funções sobre a string a reconhecer **s**:

- avança(s) – avança para o símbolo seguinte na string
- act(s) – dá o símbolo actual

O algoritmo do reconhecedor Top-Down dirigido por tabela vai ser um ciclo que executará acção seguinte até que a acção seja "Reconhecimento"ou "Erro".

```
proc Reconhecedor ( in s : stream(T)[* string a reconhecer *]
                    in orac : oraculo
                    in prod : [array de produção]
                    out suc : bool[* sucesso no reconhecimento *])
{ IniciaO()
  PushO([fim de string])
  PushO(Axioma)
  ac ← orac[act(s), TopoO()]
  enquanto ac.t ∉ {Rec, Erro} ⟶
      { se ac.t = avanca ⟶ { PopO()
                             avanca(s)
                           ac.t = producao ⟶ { PopO()
                                               p ← Prod[ac.numprod]
                                               para simb ∈ inverte(p.direito) fazer
                                                   { PushO(simb)
        ac ← orac[act(g), topoO()]
  suc ← ac.t = Rec
```

Neste algoritmo não foi ainda considerada a questão da semântica. Uma maneira habitual de abordar esta questão é acrescentar mais um tipo de acção (acção **semântica**) e mais um tipo de objectivo:

- objectivo – tipo: **stack(T|NT|semantico)**.

Para além disto acrescentaremos mais um campo **sem** na definição de produção. Esse campo vai corresponder ao nº da acção semântica a executar no final de reconhecer essa expressão:

- producao =
  $esq : NT$
  $dir : (T|NT) - seq$
  $sem : Int$

Deverá também existir um procedimento **execsemantica** que execute a acção cujo código lhe for passado.

Por último para permitir que as acções semânticas possam usar valores associados aos símbolos do lado direito das produções e possam armazenar Valores associados ao lado esquerdo, utilizaremos uma **stack(valores)** com funções:

- PushV: Val $\rightarrow$
- PopV: $\rightarrow$ Val
- IniciaV: $\rightarrow$

```
proc Reconhecedor ( in s : stream(T)[* string a reconhecer *]
                    in orac : oraculo
                    in prod : [array de produção]
                    in execsemantica : Procedimento
                    out suc : bool[* sucesso no reconhecimento *])
{ IniciaO()
  IniciaV()
  PushO([fim de string])
  PushO(Axioma)
  ac ← orac[act(s), TopoO()]
  enquanto ac.t ∉ {Rec, Erro} ⟶
      { se ac.t = avanca ⟶ { PushV(Val(act(s)))
                             PopO()
                             avanca(s)
        ac.t = producao ⟶ { PopO()
                            p ← Prod[ac.numprod]
                            PushO(p.sem)
                            para simb ∈ inverte(p.direito) fazer
                                { PushO(simb) }
        ac.t = semantica ⟶ { execsemantica(ac.sem) }
      ac ← orac[act(g), topoO()] }
  suc ← ac.t = Rec }
```

Por exemplo, no caso da calculadora, teríamos:

```
proc execsemantica ( a : int)
{ se a = 1 ⟶ { val1 ← PopV()
               val2 ← PopV()
               escreve(val1)
  a = 2 ⟶ { val1 ← PopV()
             val ← val1
  a = 3 ⟶ { val1 ← PopV()
             val2 ← PopV()
             val3 ← PopV()
             val ← val2
  a = 4 ⟶ { val1 ← PopV()
             val2 ← PopV()
             val ← soma(val2)
  a = 5 ⟶ { val1 ← PopV()
             val2 ← PopV()
             val ← mult(val2)
  a = 6 ⟶ { val1 ← PopV()
             val2 ← PopV()
             val ← cons(val1, val2)
  a = 7 ⟶ { val ← <>
  PushV(val) }
```

Saliente-se ainda que a determinação de **valX** podia ter sido gerada automaticamente com base no comprimento do lado direito da produção a que esta acção diz respeito.

Exercício 4.4 *Estudar o efeito de implementar a stack de valores com um array (STACK) e um índice actual (SP) e substituindo as ocorrências de valX por val(X) definido como:*

STACK [SP - compDoLadoDireito + X]
De modo a que por exemplo a acção associada à produção

$$Lista \rightarrow Exp\ Lista$$

pudesse ser escrita como:

$$\begin{cases} ... \\ \textbf{se} \quad a = 6 \longrightarrow \begin{cases} val \leftarrow cons(val(1), val(2)) \\ ... \end{cases} \\ \qquad ... \longrightarrow \qquad ... \\ SP \leftarrow SP - compDoLadoDireito(a) \\ PushV(val) \end{cases}$$

4.4 Conflitos LL

Nos exemplos anteriormente analisados não surgiram problemas já que foi possível decidir qual o caminho (produção) a seguir com base no cálculo dos lookahead de cada produção.

Por vezes surgem situações em que os lookahead de duas produções associadas a um mesmo lado esquerdo **A**, têm elementos comuns, colocando-nos no dilema de qual caminho seguir quando no reconhecimento desse símbolo não terminal **A** aparecer um desses terminais comuns.

Essa situação chama-se conflito LL1 e corresponde a uma impossibilidade de construção de reconhecedor TopDown dos tipos atrás descritos.
$conflitoll1(G) = \exists A \rightarrow rhs1, A \rightarrow rhs2 \in Porducoes :$
$lookahead(A \rightarrow rhs1) \cap lookahead(A \rightarrow rhs2) \neq \{\}$

Em termos dos reconhecedores Recursivo descendente os conflitos correspondem a situações em que existe mais do que uma guarda

$$simbseg \in lookahead(A \rightarrow rhs1)$$

verdadeira, portanto mais que um caminho possível (ou ainda não determinismo do algoritmo). Em termos do reconhecedor TopDown dirigido por tabela os conflitos correspondem à tentativa de colocar duas acções diferentes dentro da mesma posição do oráculo.

4.4.1 Analise breve de alguns conflitos LL1

Gramáticas ambíguas

Uma gramática ambígua tem sempre conflitos (LL1 e não só)
Observação: Se há mais que um caminho possível para o reconhecimento, então existe pelo menos um não terminal para o qual haja duas produções alternativas possíveis para uma mesma string a reconhecer.
Sugestão: reescrever a gramática dum modo não ambíguo com base em informação acerca das prioridades e associatividades dos operadores.

Gramáticas recursivas à esquerda

Uma gramática com recursividade à esquerda tem conflitos LL1
Observação: considere-se a seguinte gramática:

$$\begin{array}{rcl} A & \rightarrow & A\ X \\ & | & Y \end{array}$$

então

$$lookahead(A \rightarrow A\ X) \cap lookahead(A \rightarrow Y) \supseteq First\_str(Y)$$

porquê?
Sugestão: reescrever a gramática com recursividade à direita:

$$
\begin{array}{lcl}
A & \rightarrow & Y\ cont \\
cont & \rightarrow & X\ cont \\
 & | & \varepsilon
\end{array}
$$

Saliente-se ainda que a recursividade indirecta à esquerda também é fonte de conflitos.

Produções a começar pelo mesmo símbolo

Uma gramática em que existam duas produções com o mesmo lado esquerdo e cujo lado direito começe pelo mesmo símbolo, têm conflitos LL1.
Observação: considere-se a seguinte gramática:

$$
\begin{array}{lcl}
A & \rightarrow & X\ Y \\
 & | & X\ Z
\end{array}
$$

então

$$lookahead(A \rightarrow X\ Y) \cap lookahead(A \rightarrow X\ Z) \supseteq First(X)$$

porquê? Sugestão: reescrever a gramática "colocando esse símbolo em evidência":

$$
\begin{array}{lcl}
A & \rightarrow & X\ cont \\
cont & \rightarrow & Y \\
 & | & Z
\end{array}
$$

5 Reconhecedores Bottom-Up

5.1 Funcionamento geral

Este método vai basear-se num objecto a que chamaremos **conclusões** que corresponde à interpretação que foi dada à string analisada até ao momento.

As conclusões serão uma sequência de símbolos terminais ou não:

$$conclusoes : (T|NT) - seq$$

Inicialmente as conclusões são nenhumas, ou seja <>.

Quando se analisa um novo símbolo da string de entrada, isso corresponde, numa primeira fase, a acrescentá-lo às conclusões. A esta acção chamaremos **Transição**.

Em certas situações é possível substituir um conjunto de símbolos das conclusões (correspondente a um lado direito de uma produção) por um símbolo não terminal que os "resuma"(o respectivo lado esquerdo). A esta acção chamaremos **redução**.

O objectivo final é obter o axioma como conclusão e ter analisado a totalidade da string de entrada. Neste caso diremos que a string foi reconhecida com sucesso.

Em resumo, o método vai basear-se nas seguintes acções possíveis:

- "Transição- aceitação do próximo terminal e acrescentar o mesmo à string de conclusões.
- "Redução- substituição de um sufixo α de conclusões pelo símbolo não terminal A tal que $A \rightarrow \alpha$
- "Aceitação- proximo símbolo = $ (fim de string) e conclusões = <axioma>
- "Erro- não é possível reduzir nem transitar.

Invariante do método :

$$conclusoes \Rightarrow^* [\text{string j'a analisada}]$$

5.1.1 Autómato LR0

A decisão de qual a acção a efectuar em cada situação vai ser tomada com base num autómato cujos estados descrevem situações.

Cada estado do autómato LR0 vai descrever uma situação e cada sequência conclusões vai estar associada a um percurso dentro desse autómato.

Consideremos por exemplo a seguinte gramática:

```
Z → S $          (0)
S → A a          (1)
  |   b           (2)
A → a A          (3)
  |   c           (4)
```

Cada item de um estado vai estar associado a uma situação possível. A posição actual é representada por um ".".

O item inicial será portanto `[ Z → . S $ ]`

Quando a posição actual é antes de um símbolo não terminal, então fazem também parte desse mesmo estado os items referentes às produções em que esse símbolo não terminal pode derivar, com o "."posicionado no início.

Fazem portanto parte do estado inicial os item `[ S → . A a ]` e `[ S → . b ]` e ainda os item `[ A → . a A ]` e `[ A → . c ]` ou seja o fecho do item inicial.

Os estados seguintes vão ser obtidos partindo de estados já existentes e transitando (i. e. avançar o "."um símbolo) por um símbolo que apareça imediatamente a seguir a cada ".".

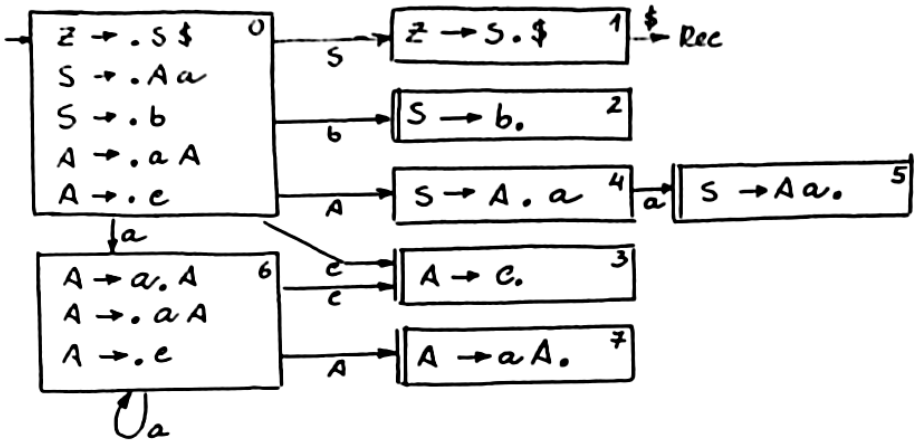


Figura 1: autómato LR0

Saliente-se que neste autómato existem certos estados (exemplo 2,5,3,7) que correspondem a situções em que podemos concluir que uma redução pode ser feita.

Os item em que a posição actual (".") é o fim da produção dizem-se **item de redução** (exemplo os item contidos nos estados 2,5,3,7 são item de redução). Os item em que a posição actual está imediatamante antes de um símbolo terminal dizem-se **item de transição terminal** os outros são chamados **item de transição não terminal**.

Vamos de seguida construir uma tabela que me descreva as acções a efectuar em função do estado em que estamos e do símbolo que nos surge. Os símbolos irão surgir por leitura da string de entrada (terminais) ou por substituição de outros previamente existentes (redução).

	$	a	b	c		S	A
0		6	2	3		1	4
1	rec						
2	P2	P2	P2	P2			
3	P4	P4	P4	P4			
4		5					
5	P1	P1	P1	P1			
6		6		3			7
7	P3	P3	P3	P3			

Legenda:

- (vazio) - erro
- rec – reconhecimento
- Pn – Redução pela produção nº n
- n – Transição para o estado nº n

Esta tabela é composta de duas partes Tabela T (também chamada tabela ACCAO)(composta pelas colunas referentes aos simbolos terminais) e a tabela NT (tambem chamada tabela GOTO)(referente aos simbolos não terminais). Como era de prever, um simbolo não terminal surge após uma redução levando a que a acção associada ao seu aparecimento seja invariavelmente uma transição.

Vamos de seguida exemplificar o funcionamento do reconhecedor BottomUp com uma tabela em que se vai apresentando as sucessivas modificações de:

- caminho percorrido até ao momento,
- conclusões (i.e. interpretação que foi feita da string já analisada)
- String que falta analisar
- acção seguinte
- árvore de derivação

caminho	conclusões	string	acção seg	arvore
0		aaca$	trans a 6	a a c a $
06	a	aca$	trans a 6	
066	aa	ca$	trans c 3	
0663	aac	a$	reduc P4	A
066	aaA	a$	trans A 7	
0667	aaA	a$	reduc P3	A
06	aA	a$	trans A 7	
067	aA	a$	reduc P3	A
0	A	a$	transNT A 4	
04	A	a$	trans a 5	
045	Aa	$	reduc P1	S
0	S	$	trans S 1	
01	S	$	Reconheci	

Como vimos há uma ligeira diferença entre as transições por símbolos terminais e por símbolos não terminais. Na primeira situação há que avançar na string de entrada enquanto que na segunda não há. Daí serem normalmente consideradas duas acções distintas:

- trans_T – transição por terminal (ou simplesmente emphtrans)
- trans_NT – transição por não terminal

Saliente-se ainda que o *caminho* e as *conclusões* são duas representações da mesma informação. Repare-se que, por exemplo, associado ao caminho 0667 está implicitamente associado *aaA* já que é o peso dos ramos utilizados no percurso (`0-a-6-a-6-A-7`).

Saliente-se ainda que a árvore de derivação foi construída das folhas para a raíz (daí o nome de BottomUp).

Analisando ainda a evolução de "caminho"verificamos que as operações que sobre ele são efectuadas são:

- transição – acresenta um elemento no fim de "caminho"
- redução – retira vários elementos do fim de "caminho"e insere um novo elemento no fim de "caminho"

Dadas as suas características sequenciais e o facto de os elementos serem inseridos e retirados do mesmo lado de caminho, concluimos que este objecto pode ser modelado por uma **stack de (T | NT)**.

5.2 Algoritmo LR

Na sequência da secção anterior vamos considerar os seguintes objectos:

Objecto	modelo	funções
Caminho	stack(T ou NT)	iniciaC: $\rightarrow$ pushC:T ou NT $\rightarrow$ popC: $\rightarrow$ topoC: $\rightarrow$ T ou NT
String	stream(T)	iniciaS: $\rightarrow$ avanca: $\rightarrow$ act: $\rightarrow$ T

proc *reconhecedorLR* (**in** $Orac : automatoLR$
in $s : stream(T)$
in $prod : producoes$
in $semantica : procedimento$
out $suc : bool$)

$$
\left\{
\begin{array}{l}
iniciaS() \\
iniciaC() \\
pushC(EstadoInicial) \\
ac \leftarrow orac[topoC, act(s)] \\
\textbf{enquanto}\ \ ac.tipo \notin \{Rec, Erro\} \longrightarrow \\
\quad \left\{
\begin{array}{lll}
\textbf{se} \quad ac.tipo = trans \longrightarrow & \left\{
\begin{array}{l}
pushC(ac.estado) \\
\quad \textbf{se} \quad trans\_T \longrightarrow \left\{ avanca(s) \right. \\
\qquad\quad \textbf{senão} \longrightarrow \left\{ skip \right. \\
ac \leftarrow orac[topoC, act(s)]
\end{array}
\right. \\
ac.tipo = Reduc \longrightarrow & \left\{
\begin{array}{l}
p \leftarrow ac.producao \\
\textbf{para}\ \ i \in 1, 2, ..., prod[p].comp\ \textbf{fazer} \\
\quad \left\{ popC() \right. \\
semantica(p) \\
ac \leftarrow orac[topoC, prod[p].lhs]
\end{array}
\right.
\end{array}
\right. \\
suc \leftarrow ac.tipo = Rec
\end{array}
\right.
$$

Nota: as considerações feitas acerca da semantica nos reconhecedores TopDown nomeadamente acerca da *stack(valor)*, são válidas também neste contexto.

5.3 Conflitos LR0

Como vimos na secção anterior, o autómato LR0 baseia-se em 3 tipos de item:

- item de redução : associado a acções de redução
- item de transição por símbolo terminal : associado a acções trans_T
- item de transição por símbolo não terminal : associado a trans_NT

As acções trans_T de um mesmo estado nunca "colidem"entre si (já que estão associadas a terminais diferentes, ou então transitão para o mesmo estado). As acções trans_NT de um estado não colidem com nenhuma outra acção já que essas transições são forçadas por reduções anteriores, e surgem como parte integrante dessas mesmas reduções.

Porém quando num estado existem simultaneamente dois item de redução diferentes (portanto haja hipótese de tirar duas conclusões diferentes), iremos ter uma situação de conflito **Redução-Redução**, conflito esse que corresponde a querer preencher a linha da tabela T referente a esse estado com duas acções diferentes.

Quando num estado haja um item de redução e um item de transição por simbolo terminal (portanto haja hipótese de tirar já conclusões ou então, se o simbolo seguinte permitir a transição, transitar), estamos numa situação semelhante, só que o problema não se refere à linha inteira. Este conflito é chamado de **Transição–Redução**.

consideremos a seguinte gramática

(0) Z → S $
(1) S → A a
(2) | b
(3) A → A a
(4) | ε

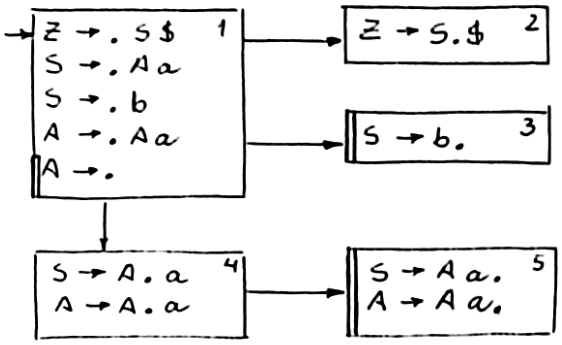


Figura 2: autómato LR0

	$	a	b	S	A	
1	P4	P4	P4 3	2	4	(conflito Tr-Red)
2	Rec					
3	P2	P2	P2			
4		5				
5	P1 P3	P1 P3	P1 P3			(conflito Red-Red)

Em resumo, analisando o autómato LR0 detectamos conflitos:

- Redução-Redução quando num mesmo estado existem dois item de redução. (exemplo: estado 5).
- Transição-Redução quando num mesmo estado existe um item de redução e um ou mais item de transição por símbolo terminal. (exemplo: estado 1 entre os item `[ S → . b ]` e o item `[ S → . ]`).

5.4 Automatos SLR1

Na construção das Tabelas LR0, as reduções foram efectuadas incondicionalmente, i. e., sem ter em conta o(s) simbolo(s) terminal(is) que se seguem[3].

Quando há conflitos somos forçados a ser mais selectivos nas reduções a fazer.

[3]O algarismo "0"de LR0 significa precisamente isso: reduções feitas olhando aos zero simbolos seguintes

O método **SLR1** vai basear-se no autómato LR0 e no cálculo dos **follow** dos não terminais da gramática. Todos os item de redução `[B → cont .]` irão ser substituídos por item LR1 da forma `[ B → cont . | follow(B) ]`, i. e., o item passa a incluir não só o item LR0 como o conjunto de continuações possíveis. Essas continuações são chamadas **Prefixos** do que se segue.

Os prefixos_SLR1(`[ A → cont . ]` = follow(A).

Na construção das tabelas SLR1, as reduções `[ B → cont . ]`, só serão selecionadas quando o símbolo terminal que se seguir pertencer às continuações válidas daquele item (ou seja pertencer a *follow(B)*).

```
follow(S) = {$}
follow(A) = {a}
```

	$	a	b	S	A
1		P4	3	2	4
2	Rec				
3	P2				
4		5			
5	P1	P3			

Comparando com a tabela LR0 anteriormente calculada, observamos que os conflitos foram levantados e que noutras situações passa a haver uma detecção de erros mais cedo.

Há gramáticas que apresentam conflitos SLR1, i. e., em que o método SLR1 não vai ser suficiente para decidir qual a acção seguinte a efectuar.

Uma gramática verifica a condição SLR1 sse em qualquer estado do seu automato LR0 não existem conflitos SLR1.

5.4.1 Conflitos SLR1 Redução-Redução

Definição:
Um estado apresenta conflitos redução-redução se:
Existem nesse estado 2 item `[A → cont . |p]` e `[A' → cont' . |p']` em que `p = follow(A)` e `p' = follow(A')` de tal modo que $p \cap p' \neq \{\}$

5.4.2 Conflitos SLR1 Transição-Redução

Definição:
Um estado apresenta conflitos Transição-redução se:
Existem nesse estado 2 item `[A → inic . a cont ]` e `[A' → cont' . |p']` em que **a** $\in$ **T** `p' = follow(A')` de tal modo que $a \in p'$

5.5 Automatos LALR1

O método LALR1 segue a mesma ideia que o método SLR1, só que os prefixosLALR1 são mais selectivos que os prefixosSLR1. Deste modo certos conflitos SLR1 são levantados por este método.

O cálculo dos prefixosLALR1 de cada item vai levar em conta donde é que esse item proveio: vai ter em conte o contexto que deu origem ao item.

5.5.1 Calculo dos prefixosLALR1

PrefixoLALR(`estadoInical,[Z → .S $]`) = {}

PrefixoLALR(`q:estado,[A → h X . j ]`)

$$\begin{cases} \textbf{para } q' \in [\text{estados com [A -> h . X j]] } \textbf{fazer} \\ \quad \{\ la \leftarrow la \cup parefixoLALR(q', [A->h.X\ j]) \end{cases}$$

PrefixoLALR(`q:estado,[A → h . ]`)

$$\begin{cases} \textbf{para } [B \to x.A\ y] \in q \textbf{ fazer} \\ \quad \{\ la \leftarrow la \cup First(y.prefixoLALR(q, [B->x.A\ y])) \end{cases}$$

nota: se y é não anulável, podemos retirar prefixoLALR([B –> x . A y]); (porquê?)

Os prefixosLALR1 calculam-se resolvendo iterativamente o sistema de equações 1,2,3 percorrendo o autómato LR0 em cada iteração.

Na prática apenas nos interessam calcular os prefixosLALR dos item de redução.

5.5.2 Exemplo

consideremos a seguinte gramática

(0) Z → S $
(1) S → L = R
(2) | R
(3) L → * R
(4) | id
(5) R → L

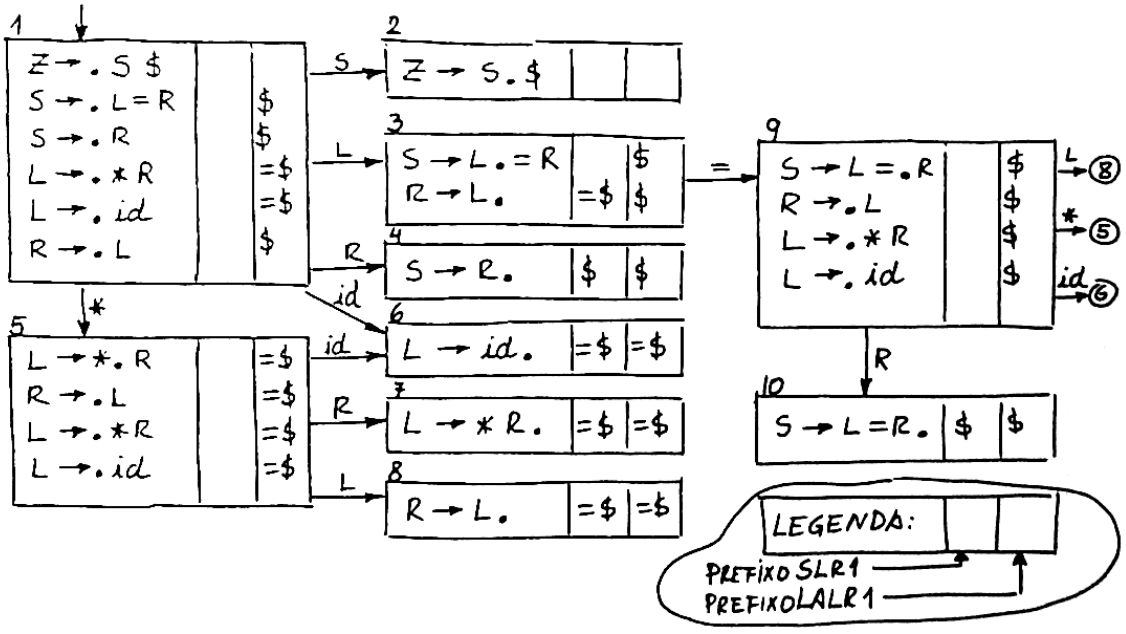


Figura 3: automato LR0, SLR1, LALR1

Observações:

- Existe conflito LR0 no estado 3
- Continua a haver conflito SLR1 no estado 3 ($=\in \{\$, =\}$)
- $prefixoLALR(3, [R \rightarrow L.]) = prefixoLALR(1, [R \rightarrow .L]) =$
  $prefixoLALR(1, [S \rightarrow .R]) = \$$
  Não há conflito LALR1

Teorema: $prefixosLALR1(q, A \rightarrow h.) \subseteq follow(A)$
A coincidência dos dois conjuntos dá-se quando o item em causa aparece num único estado.

Teorema: Os item de redução que ocorrem num unico estado do automato LR0 têm $prefixosLALR1(q, A \rightarrow h.) = follow(A)$

Exercício 5.1 *Considere a seguinte gramática:*

```
Z → S $
S → A a A b
  |  B a B a
A →             /* vazio */
B →             /* vazio */
  |  c
```

a) mostre que tem conflitos LR0, SLR1
b) mostre que nao tem conflitos LALR1

5.5.3 Condição LALR1

Uma gramática verifica a condição LALR1 sse em qualquer estado do seu automato LR0 não existem conflitos LALR1.

5.5.4 Conflitos LALR1 Redução-Redução

Definição:
Um estado **q** apresenta conflitos redução-redução se:
Existem nesse estado 2 item `[A → cont . ]` e `[A' → cont' . ]` de tal modo que: $PrefixoLALR(q, [A \rightarrow cont.]) \cap PrefixoLALR(q, [A' \rightarrow cont.]) \neq \{\}$

5.5.5 Conflitos LALR1 Transição-Redução

Definição:
Um estado **q** apresenta conflitos Transição-redução se:
Existem nesse estado 2 item `[A → inic . a cont ]` e `[A' → cont' .]` em que $\mathbf{a} \in \mathbf{T}$ de tal modo que $a \in prefixoLALR(q, [A' \rightarrow cont'.])$

5.6 Algoritmo com acções Transição-Redução

Duma análise dos autómatos anteriores verificamos que é frequente a existência de estados em que aparece um unico item de redução.

Durante o funcionamento, a esses estados vão estar associados transições garantidamente seguidas de reduções. Uma variante do método LR consiste em considerar mais um tipo de acção (a acção transição Redução) que consiste em fundir estas duas.

Com este tipo de acção torna-se possível eliminar os estados que contêm apenas uma redução permitindo reduzir o tamanho das tabelas T e NT e tornar ligeiramente mais rápido o algoritmo.

Tomando o exemplo anterior ao usarmos acções transição redução, eliminamos os estados 4,6,7,8,10 obtendo a seguinte tabela:

	=	*	id	$	S	L	R
1		5	tr4		2	3	tr2
2				Rec			
3	9			R5			
5		5	tr4			tr5	tr3
9		5	tr4			tr5	tr1

Legenda: tr5 = transicao reducao segundo a producao 5

O algoritmo com as novas acções seria:

proc *reconhecedorLR* (**in** $Orac : automatoLR$
in $s : stream(T)$
in $prod : producoes$
in $semantica : procedimento$
out $suc : bool$)

$iniciaS()$
$iniciaC()$
$pushC(EstadoInicial)$
$ac \leftarrow orac[topoC, act(s)]$
enquanto $ac.tipo \notin Rec, Erro \longrightarrow$
- **se** $ac.tipo = Trans \longrightarrow$
  - $pushC(ac.estado)$
  - **se** $Trans\_T \longrightarrow$ $avanca(s)$
    senão $\longrightarrow$ $skip$
  - $ac \leftarrow orac[topoC, act(s)]$
- $ac.tipo = Reduc \longrightarrow$
  - $p \leftarrow ac.producao$
  - **para** $i \in 1, 2, ..., prod[p].comp$ **fazer** $popC()$
  - $semantica(p)$
  - $ac \leftarrow orac[topoC, prod[p].lhs]$
- $ac.tipo = TransRed \longrightarrow$
  - $p \leftarrow ac.producao$
  - **se** $TransRed\_T \longrightarrow$ $avanca(s)$
    senão $\longrightarrow$ $skip$
  - **para** $i \in 2, 3, ..., prod[p].comp$ **fazer** $popC()$
  - $semantica(p)$
  - $ac \leftarrow orac[topoC, prod[p].lhs]$

$suc \leftarrow ac.tipo = Rec$

6 Análise de um Exemplo

6.0.1 Enunciado

Pretende-se construir um interpretador para cálculo de expressões numéricas envolvendo somas, subtrações, multiplicações e divisões podendo ainda ser usados parentesis. Os operandos serão reais podendo (ou não ter sinal).

Pretende-se ainda que a habitual relação de prioridades dos operadores binários seja respeitada. Os operadores binários descritos são associativos à esquerda.

Objectivos:

a) Escrever uma gramática para descrever expressão .

b) Constatar a impossibilidade de traduzir esta gramatica para expressão regular (e portanto constatar a impossibilidade de construir um reconhecedor para este tipo de expressões baseado em autómatos determinísticos). (Solução não apresentada).

c) construir uma calculadora baseada no método de reconhecimento recursivo descendente

d) construir uma calculadora baseada no método de reconhecimento LR

e) Comparar a maneira como estão escritas as acções semânticas num caso e noutro (a alinea d necessita de stacks auxiliares para simular a recursividade que existe na alinea c)

f) construir um reconhecedor usando LEX e YACC

6.1 Escrita da gramatica

Hipótese 1:

Z	→	E $	
E	→	E + E	
	|	E – E	
	|	E * E	
	|	E / E	
	|	rss	**r**eal **s**em **s**inal
	|	+ rss	
	|	– rss	
	|	(E)	

Exercício 6.1 *Constactar que esta gramática inclui todos os casos de expressões referidas mas é ambigua e não contem informação nenhuma acerca de prioridades e associatividades dos operadores.*

Hipótese 2:

Z	→ E $	
E	→ E opadt P | P	E=soma de parcelas
P	→ P opmult F | F	P=produto de factores
F	→ opun F | (E) | rss	
opadt	→ + | –	operador aditivo
opmult	→ * | /	operador multiplicativo
opun	→ + | –	operador unario

Exercício 6.2 *Construir a árvore de derivação de algumas expressões e constate que a gramática não é ambígua e que nas derivações correspondente a somas de mais que duas parcelas, está a ser implicitamente feita uma associação à esquerda.*

Exercício 6.3 *Verificar que esta gramática não é LL1 (recursiva à esquerda $\Longrightarrow$ não LL1)*

Exercício 6.4 *Verificar que esta gramática não é LR0*

Exercício 6.5 *Verificar que esta gramática é SLR1*

Hipótese 3:

Z	→ E $	
E	→ P opadt E | P	E=soma de parcelas
P	→ F opmult P | F	P=produto de factores
F	→ opun F | (E) | rss	
opadt	→ + | –	operador aditivo
opmult	→ * | /	operador multiplicativo
opun	→ + | –	operador unario

Exercício 6.6 *Construir a árvore de derivação de algumas expressões e constate que a gramática não é ambígua e que nas derivações correspondente a somas de mais que duas parcelas, está a ser implicitamente feita uma associação à direita.*

Exercício 6.7 *Verificar que esta gramática não é LL1*

Exercício 6.8 *Verificar que esta gramática não é LR0*

Exercício 6.9 *Verificar que esta gramática é SLR1*

Hipótese 4:

Z	→ E $	
E	→ P Econt	continuacao da expressao
Econt	→ ε | opadt E	
P	→ F Pcont	
Pcont	→ ε | opmult P	continacao da parcela
F	→ opun F | (E) | rss	
opadt	→ + | –	operador aditivo
opmult	→ * | /	operador multiplicativo
opun	→ + | –	operador unario

Exercício 6.10 *Verificar que esta gramática é LL1*

Producoes	Lookahead 1
Z → E $	rss (+ –
E → P Econt	rss (+ –
Econt → ε	) $
Econt → opadt E	+ –
P → F Pcont	rss (+ –
Pcont → ε	) $ + –
Pcont → opmult P	* /
F → opun F	+ –
F → (E)	(
F → rss	rss
opadt → +	+
opadt → –	–
opmult → *	*
opmult → /	/
opun → +	+
opun → –	–

Exercício 6.11 *Verificar que esta gramática não é LR0*

Exercício 6.12 *Verificar que esta gramática é SLR1*

Exercício 6.13 *Construir uma gramática para tratar também de operadores exponenciais (prioridade maior que os multiplicativos e associativadade à direita) e ainda invocação de funções da forma f(... , ... , ...)*

6.2 Reconhecedor recursivo Descendente

Usando a gramática *hip4* vamos construir um reconhecedor com base num conjunto de funções de reconhecimento (uma para cada símbolo) conforme a estrutura de base atrás descrita.

6.2.1 Listagem de um reconhecedor Rec. Desc. em C

```
/*-----------------------------------------------------------------
| Z      → E $                                                    |
| E      → P Econt                                                |
| Econt  →   | opadt E                                             |
| P      → F Pcont                                                |
| Pcont  →   | opmult P                                            |
| F      → opun F | ( E ) | inteiro sem sinal                     |
| opadt  → + | -                                                   |
| opmult → * | /                                                   |
| opun   → + | -                                                   |
-----------------------------------------------------------------*/

#include <ctype.h>
#include <stdio.h>
#include <malloc.h>
#include <string.h>

typedef struct scont            /* valor associado a Econt e Pcont */
        { char op;              /*  (NULL ou (op,val)              */
```

```
           float val; } *cont;

float recZ(),recE(),recP(),recF(),calc();
char recopadt(),recopmult(),recopun();
cont recEcont(),recPcont(),mkcont();
void rec(), erro();

#define nulo(a) (a == NULL)

/*-------------------------------------------------------------------
| Reconhecedor Recursivo descendente; invocar com recZ():float       |
-------------------------------------------------------------------*/
float recZ(){                          /* reconhece e calcula Z       */
  float val_1;
  val_1=recE();
  rec('$');
  return val_1; }

#define recopun() recopadt()        /* operador unario             */

float recE(){                       /* reconhece e calcula Expr    */
  float val_1;
  cont  val_2;
  val_1=recP();
  val_2=recEcont();
  return( nulo(val_2)? val_1: calc(val_1,val_2->op,val_2->val)); }

float recP(){                       /* reconhece e calcula Parcela */
  float val_1;
  cont val_2;
  val_1=recF();
  val_2=recPcont();
  return( nulo(val_2)? val_1: calc(val_1,val_2->op,val_2->val)); }

cont recEcont(){                     /* continuacao de Expressao    */
  float val_2;
  char val_1;
  if     (actual_in(")$"))
      return((cont)NULL);
  else if(actual_in("+-"))
      { val_1=recopadt();
        val_2=recE();
        return( mkcont(val_1,val_2) );
      }
  else erro("continuacao de Expressao invalida"); }

char recopadt(){                    /* operador aditivo           */
  if(actual=='+')
    { rec('+');
      return('+');}
  else if(actual=='-')
    { rec('-');
```

```
      return('-');}
  else erro("invalido op aditivo"); }

char recopmult(){                    /* operador multiplicativo    */
  if(actual=='*')
    { rec('*');
      return('*');}
  else if(actual=='/')
    { rec('/');
      return('/');}
  else erro("invalido op multiplicativo"); }

float recF(){                        /* rec. e calcula factor      */
  float val_2;
  char val_1;
  if(actual=='(')
     { rec('(');
       val_2 =recE();
       rec(')');
       return val_2;
     }
  else if(actual_in("+-"))
     { val_1=recopun();
       val_2=recF();
       return((val_1=='-')?-val_2:val_2);
     }
  else if(actual=='i')
     { rec('i');
       return (valor);
     }
  else erro("termo invalido"); }

cont recPcont(){                     /* rec continuacao parcela  */
  float val_2;
  char val_1;
  if     (actual_in(")$+-"))
      return((cont)NULL);
  else if(actual_in("*/"))
     { val_1=recopmult();
       val_2=recP();
       return( mkcont(val_1,val_2) ); } }

#define avanca actual=get_simbolo()

void rec(char c){                    /* reconhece simb terminal     */
  if(actual == c) avanca;
  else { char aux[70];
         sprintf(aux,"simbolo .%c. quando se esperava .%c.",actual,c);
         erro(aux); }
}

int main(){
```

```
  while(1){
    printf("?\n");
    gets(gama);
    iniciagama;
    printf(" %f\n", recZ()); }
  return 0;
}

/*-------------------------------------------------------------------
| Funcoes referentes 'a string a reconhecer                         |
-------------------------------------------------------------------*/
char gama[100];       /* string a reconhecer                        */
int ult_analisada;    /* apont para a ultima posicao analisada de gama */
char actual;          /* codigo do actual simbolo Termi. de gama    */
float valor;          /* valor associado a actual caso exista       */

#define iniciagama {ult_analisada=-1;actual=get_simbolo();}

char get_simbolo(){                       /* comer espacos iniciais */
  do { ult_analisada++;}
  while(isspace(gama[ult_analisada]));
                                          /* fim de string          */
  if(gama[ult_analisada]=='\0')
     return ('$');
                                          /* numero                 */
  else if(isdigit(gama[ult_analisada]))
     { valor=0;
       do { valor=valor*10 + gama[ult_analisada]-'0';
            ult_analisada++;}
       while(isdigit(gama[ult_analisada]));
       ult_analisada--;
       return ('i'); }
                                          /* operador ou parentesis */
  else if(strchr("()/*+-",gama[ult_analisada]) != NULL)
     return (gama[ult_analisada]);

  else erro("simbolo desconhecido");
}

#define actual_in(s) (strchr(s,actual) != NULL)

/*-------------------------------------------------------------------
| Erro                                                              |
-------------------------------------------------------------------*/
void erro(char *s){  /* escreve a string indicando a posic. actual */
  int i;             /* e depois aborta                            */
  fprintf(stderr,"Erro-%s \n%s\n",s,gama);
  for(i=0;i<ult_analisada;i++) fprintf(stderr," ");
  fprintf(stderr,"^\n");
  exit(1);}

/*-------------------------------------------------------------------
```

```
| Funcoes referentes a semantica                                        |
-----------------------------------------------------------------------*/

cont mkcont(char a, float b){                 /* constroi par operacao,valor */
  cont aux;
  aux=(cont) malloc(sizeof(struct scont));
  aux->op=a;
  aux->val=b;
  return aux; }

float calc(float v1, char op, flloat v2){   /* calcula operacao binaria    */
 switch(op)
 { case '+': return(v1+v2);
   case '-': return(v1-v2);
   case '*': return(v1*v2);
   case '/': return(v1/v2); } }

-----------------------------------------------------------------------
```

6.3 Reconhecedor LR

6.3.1 Comentários gerais

Neste método, a complexidade algorítmica será transferida para estruturas de dados, nomeadamente para as tabelas T e NT, para o vector PRODUÇÕES e para a stack de estados.

As acções semânticas são menos claras que no exemplo anterior já que surge a necessidade de implementar stacks valor para os dados intermédios (no caso anterior era usada implicitamente a stack do sistema e criadas novas instancias das variáveis quando era feita a invocação recursiva).

Analise-se no entanto com cuidado as acções semânticas associadas a cada produção e facilmente se encontrará semelhança com o caso anterior.

São apresentados dois reconhecedores um correspondente à gramática *hip4* (para comparação com o anterior) e uma outra *hip2* que implementa associatividade à esquerda e que não sendo LL1 não podia ser reconhecida pelo método recursivo descendente. Saliente-se também que a segunda versão é mais natural que a primeira.

6.3.2 Cálculo do automato LR0 e SLR1

Apresenta-se de seguida o automato LR0 correspondente à *hip2* tendo surgido alguns conflitos que foram levantados segundo o método SLR1.

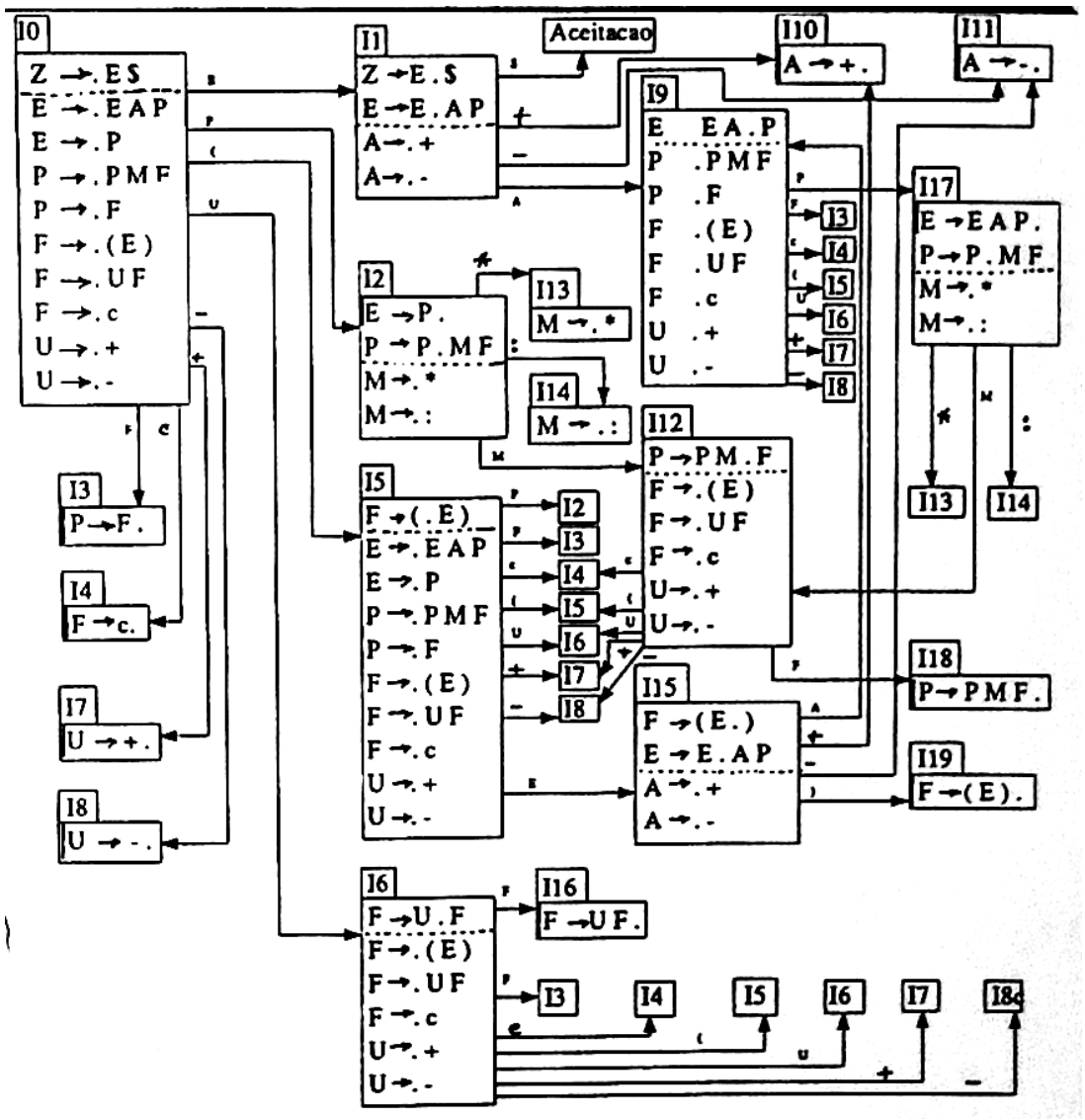


Figura 4: automato LR0

O método SLR1 os conflitos são *levantados* da seguinte maneira:

A redução pela produção $A \rightarrow \alpha$ só se faz se seguir um símbolo pertencente a *follow(A)*. Para isso devemos calcular os conjuntos Follow_1 de cada símbolo não terminal.

Simbolo	**First 1**	**Follow 1**
E	c (+ −	$ + −)
P	c (+ −	$ + −) * /
F	c (+ −	$ + −) * /
opadt	+ −	c (+ −
opmult	* /	c (+ −
opun	+ −	c (+ −

Deste autómato derivam directamente as tabelas usadas no algoritmo LR.

Note-se que nas tabelas as diversas acções

acção = união(transição, redução, erro, aceitação)

estão codificadas com um único inteiro por questões de eficiência (sendo a sua aparência e legibilidade melhorada à custa dos *define* utilizados)

Exercício 6.14 *Reescreva o autómato anterior usando acções Transição-Redução, levando a uma diminuição grande do nº de estados*

6.3.3 Listagem C de um reconhecedor SLR1

```
-----------------------------------------------------------------------
#include <stdio.h>
#include <ctype.h>
#define TRANSICAO 1
#define REDUCAO   2
#define ACEITACAO 3
#define ERRO      4

/*************************************************************************/
/**                                                                     **/
/**                          TABELAS DO PARSER                          **/
/**                                                                     **/
/*************************************************************************/

#define TERMINAIS   8
#define NAOTERMINAIS  6
#define MAXPROD 13
#define ESTADOS 20
#define PRODUCAO(x) (x-30)

#define ERR -1
#define AC  100
#define T(x) x                  /* accao de transicao para o estado x */
#define P(x) (30+x)             /* accao de reducao pela producao x */

int TabelaT [ESTADOS][TERMINAIS] =
{       /*   0       1       2       3       4       5       6       7
             $       )       (       :       *       -       +       c */
/*  0 */   ERR  ,  ERR  ,  T(5)  ,  ERR  ,  ERR  ,  T(8)  ,  T(7)  ,  T(4)  ,
/*  1 */    AC  ,  ERR  ,  ERR   ,  ERR  ,  ERR  , T(11)  , T(10)  ,  ERR   ,
/*  2 */   P(1) ,  P(1) ,  ERR   , T(14) , T(13) ,  P(1)  ,  P(1)  ,  ERR   ,
/*  3 */   P(3) ,  P(3) ,  ERR   ,  P(3) ,  P(3) ,  P(3)  ,  P(3)  ,  ERR   ,
/*  4 */   P(4) ,  P(4) ,  ERR   ,  P(4) ,  P(4) ,  P(4)  ,  P(4)  ,  ERR   ,
/*  5 */   ERR  ,  ERR  ,  T(5)  ,  ERR  ,  ERR  ,  T(8)  ,  T(7)  ,  T(4)  ,
/*  6 */   ERR  ,  ERR  ,  T(5)  ,  ERR  ,  ERR  ,  T(8)  ,  T(9)  ,  T(4)  ,
/*  7 */   ERR  ,  ERR  , P(11)  ,  ERR  ,  ERR  , P(11)  , P(11)  , P(11)  ,
/*  8 */   ERR  ,  ERR  , P(12)  ,  ERR  ,  ERR  , P(12)  , P(12)  , P(12)  ,
/*  9 */   ERR  ,  ERR  ,  T(5)  ,  ERR  ,  ERR  ,  T(8)  ,  T(7)  ,  T(4)  ,
/* 10 */   ERR  ,  ERR  ,  P(7)  ,  ERR  ,  ERR  ,  P(7)  ,  P(7)  ,  P(7)  ,
/* 11 */   ERR  ,  ERR  ,  P(8)  ,  ERR  ,  ERR  ,  P(8)  ,  P(8)  ,  P(8)  ,
/* 12 */   ERR  ,  ERR  ,  T(5)  ,  ERR  ,  ERR  ,  T(8)  ,  T(7)  ,  T(4)  ,
/* 13 */   ERR  ,  ERR  ,  P(9)  ,  ERR  ,  ERR  ,  P(9)  ,  P(9)  ,  P(9)  ,
/* 14 */   ERR  ,  ERR  , P(10)  ,  ERR  ,  ERR  , P(10)  , P(10)  , P(10)  ,
/* 15 */   ERR  , T(19) ,  ERR   ,  ERR  ,  ERR  , T(11)  , T(10)  ,  ERR   ,
/* 16 */   P(6) ,  P(6) ,  ERR   ,  P(6) ,  P(6) ,  P(6)  ,  P(6)  ,  ERR   ,
```

```
/* 17 */   P(0) ,  P(0) ,  ERR   , T(14) , T(13) ,  P(0)  ,  P(0)  ,  ERR   ,
/* 18 */   P(2) ,  P(2) ,  ERR   ,  P(2) ,  P(2) ,  P(2)  ,  P(2)  ,  ERR   ,
/* 19 */   P(5) ,  P(5) ,  ERR   ,  P(5) ,  P(5) ,  P(5)  ,  P(5)  ,  ERR
};

int TabelaNT [ESTADOS][NAOTERMINAIS] =
{       /*   0       1       2       3       4       5
             E       P       F       A       M       U */
  /*  0 */   T(1) ,  T(2) ,  T(3) ,  ERR  ,  ERR  ,  T(6) ,
  /*  1 */   ERR  ,  ERR  ,  ERR  ,  T(9) ,  ERR  ,  ERR  ,
  /*  2 */   ERR  ,  ERR  ,  ERR  ,  ERR  , T(12) ,  ERR  ,
  /*  3 */   ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,
  /*  4 */   ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,
  /*  5 */  T(15) ,  T(2) ,  T(3) ,  ERR  ,  ERR  ,  T(6) ,
  /*  6 */   ERR  ,  ERR  , T(16) ,  ERR  ,  ERR  ,  T(6) ,
  /*  7 */   ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,
  /*  8 */   ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,
  /*  9 */   ERR  , T(17) ,  T(3) ,  ERR  ,  ERR  ,  T(6) ,
  /* 10 */   ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,
  /* 11 */   ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,
  /* 12 */   ERR  ,  ERR  , T(18) ,  ERR  ,  ERR  ,  T(6) ,
  /* 13 */   ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,
  /* 14 */   ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,
  /* 15 */   ERR  ,  ERR  ,  ERR  ,  T(9) ,  ERR  ,  ERR  ,
  /* 16 */   ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,
  /* 17 */   ERR  ,  ERR  ,  ERR  ,  ERR  , T(12) ,  ERR  ,
  /* 18 */   ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,
  /* 19 */   ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR  ,  ERR
};

int PRODUCOES [MAXPROD][2] =
{ /* N. da Prod    PRODUCAO          N.S.Direito   S.Esquerdo */
  /*    0        E → E A P     */       3       ,     0      ,
  /*    1        E → P         */       1       ,     0      ,
  /*    2        P → P M F     */       3       ,     1      ,
  /*    3        P → F         */       1       ,     1      ,
  /*    4        F → c         */       1       ,     2      ,
  /*    5        F → ( E )     */       3       ,     2      ,
  /*    6        F → U F       */       2       ,     2      ,
  /*    7        A → +         */       1       ,     3      ,
  /*    8        A → -         */       1       ,     3      ,
  /*    9        M → *         */       1       ,     4      ,
  /*   10        M → :         */       1       ,     4      ,
  /*   11        U → +         */       1       ,     5      ,
  /*   12        U → -         */       1       ,     5
};


/*****************************************************************************/
/**                                                                         **/
/**                              DA_SIMBOLO                                 **/
/**                                                                         **/
```

```
/*****************************************************************************/

char gama [100];
int p;
float numero;

int da_simbolo () {
  float pot;
  int R;

  /* come espacos */
  while ((gama [p] != 0) && (gama [p] <= 32))
     p++;
  if (isdigit (gama [p]) || (gama [p] == '.'))
     { /* e' um numero */
       R = 7;
       numero = 0.0;
       while (isdigit (gama [p]))
         { numero = (numero * 10.0) + (float) (gama [p] - '0') ;
           p++ ;
         }
       if (gama [p] == '.')
          { p++; pot = 0.1;
            while (isdigit (gama [p]))
              { numero = numero + (pot * (float) (gama [p] - '0')) ;
                p++; pot = pot / 10.0;
              }
          }
     }
  else { switch (gama [p])
          { case '+' : R = 6; break;
            case '-' : R = 5; break;
            case '*' : R = 4; break;
            case ':' :
            case '/' : R = 3; break;
            case '(' :
            case '[' :
            case '{' : R = 2; break;
            case ')' :
            case ']' :
            case '}' : R = 1; break;
            case 0   : R = 0; break;
            default  : R = -1;
          }
         p++;
       }
  return (R);
}

/*****************************************************************************/
/**                                                                         **/
/**                            ACCOES SEMANTICAS                            **/
```

```
/**                                                                          **/
/******************************************************************************/


char tipodeadicao [100], tipodemultiplicacao [100], tipodeunario [100];

#define pop_adicao tipodeadicao [(tipodeadicao [0])--]
#define pop_multiplicacao tipodemultiplicacao [(tipodemultiplicacao [0])--]
#define pop_unario tipodeunario [(tipodeunario [0])--]
#define push_adicao(x) tipodeadicao [++(tipodeadicao [0])] = x
#define push_multiplicacao(x) tipodemultiplicacao [++(tipodemultiplicacao [0])] = x
#define push_unario(x) tipodeunario [++(tipodeunario [0])] = x
float stack [100];
int stackpointer;

init_stack () {
  stackpointer = -1; }

float topo_stack () {
  return (stack [stackpointer]); }

float pop_stack (){
  stackpointer = stackpointer - 1;
  return (stack [stackpointer + 1]); }

push_stack (float x){
  stackpointer = stackpointer + 1;
  stack [stackpointer] = x; }

executa_accao (int n){
  float o1, o2, Result;
  char tipo;

  switch (n)
    { case  0 : /* E → E M P */
                o1 = pop_stack ();
                o2 = pop_stack ();
                tipo = pop_adicao ;
                switch (tipo)
                   { case '+' : Result = o2 + o1; break;
                     case '-' : Result = o2 - o1;
                   }
                push_stack (Result);
                break;
      case  1 : /* E → P     */
                break;
      case  2 : /* P → P M F */
                o1 = pop_stack ();
                o2 = pop_stack ();
                tipo = pop_multiplicacao;
                switch (tipo)
                   { case '*' : Result = o2 * o1; break;
```

```
                    case ':' : Result = o2 / o1;
                  }
               push_stack (Result);
               break;
      case  3 : /* P → F     */
               break;
      case  4 : /* F → c     */
               push_stack (numero);
               break;
      case  5 : /* F → ( E ) */
               break;
      case  6 : /* F → U F   */
               o1 = pop_stack ();
               tipo = pop_unario;
               switch (tipo)
                  { case '+' : Result = o1; break;
                    case '-' : Result = (-1.0) * o1;
                  }
               push_stack (Result);
               break;
      case  7 : /* A → +     */
               push_adicao ('+');
               break;
      case  8 : /* A → -     */
               push_adicao ('-');
               break;
      case  9 : /* M → *     */
               push_multiplicacao ('*');
               break;
      case 10 : /* M → :     */
               push_multiplicacao (':');
               break;
      case 11 : /* U → +     */
               push_unario ('+');
               break;
      case 12 : /* U → -     */
               push_unario ('-');
               break;
      case AC : Result = pop_stack ();
               printf ("\nValor da Expressao: %f", Result);
   }
}

accao_inicial ()
{
  p = 0;
  init_stackestados ();
  init_stack ();
  tipodeunario [0] =
  tipodemultiplicacao [0] =
  tipodeadicao [0] = 0;
}
```

```
/*****************************************************************************/
/**                                                                         **/
/**                                  PARSER                                 **/
/**                                                                         **/
/*****************************************************************************/

int stackestados [100];

init_stackestados (){
   stackestados [0] = 0;
}

int topo_stackestados (){
  return (stackestados [stackestados [0]]);
}

int pop_stackestados (){
  stackestados [0] = stackestados [0] - 1;
  return (stackestados [stackestados[0]+1]);
}

push_stackestados (int X){
  stackestados [0] = stackestados [0] + 1;
  stackestados [stackestados[0]] = X;
}

int tipodeaccao (int acc){
  int tipo;

  if ((acc >= 0) && (acc < ESTADOS))
      tipo = TRANSICAO;
  else if ((acc >= ESTADOS) && (acc <= 42))
      tipo = REDUCAO;
  else if (acc == AC)
      tipo = ACEITACAO;
  else
      tipo = ERRO;

  return (tipo);
}

parser(){
  int s,x,accao,tipo,X,i;
  int producao;
  int SimbEsq, estado;

  accao_inicial ();
  push_stackestados (0);
  x = da_simbolo ();
  do { s = topo_stackestados ();
```

```
        accao = TabelaT [s][x];
        tipo = tipodeaccao (accao);
        switch (tipo)
          { case TRANSICAO : push_stackestados (accao);
                             x = da_simbolo ();
                             break;
            case REDUCAO   : producao = PRODUCAO(accao);
                             for (i=0; (i<PRODUCOES [producao][0]); i++)
                                 pop_stackestados ();
                             SimbEsq = PRODUCOES [producao][1];
                             estado  = topo_stackestados ();
                             push_stackestados (TabelaNT [estado][SimbEsq]);
                             executa_accao (producao);
                             break;
            case ACEITACAO : executa_accao (AC); break;
            case ERRO      : printf ("\n\nErro sintatico \n");
                             printf ("%s\n", gama);
                             for (i=1;(i<p);i++) printf (" ");
                             printf ("\^\n");
          }
      }
  while ((tipo != ERRO) && (tipo != ACEITACAO));
}

main (){
  printf ("\nExpressao : "); gets (gama);
  while (gama [0])
     { parser ();
       printf ("\nExpressao : ");
       gets (gama);
     }
}
```

6.4 Geradores Automáticos de Tradutores

Neste ponto ilustra-se o uso de **lex** e **yacc** na resolução do problema proposto.

A versão para LEX/YACC que a seguir se apresenta segue a mesma gramática e a mesma filosofia que a anteriormente analisada. No final deste secção será apresentada uma versão bastante mais simples.

```
-----------------------------------------------------------------------
(lex) : geracao da funcao da-simbolo
-----------------------------------------------------------------------
iss              ([0-9]+)
   double atof();
%%
{iss}("."{iss})?    {yylval.val=atof(yytext);
                     return('i');}
[()*+/-]             return(yytext[0]);
.                   ;
\n                   return('$');
```

```
%%
-------------------------------------------------------------------
(yacc)
-------------------------------------------------------------------
%{ typedef struct scont             /* valor associado a Econt e Pcont */
            { char op;              /*           (op,val)              */
              float val; }cont;
%}
%union {cont con;
        float val;
        char  c;}
%type <val> F E P 'i'
%type <c> opadt opmul
%type <con> Econt Pcont

%%
Z       : E '$'    {printf("%f\n",$1);return;}
        ;
E       : P Econt  {$$=calc($1,$2->op,$2->val);}
        ;
Econt   :          {mkcont($$,'+',0.0);}
        | opadt E  {mkcont($$,$1,$2);}
        ;
P       : F Pcont  {$$= calc($1,$2->op,$2->val);}
        ;
Pcont   :          {mkcont($$,'*',1.0);}
        | opmul P  {mkcont($$,$1,$2);}
        ;
F       : '(' E ')'     {$$=$2;}
        | 'i'           {$$=$1;}
        | opadt F       {$$=($1=='-')?-$2:$2;}
        ;
opadt   : '+'      {$$='+';}
        | '-'      {$$='-';}
        ;
opmul   : '*'      {$$='*';}
        | '/'      {$$='/';}
        ;
%%
#include <ctype.h>
#include <stdio.h>
#include <malloc.h>
#include <string.h>

#define actual_in(s) (strchr(s,actual) != NULL)

#define mkcont(cont,a,b) cont.op=a;cont.val=b

float calc(v1,op,v2)            /* calcula operacao binaria    */
float v1,v2;
char op;
{ switch(op)
```

```
 { case '+': return(v1+v2);
   case '-': return(v1-v2);
   case '*': return(v1*v2);
   case '/': return(v1/v2); } }

main(){
  while(1){
    printf("?\n");
    yyparse(); }
}

#include "lex.yy.c"
-----------------------------------------------------------------------
```

Segue-se uma outra versão do mesmo problema mas em que a gramática foi escrita de modo ambiguo e acrescentar informação acerca de associatividade de operadores e prioridades de operadores.

No gerador utilizado (YACC) as prioridades são descritas através da ordem de declaração dos operadores nas clausulas simultaneamente definem a associatividade dos operadores:

```
                                prioridade minima
  %right '='
  %left '+' '-'
  %left '*' '/'
  %left opuna
                                prioridade maxima
```

Nesta versão, foram introduzidas as seguintes extensões:

1. . - resultado da expressão anterior
2. <letraMinuscula> - variável temporária à qual podemos atribuir (=) o valor de expressões.

```
-----------------------------------------------------------------------
(yacc)
-----------------------------------------------------------------------
%{   double variavel[27],
            ultimo;
#define reg(x) variavel[x-'a']
%}

%union {double val;
        char   var;}

%token <val>  REAL
%token <var>  VAR
%type  <val>  E

%right '='
%left '+' '-'
%left '*' '/'
%left opuna

%%
Z      : E '$'  {printf("%f\n",$1);ultimo=$1;}   Z
```

```
       |
       ;
E      : E '+' E               {$$=$1+$3;}
       | E '-' E               {$$=$1-$3;}
       | E '/' E               {$$=$1/$3;}
       | E '*' E               {$$=$1*$3;}
       | '(' E ')'             {$$=$2;}
       | REAL                  {$$=$1;}
       | '+' E  %prec opuna    {$$=$2;}
       | '-' E  %prec opuna    {$$=-$2;}
       | '.'                   {$$=ultimo;}
       | VAR                   {$$=reg($1);}
       | VAR '=' E             {$$=reg($1)= $3;}
       ;
%%
#include <stdio.h>

main(){
 printf("?\n");
 yyparse();
}

#include "lex.yy.c"
```

(lex)

```
iss              ([0-9]+)
variavel          [a-z]
                        double atof();
%%
{iss}("."{iss})?    {yylval.val=atof(yytext);
                     return(REAL);}
{variavel}          {yylval.var=yytext[0];
                     return(VAR);}
[()*+/=.-]           return(yytext[0]);
\n                   return('$');
.                   ;
%%
```
