7. Manipulação de uma árvore geneológica a partir de um ficheiro de comandos - exemplo reduzido

Especificação reduzida:

Um programa reconhece uma linguagem de comandos. A linguagem especifica acções de manipulação de uma árvore geneológica de varões.

Linguagem:

O primeiro comando é o comando de inserção e tem a forma:

INSERIR-ANCESTRAL 'nome'

Podem seguir-se comandos de nascimento

NASCEU 'nome' FILHO-DE 'nome'

ou consulta:

PAI-DE? 'nome'

O último comando é necessáriamente FIM.

A disposição dos comandos no ficheiro input não tem regras de formatação. Apenas é relevante a ordem de apresentação.

Este exemplo também recomenda 2 gramáticas. A gramática de baixo nível e outra de alto nível.

Um espaço ou o fim de linha funcionam como separadores. Qualquer átomo da linguagem não pode conter separadores e quaisquer 2 átomos têm de ser separados por pelo menos 1 separador.

Definições relevantes para o tratamento:

Linguagens:

Alto nível ou nível identificado por 1:

input
<input-reconhecido> ≡

inserir apelido ((nascer apelido filho apelido)+ (pai apelido))* fim

onde os símbolos terminais são definidos por:

Baixo nível ou nível identificado por 2:

inserir ≡ INSERIR-ANCESTRAL

apelido ≡ "'" (caracter ≠ de "'")* "'"

nascer ≡ NASCEU

filho ≡ FILHO-DE

pai ≡ PAI-DE?

fim ≡ FIM

b) Algoritmos

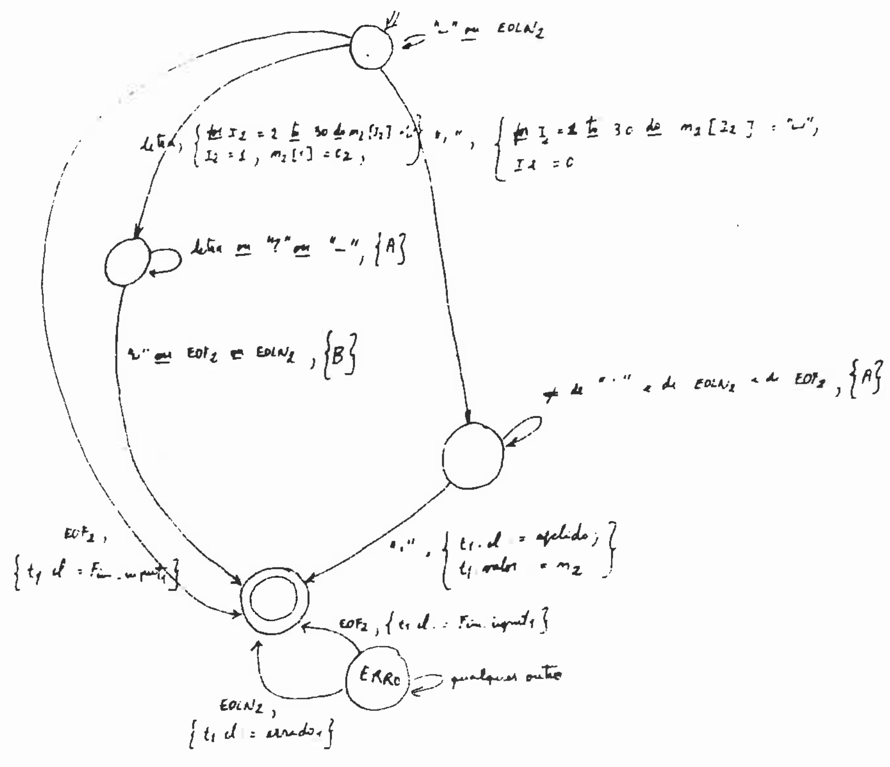
```
var numero-de-linha; (*inicializada globalmente a 1 *)
type nome = paked array [1 .. 30] of char;
     conjunto-classes-1 = (inserir, nascer, filho,ape-
               lido, pai, fim, Fim-input-1, errado-1);
     conjunto-terminal-1 = Record
                             cl: conjunto-classes-1;
                             valor: nome
                           end;
     procedure ler-terminal-1 (var t1: conjunto-termi
                                                nal-1);
        (* sempre que é desencadeada a sua execução
           despreza os separadores presentes no input,
           em seguida lê caracteres até compor um ter
           minal do nível 1. Se o ficheiro terminou
           devolve t1.cl = Fim-input1. Se o terminal
           é um nome, compõe o seu valor em t1.valor
           com espaços à direita e devolve t1.cl=apelido.
```

```
                    Se surge um erro assinala o erro e em se
                    guida despreza todos os caracteres do in
                    put até resincronizar com o fim da linha.
                    Após a resincronização termina a execu-
                    ção mas devolve t1.cl = errado1 ou t1.cl=
                    = Fim-input1 se no entretanto o ficheiro
                    acabou durante o processo da recuperação*)

     const EOLN2 = chr (13); (*código interno de carriage
                                                   return*)
           EOF2 =  chr (11); (*código interno não usado*)
           TAB2 =  chr (8)  ; (*código interno de TAB*)
      type conjunto-terminal2 = char (*acrescido de EOLN2
                                                 e EOF2*);
  procedure ler-terminal2 (var C: char);
     begin
         if eoln (input) then begin
                                   C: = EOLN2;
                                   readln;
                                   numero-de-linha:= nu
                                       mero-de-linha+1
                               end
         else if eof (input) then C: = EOF2

         else begin
                read (C);

                if C = TAB2 then C: = '␣' (*uniformiza
                                          ção de separadores*)
             end
     end (* ler-terminal *);
  var n2: nome; I2: integer; t2: char; (*terminal2*)
  begin (*algoritmo segundo o automato que se apresenta a
                                                    seguir*)
  end (* ler-terminal1 *);
```




```
A ≡  if I2 ≤ 29 then
       begin
           I2 := I2 + 1,
           m2[I2] := c2
       end
     else
       begin
           assinalar.erro( );
           (* sai por erro *)
       end;
```

```
B ≡  if m2 = 'INSERIR.ANCESTRAL' then
            t1 cl := inserir
     else if  m2 = 'FIM           ' then
            t1 cl := fim
     else if
            :
     else begin
            assinalar.erro( )
            (* sai por erro *)
          end;
```

Repare-se que a técnica de tratamento de erros é sempre entrar em erro sem avançar o input. Erro despreza toda a linha corrente. A sintaxe também é aproveitada para tratar erros semânticos,isto é:

mais de 30 caracteres num atomo
palavra chave desconhecida.

Os erros de baixo nível são assinalados a este nível. A variável global número de linha, serve para tornar as mensagens de erro mais explícitas.

O facto de que se deu um erro, tem também de ser assinalado ao automato superior para que este possa ressincronizar a sua acção sem assinalar novo erro.

A árvore é manipulada pelas acções semanticas:

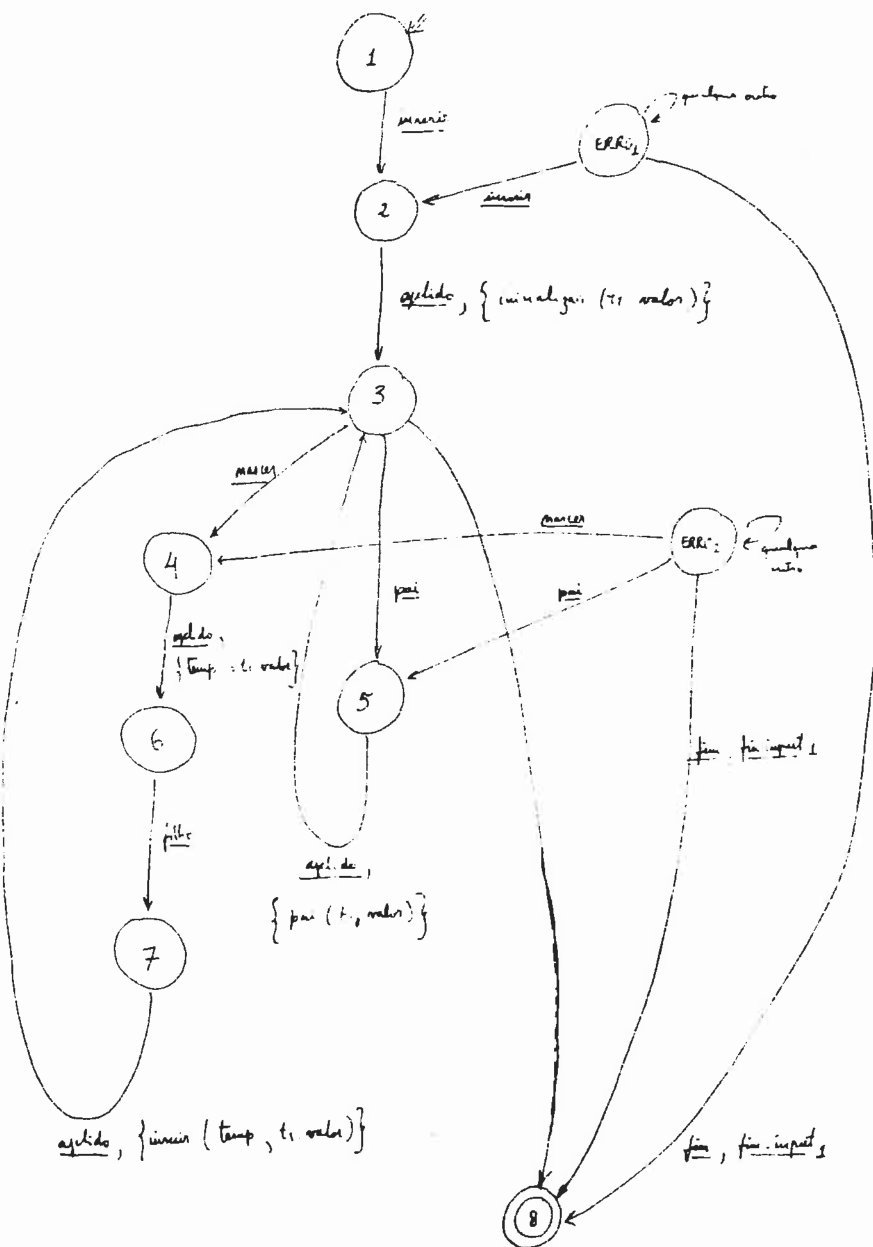
inicializar (n: nome); (*cria a árvore com o ancestral na raiz*)

inserir (n_1, n_2: nome); (*insere n_1 como filho de n_2 *)

pai (n_1: nome); (* escreve no output: o pai de n_1 é: ... *)

Com estas acções semanticas o automato do nível 1 é:

var t1 : conjunto_terminal1 ; Temp : nome .



ERRO$_1$ é o estado de recuperação de erros invocado nos estados 1 ou 2 sempre que no input surja algum terminal ilegal (incluindo errado), o seu conjunto de resincronização é:

fim, inserir , único dos comandos que permitem continuar o processamento.

ERRO$_2$ é o estado de recuperação de erros invocado em qualquer um dos outros estados, o seu conjunto de resincronização é:

pai, nascer, fim , início dos estados que permitem continuar o processamento.

Este exemplo mostra que o início dos comandos é o ponto vital para a resincronização. É por esta razão que uma boa solução para apoiar o tratamento de erros é incluir um terminal especial que fecha todos os comandos! Este terminal é o único elemento de todos os conjuntos de resincronização e o estado de erro pode ser único. O facto de se tentar manipular a árvore sem a inicializar passaria a ser um erro assinalado pela semantica.

8. Manipulação da árvore a partir de comandos dados interactivamente.

Este exemplo é muito semelhante ao anterior com a diferença que os terminais do nível 2 são lidos do terminal da consola do computador. O terminal do nível 1, Fim-input-1, desaparece pois num programa interactivo não há fim a não ser que o utilizador o peça explicitamen-

te. A técnica de tratamento de erros pode ser a sugerida no exemplo anterior, transformando o fim da linha numa terminação explicita de cada comando. O fim de linha terá assim, de ser um terminal explicito da gramática de alto nível. Uma função semantica o prompting.

9. Exercícios

Para todos os problemas propostos em 2 ou 3, nos quais se pedia para obter a gramática que gerava a linguagem de todas as sessões de trabalho ao terminal ou um input para um programa, trabalhe o problema até ao fim, segundo a forma especificada neste capítulo, se a linguagem envolvida for regular.

Um método expedito de verificar se se está em presença de uma linguagem regular, quando a gramática obtida não o fôr, consiste em tentar obter uma expressão regular que denote a mesma linguagem que a gramática gera.

Isso pode ser feito, na grande maioria dos casos em que a linguagem é regular, introduzindo o operador* para substítuir as produções recursivas e substituindo sucessivamente as não-terminais cujas produções apenas têm terminais na parte direita, por aqueles terminais nas produções onde figuram. Se no fim obter uma expressão apenas com terminais, então está provado que a linguagem é regular. No caso sontrário, nada está provado.

10. Chamada de atenção para algumas faltas de rigor introduzidas atrás.

Em todos os exemplos anteriores, sempre que se usam vários autómatos em hierárquia, a estrutura do autómato de baixo nível é deduzida de uma forma informal. No exemplo do tele-carregador é dito que a linguagem reco-

nhecida é denotada por:

$$\alpha(A + B + F + \alpha) + \overline{qq} \text{ outro} \neq \alpha$$

quando na verdade a linguagem presente à entrada do micro-computador é denotada por:

$$(\alpha(A + B + F + \alpha) + \overline{qq} \text{ outro} \neq \alpha)^*$$

Se introduzissemos nesta expressão regular a função semantica: passagem de um atomo elementar reconhecido ao autómato de alto nível:

$$(\alpha(A \{\text{passar " A"}\} + B \{\text{passar " B"}\} + F \{\text{passar " F"}\} + \alpha\{\text{passar "}\alpha\text{"}\}) + \overline{qq} \text{ outro} \neq \alpha \{\text{passar "outro"}\})^*$$

Obtinhamos um novo autómato cujo estado inicial seria de aceitação e que uma vez executada uma das acções semânticas regressaria ao estado inicial. Para implementar esse autómato, assim como a forma de passagem do átomo colectado ao autómato de alto nível teriam de ser usados métodos diferentes dos apresentados.

Dado que quando este automato executa a acção semântica ainda não leu um terminal pertencente ao átomo seguinte é possível implementá-lo como foi apresentado anteriormente, sendo o efeito da operação * substituído pela sua reactivação por uma nova chamada do procedimento ler-terminal pelo autómato de alto nível.

No outro exemplo, a manipulação da árvore geneológica, o problema surge de uma forma mais subtil.

É dito que a linguagem de baixo nível é denotada por:

"letra" ("letra" + "-" + "?")* + "'" caracter* "'" + EOF

Na verdade a linguagem significativa para o autómato de baixo nível é:

("separador" $^+$ "letra" ("letra" + "-" + "?")* + "separador" $^+$ "'" caracter* "'")* EOF

Onde o símbolo + em expoente significa 1 ou mais vezes.

As expressões "separador"$^+$ são introduzidas de uma forma forçada. Por exemplo PAI-DE?'ANTÔNIO SOUZA' conduz a um erro segundo aquela expressão, quando na verdade não o deveria. Se os separadores fossem opcionais aconteceria que não era possível simular o reconhecimento desta linguagem pelo autômato introduzido, pois quando a seguir ao "?" se encontrasse a "'" ter-se-ia de executar a acção semântica de envio de <u>pai</u> mas o estado seguinte não poderia ser o estado inicial, pois, este avançaria de novo sobre o input e desprezaria o símbolo "'" o que conduziria a um erro fatal. É a presença obrigatória do separador que permite simular o reconhecimento dos átomos pelo autômato apresentado, pois, quando este recomeça a análise são os separadores que são desprezados.

No caso de uma implementação interactiva, o separador "carriage return" é também um terminal significativo para o autômato de alto nível e o problema é ainda mais crítico.

Na prática este problema resolve-se tornando a variável t (terminal corrente) uma variável global que no início de todo o processo é inicializada com '␣' e o autômato de baixo nível deixa de avançar o input antes de entrar no estado inicial.

No entanto, se esta solução é possível na prática ela invalida o método atrás exposto, enquanto método totalmente suportado e justificado teòricamente. A solução deste problema tem de ser encontrada num quadro diferente do introduzido nestes 2 exemplos.

6. AUTÓMATOS DE PILHA

1. Definição

Um autómato de pilha (AP) M é um sistema

$$M = (Q, T, W, \delta, q_0, z_0, F),$$

em que

Q é um conjunto finito de estados

T é um alfabeto de entrada

W é o alfabeto de pilha

$q_0 \in Q$ é o estado inicial

$z_0 \in W$ é o símbolo da base da pilha

$F \subset Q$ é o conjunto dos estados finais

δ é a função transição de configuração, que aplica

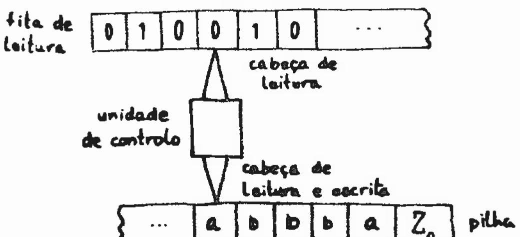
$$Q \times T_\lambda \times W \quad \text{em} \quad P_f(Q \times W^*).$$

NOTAÇÃO. 1) T_λ representa o conjunto $T \cup \{\lambda\}$.

2) $P_f(x)$ é o conjunto dos subconjuntos finitos de x

2. Interpretação

A descrição física é semelhante à do AF, só que existe mais uma fita - a "pilha" - onde a unidade de controlo lê e escreve símbolos. Além disso, o "funcionamento" do AP é não-determinista.



Inicialmente:

1) Uma palavra $x \in T^*$ está escrita na fita de leitura, e a cabeça de leitura está colocada sobre o 1º símbolo de x (o mais à esquerda);

2) A unidade de controlo está no estado inicial q_0;

3) A pilha contém unicamente o símbolo Z_0 na sua base (símbolo mais à direita) e a cabeça de leitura e escrita está colocada sobre Z_0.

A partir daqui, o AP começa a funcionar:
Em certa etapa do processo:
1) A cabeça de leitura está sobre uma célula da fita de leitura contendo um símbolo $a \in T$ que ocorre na palavra x;

2) A unidade de controlo está num estado q;

3) Na pilha está inscrita uma palavra de W^*, podendo-se dar dois casos:

a) essa palavra é a palavra vazia λ;

b) a palavra é não-vazia, e o primeiro símbolo (Z, digamos) está a ser inspecionado pela cabeça de leitura e escrita. Podemos designar essa palavra por $Z\gamma$, com $Z \in W$ e $\gamma \in W^*$ (no início do processo, $\gamma = \lambda$ e $Z=Z_0$).

As alíneas 2 e 3 desta "descrição instantânea" cristalizam-se na definição de "configuração".

Uma configuração de um AP é um par $(q,\gamma) \in Q \times W^*$.

O funcionamento de um AP vai consistir numa sucessão de transições de configurações, partindo da configuração inicial (q_0, Z_0).

Essas transições vão ser feitas não-deterministicamente, comandadas pela função δ , que como vimos se chama "transição de configuração".

Transição de configuração:

Consideremos a descrição instantânea anterior.

1) Se a palavra inscrita na pilha for λ (i.e., se a

configuração for (q,λ)) não há transição possível;

2) Caso contrário, a transição vai depender dos símbolos q, Z, podendo o AP "escolher" entre ignorar ou não o símbolo a (esta possibilidade de escolha traduz o 1º aspecto não-determinista do AP). Esta possibilidade de escolha existe visto que tanto (q, λ, Z) ("ignorar a") como (q, a, Z) ("não ignorar a") pertencem a $QxT_\lambda xW$, que é o domínio de δ.

a) Ignorando a, suponhamos que:

$$\delta(q, \lambda, Z) = \{(q_1, \gamma_1), \ldots, (q_r, \gamma_r)\} \subset P_f(QxW^*)$$

Então o AP pode escolher qualquer um dos pares acima enumerados. Se escolheu (q_i, γ_i), transita para a configuração $(q_i, \gamma_i\gamma)$ e <u>mantém</u> a cabeça de leitura sobre o símbolo a (o novo estado da unidade de controlo é q_i, e a cabeça de leitura e escrita apagou Z e escreveu γ_i, ficando a inspeccionar o primeiro símbolo de $\gamma_i\gamma$; se $\gamma_i=\lambda$, esta acção reduz-se a apagar Z e inspeccionar o 1º símbolo de γ). (Se $\delta(q, \lambda, Z) = \emptyset$, não há transição possível ignorando a.)

b) Tendo a em consideração, se

$$\delta(q, a, Z) = \{(q'_1, \gamma'_1), \ldots, (q'_s, \gamma'_s)\} \subset P(QxW^*),$$

tudo se passa como anteriormente, com a excepção que a cabeça de leitura avança para o próximo símbolo de x (se existir; se não, para a próxima célula da fita, que está vazia). Assim, se a escolha recair sobre (q'_i, γ'_i), a próxima configuração será $(q'_i, \gamma'_i\gamma)$. (Se $\delta(q, a, Z) = \emptyset$, não há transição possível tendo a em consideração).

O 2º aspecto do não-determinismo de um AP traduz-se na possibilidade das escolhas a fazer em a) e b).

A formalização desta noção de transição de configuração é a seguinte:

Escrevemos

$$(q,\ Z\gamma) \vdash_X (q',\gamma'\gamma)$$

se

$$(q,\ Z\gamma) \in Q \times W^+$$

$X \in T_\lambda$ (i.e., ou $X \in T$ ou $X = \lambda$)

$(q',\ \gamma') \in \delta(q,\ X,\ Z)$

A expressão

$$(q,\ Z\gamma) \vdash_X (q',\ \gamma'\gamma)$$

significa que o AP pode transitar da configuração $(q,\ Z\gamma)$ para a configuração $(q',\ \gamma'\gamma)$ tendo em conta X. Não há "obrigatoriedade" visto que o AP é não-determinista.

Após ter lido a palavra x:

O AP encontra-se numa configuração (q,γ). Como esta se obteve da configuração inicial $(q_0,\ Z_0)$ por meio de uma série de transições, escrevemos:

$$(q_0,\ Z_0) \vdash_x^* (q,\gamma).$$

Uma vez mais, isto significa que o AP pode transitar de $(q_0,\ Z_0)$ para (q,γ) após ter lido x.

Critérios de aceitação de palavras:

1º $x \in T^*$ é aceite se

$$(q_0,\ Z_0) \vdash_x^* (q,\gamma) \quad \text{com } \gamma = \lambda.$$

2º $x \in T^*$ é aceite se

$$(q_0,\ Z_0) \vdash_x^* (q,\gamma) \quad \text{com } q \in F.$$

No primeiro caso diz-se que x é aceite por pilha vazia.

No segundo, que x é aceite pelo critério dos estados finais.

Um AP funciona uniformemente por um ou outro dos critérios, isto é, não pode funcionar segundo o 1º critério para certas palavras e segundo o 2º critério para outras palavras.

A linguagem aceite por um AP M é L (M) = conjunto das palavras de T* aceites por M.

Prova-se que: para todo o AP M_1 reconhecendo palavras pelo critério da pilha vazia existe um AP M_2 reconhecendo palavras pelo critério dos estados finais tal que $L(M_2) = L(M_1)$, e inversamente.

Assim, não existe nenhuma razão teórica que favoreça em cada caso, a utilização de um ou outro destes 2 critérios.

EXEMPLO

$$M = (\{q_0, q_1\}, \{0, 1\}, \{z_0, 0, 1\}, \delta, q_0, z_0, \emptyset)$$

δ :

$(q_0, 0, z_0) \mapsto \{(q_0, 0z_0)\}$ $\quad (q_0, 1, 1) \mapsto \{(q_0, 11), (q_1, \lambda)\}$

$(q_0, 1, z_0) \mapsto \{(q_0, 1z_0)\}$ $\quad (q_1, 0, 0) \mapsto \{(q_1, \lambda)\}$

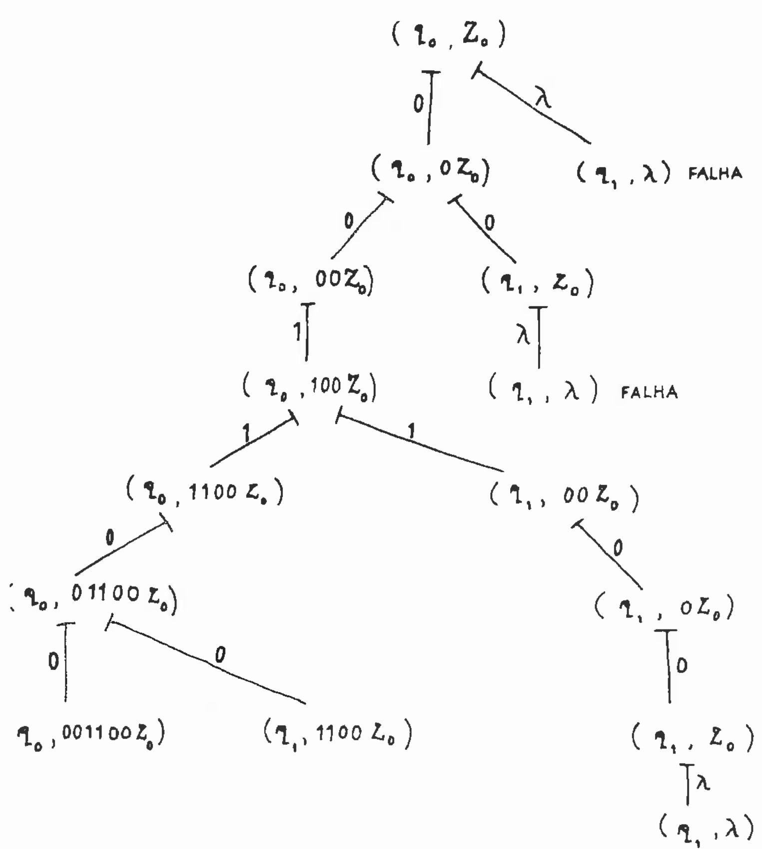
$(q_0, 0, 0) \mapsto \{(q_0, 00), (q_1, \lambda)\}$ $\quad (q_1, 1, 1) \mapsto \{(q_1, \lambda)\}$

$(q_0, 0, 1) \mapsto \{(q_0, 01)\}$ $\quad (q_0, , z_0) \mapsto \{(q_1, \lambda)\}$

$(q_0, 1, 0) \mapsto \{(q_0, 10)\}$ $\quad (q_1, , z_0) \mapsto \{(q_1, \lambda)\}$

Este AP aceita a linguagem $\{w. w^r : w \in \{0, 1\}^*\}$ pelo critério da pilha vazia (w^r é a palavra que se obtem de w invertindo a ordem dos símbolos de w; exemplo: $(0100011)^r=1100010$).

A palavra $001100=001(001)^r$ é aceite pelo AP. Vamos descrever o seu funcionamento por uma arborescência que contém todas as possibilidades de transição de configurações, partindo da configuração inicial (q_0, z_0).



Seja x = 001100.

Tem-se $(q_0, Z_0) \overset{*}{\underset{x}{\vdash}} (q_0, 001100\, Z_0)$, $(q_0, Z_0) \overset{*}{\underset{x}{\vdash}} (q_1, 1100\, Z_0)$ e $(q_0, Z_0) \overset{*}{\underset{x}{\vdash}} (q_1, \lambda)$. Em virtude desta última expressão, x é aceite pelo AP.

3. Autómatos de pilha e gramáticas algébricas

TEOREMA. Seja G = (N, T, S, P) uma C-gramática. Existe um AP M funcionando com o critério da pilha vazia tal que L (M) = L (G).

O AP é definido por:

$Q = \{q_0\}$ (um único estado)

$T = T$

$W = N \cup T$

$q_0 = q_0$

$Z_0 = S$

$F = \emptyset$

δ : 1) $\delta(q_0, a, A) = \emptyset \quad \forall a \in T \quad \forall A \in N$

2) $\delta(q_0, a, b) = \emptyset \quad \forall a \in T \quad \forall b \in T \quad b \neq a$

3) $\delta(q_0, a, a) = \{(q, \lambda)\} \; \forall a \in T$

4) $\delta(q_0, \lambda, a) = \emptyset \quad \forall a \in T$

5) $\delta(q_0, \lambda, A) = \{(q_0, \alpha) : A \rightarrow \alpha \in P\}$ □

EXEMPLO

Gramática

$G_0 = (\{+, *, (,), a\}, \{E, T, F\}, E, P)$

$P: E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \quad a$

AP

$M_0 = (\{q_0\}, \{+, *, (,), a\}, \{+, *, (,), a, E, T, F\}, \delta, q_0, E, \emptyset)$

$\delta(q_0, b, b) = \{(q_0, \lambda)\}$ para $b \in \{+, *, (,), a\}$

$\delta(q_0, \lambda, E) = \{(q_0, E+T), (q, T)\}$

$\delta(q_0, \lambda, T) = \{(q_0, T*F), (q_0, F)\}$

$\delta(q_0, \lambda, F) = \{(q_0, (E)), (q_0, a)\}$

δ (qualquer outra coisa) $= \emptyset$.

7. ANÁLISE DE GRAMÁTICAS LL(1)

1. Introdução

Neste capítulo apresenta-se uma classe de gramáticas "context free" - as gramáticas LL(1) - para as quais o problema da análise sintáctica descendente é particularmente simples, e tem as seguintes características:

O algorítmo de análise é determinista, no sentido em que em cada passo da análise existe (e é conhecida) uma única acção possível a executar.

Além disso, a escolha dessa acção é determinada pelo primeiro símbolo da porção do texto fonte que ainda falta analisar. (Um só símbolo de "lookahead").

Por outro lado, o algorítmo termina sempre, quer aceitando o texto fonte, quer assinalando erros.

Este capítulo trata sucessivamente de:

- definição das gramáticas LL(1);
- apresentação de algorítmos que verificam se uma gramática dada é LL(1);
- programação de analisadores para gramáticas LL(1).

2. Definição das Gramáticas LL (1)

Em todo este capítulo G=(N,T,S,P) é uma gramática "context free" que se supõe estar limpa. Recordemos que isto significa que:

- todo o símbolo não-terminal A é <u>produtivo</u>, isto é, gera uma linguagem $L(A) \neq \emptyset$;

- todo o símbolo $X \in N \cup T$, terminal ou não, é <u>acessível</u> a partir do axioma S, isto é, existem palavras $\alpha, \beta \in (N \cup T)^*$ tais que $S \Rightarrow^* \alpha X \beta$.

Para maior comodidade poremos ainda $V = N \cup T$.

A. <u>Definição do Conjunto dos Primeiros</u>

Dada uma gramática em que a regra do axioma seja,por exemplo,

$$S \rightarrow a\alpha \mid b\beta$$

em que $a,b \in T$ e $\alpha, \beta \in V^*$, toda a palavra gerada por S começa quer por 'a' quer por 'b'. Dizemos então que o conjunto dos <u>primeiros</u> de S é $\{a,b\}$ e escrevemos

$$\text{prim } (S) = \{a,b\}.$$

Mais geralmente, podemos (e é importante fazê-lo!) definir o conjunto prim (γ) para qualquer palavra $\gamma \in V^*$

Por exemplo, para as duas expansões de S temos

$$\text{prim } (a\alpha) = \{a\} , \qquad \text{prim } (b\beta) = \{b\},$$

visto que toda a palavra gerada a partir de $a\alpha$ começa necessàriamente por 'a', e toda a palavra gerada a partir de $b\beta$ começa por 'b'. A importância de se conhecerem estes conjuntos na análise sintáctica pode também ser ilustrada com o fragmento da gramática anterior. Se se pretendesse analisar uma palavra da forma $a\gamma$ a partir de S, teria de se utilizar a primeira expansão de S, visto que prim $(a\gamma)$ = prim $(a\alpha) = \{a\}$, e prosseguia-se com a análise analisando agora γ a partir de α. Se a palavra a analisar tivesse a forma $b\delta$ então utilizava-se a segunda expansão de S. Se não tivesse nenhuma destas duas formas, isto é, se o 1º símbolo da palavra a analisar não começasse por 'a' nem por 'b', teria de ser assinalado um erro sintáctico no texto fonte.

Suponhamos agora que a regra de S tinha a forma

$$S \rightarrow \alpha | \beta$$

em que prim $(\alpha)=\{a,b\}$, prim $(\beta)=\{b,c\}$. Se se pretendesse analisar a palavra $b\gamma$, o conhecimento apenas do 1º símbolo de $b\gamma$, que é 'b', não permitiria decidir entre as duas expansões de S, visto que $b \in$ prim(α) e $b \in$ prim(β), isto é em princípio $b\gamma$ tanto podia ter sido gerada a partir de α como a partir de β. Por esta razão, quando se tem uma regra da forma

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

exige-se que se $i \neq j$ então prim $(\alpha_i) \cap$ prim $(\alpha_j)=\emptyset$. Esta condição, que é obviamente necessária, não é, contudo, suficiente, e será generalizada mais adiante. Mas por agora contentemo-nos em definir rigorosamente prim (α) para toda a palavra $\alpha \in V^*$:

$$\text{prim } (\alpha)=\{a \in T: (\exists \beta \in V^*) \ \alpha \Rightarrow^* a\beta\}.$$

Assim, prim (α) é o conjunto de todos os símbolos terminais que são os 1ºs símbolos das palavras geradas por α.

A determinação de prim (α), embora não seja conceptualmente complicada, não é tão simples como o poderá ter dado a entender o fragmento de gramática anterior.

Um algoritmo será apresentado no parágrafo seguinte. Aqui, vamos extrair algumas consequências da definição, para a esclarecer melhor.

Comecemos por definir uma função booleana lambda (α), para todo o $\alpha \in V^*$, do seguinte modo:

$$\text{lambda } (\alpha) = \begin{cases} \underline{\text{true}} \text{ se } & \alpha \Rightarrow^* \lambda \\ \\ \underline{\text{false}} \text{ caso contrário.} \end{cases}$$

Como consequências simples desta definição temos:

(1) lambda (λ) = true

(2) lambda $(\alpha a \beta)$ = false, em que $a \in T$, $\alpha, \beta \in V^*$.

Em particular, lambda (a)=false para todo o $a \in T$.

(3) lambda $(\alpha\beta)$=lambda (α) and lambda (β), em que $\alpha, \beta \in V^*$.

(4) em virtude de (1,2,3), lambda (α) fica conhecido para todo o $\alpha \in V^*$ se se conhecer lambda (A) para todo o $A \in N$. No parágrafo seguinte será apresentado um algorítmo que faz esta determinação.

Voltando agora à definição de prim (α), podemos extrair dela algumas consequências:

(1) prim $(\lambda)=\emptyset$.

(2) prim $(a\alpha)=\{a\}$, em que $a \in T$ e $\alpha \in V^*$. Em particular prim $(a)=\{a\}$ para todo o $a \in T$.

(3) prim $(\alpha\beta)$= prim $(\alpha) \cup$ if lambda (α) then prim (β) else $\emptyset$, em que $\alpha, \beta \in V^*$. Como efeito, se lambda (α)= true então $\alpha \Rightarrow^* \lambda$, e há palavras que se geram a partir de β por derivações da forma

$$\alpha\beta \Rightarrow^* \beta \Rightarrow^* a\gamma.$$

Neste caso, $a \in$ prim (β) e $a \in$ prim $(\alpha\beta)$, logo é necessário incluir prim (β) em prim $(\alpha\beta)$. Consideremos por exemplo a gramática

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow a \mid \lambda \\ B &\rightarrow b \end{aligned}$$

Tem-se prim $(A)=\{a\}$, prim $(B)=\{b\}$, e, como lambda (A) = true, prim (S)= prim (AB)= prim $(A) \cup$ prim $(B)=\{a,b\}$. Isto aliás era evidente, visto que $L(S)=\{ab,b\}$.

(4) Tal como no caso de lambda, em virtude de (1,2,3), prim (α) fica conhecido para todo o $\alpha \in V^*$ se se conhecer prim (A) para todo o $A \in N$. No parágrafo seguinte apresenta-se um algorítmo que determina os primeiros de qualquer símbolo não terminal.

5. Definição do Conjunto dos seguintes

Consideremos a seguinte gramática de axioma S:

$S \rightarrow AB$

$A \rightarrow a|\lambda$

$B \rightarrow a$

A única regra com mais de 1 expansão é a regra de A, e tem-se prim (a)∩prim (λ)= {a}∩∅=∅. Tentemos, porém, analisar a palavra 'aa':

OBJECTIVO	TEXTO FONTE
S	aa
AB	aa
?	

Se o analisador tiver acesso apenas ao 1º símbolo do texto fonte, do seu ponto de vista o texto fonte tanto pode ser 'aa' como 'a'. No 1º caso, $A \rightarrow a$ é a produção que deve ser escolhida para expandir A (em AB), enquanto que a produção apropriada ao 2º caso é $A \rightarrow \lambda$. Não é pois possível decidir entre as duas acções a tomar, e o tipo de gramáticas que permite que tais situações ocorram tem de ser eliminado para que o nosso analisador funcione.

Para vermos o que há de errado com a gramática anterior, suponhamos que no objectivo AB "congelávamos" temporáriamente a expansão de A. A única expansão possível para B levaria ao novo objectivo Aa. Vê-se assim que o símbolo 'a', que é um primeiro de A, também é um "seguinte" de A, isto é, pode ocorrer a seguir a A numa palavra que deriva do axioma S. Assim, quando o 1º símbolo do texto fonte é 'a', não se sabe em princípio se ele está ali no papel de primeiro de A ou de seguinte de A, visto que A pode gerar λ .

A definição do conjunto dos <u>seguintes</u> de um símbolo não terminal A (é o único caso em que interessa definir os seguintes) é a que segue:

$$\text{seg }(A)=\{a\in T:(\exists\ \alpha,\beta\in V^*)\quad S\Rightarrow^*\alpha A a\beta\}.$$

Assim, como ficou dito acima, os seguintes de A são todos os símbolos terminais que podem ocorrer imediatamente a seguir a A numa palavra que deriva do axioma. Em face do exemplo anterior, não é difícil concluir que quando se tem uma regra da forma:

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

em que um certo α_i pode gerar a palavra vazia ($\alpha_i\Rightarrow^*\lambda$), se tem de exigir que seg $(A)\cap$ prim $(\alpha_j)=\emptyset$ para todo o $j=1,2,\dots,n$. Se notarmos que prim $(A)=$ prim $(\alpha_1)\cup$prim $(\alpha_2)\cup\dots\cup$prim (α_n), a exigência anterior é equivalente à de que seg$(A)\cap$prim $(A)=\emptyset$.

As duas condições que acabámos de ver, a que se refere aos primeiros e a que envolve os seguintes, podem ser reunidas numa só condição, que apresentaremos adiante.

Esta condição, embora necessária não é ainda suficiente para os nossos propósitos. Vamos ver agora como completá-la.

C. Gramáticas LL(1)

Recordemos que estamos a supor que $G=(N,T,S,P)$ é uma gramática "context free" limpa.

O conjunto dos símbolos directores de uma produção $A\rightarrow\alpha$ define-se do seguinte modo:

dir $(A,\alpha)=$ prim $(\alpha)\cup$ <u>if</u> lambda (α) <u>then</u> seg (A) <u>else</u> $\emptyset$.

Podemos agora definir gramática LL(1).

Uma gramática G é LL(1) se e só se, para toda a regra $A\rightarrow\alpha_1\,|\,\alpha_2\,|\,\dots\,|\,\alpha_n$ de G, **(1)** se $i\neq j$ então dir $(A,\alpha_i)\cap$dir $(A,\alpha_j)=\emptyset$; **(2)** se $\alpha_i\Rightarrow^*\lambda$ e $\alpha_j\Rightarrow^*\lambda$ então $i=j$.

A primeira destas condições não é mais do que a reunião numa só das condições anteriores relativas aos primeiros e aos seguintes. A segunda condição merece alguns comentários.

(19) Uma gramática diz-se <u>recursiva</u> se existir um símbolo não terminal A tal que $A \Rightarrow^{+} \alpha A \beta$ para alguns α, β V*. Uma gramática recursiva não é necessàriamente indesejável. Bem pelo contrário, pode demonstrar-se que se uma gramática G gerar uma linguagem infinita (que é o que sucede com maior frequência) então G é recursiva. Mas uma gramática que seja <u>recursiva à esquerda</u> já é indesejável do ponto de vista da análise sintáctica descendente: uma gramática é recursiva à esquerda se existir um símbolo não terminal A tal que $A \Rightarrow^{+} A\beta$ para algum $\beta \in V^*$.

Suponhamos que numa dada fase da análise de um certo texto fonte se chegava a uma situação:

OBJECTIVO	TEXTO FONTE
$A\alpha$	t

e que a gramática dada era recursiva à esquerda em A. Se se empregassem sucessivamente expansões que levassem de A a $A\beta$, chegava-se à situação:

$A\beta\alpha$	t

e o processo repetia-se indefinidamente, não havendo pois possibilidade de o algorítmo terminar.

Ora a primeira das condições definidoras das gramáticas LL(1) não garante só por si que a gramática não seja recursiva à esquerda. Consideremos, por exemplo, a gramática:

$$S \rightarrow A \mid B$$

$$A \rightarrow B \mid \lambda$$

$$B \rightarrow A \mid \lambda$$

Tem-se $L(S)=L(A)=L(B)=\lambda$, por conseguinte os conjun

tos directores são todos vazios, e a primeira condição verifica-se trivialmente. A gramática é, porém, recursiva à esquerda, visto que $A \Rightarrow B \Rightarrow A$. Logo a primeira condição só por si não elimina a possibilidade de existência de recursividade à esquerda. Note-se, todavia, que a gramática em exemplo não é LL(1), visto que não satisfaz a segunda condição. Pode aliás demonstrar-se a seguinte:

PROPOSIÇÃO. Uma gramática LL(1) não é recursiva à esquerda.

Demonstração. Seja G=(N,T,S,P) uma gramática LL(1) (limpa, recorde-se!) e comecemos por admitir que G é recursiva à esquerda, para em seguida chegar a uma contradição. É possível escolher um símbolo não terminal A recursivo à esquerda cuja regra tenha mais de 1 expansão.

Com efeito, se todos os símbolos não terminais recursivos à esquerda tivessem uma regra apenas com uma expansão então todos esses símbolos seriam improdutivos, contrariamente à hipótese de que a gramática é limpa. Em vez de demonstrarmos esta afirmação observemos a gramática:

$$S \longrightarrow XAa$$

$$X \longrightarrow \lambda$$

$$A \longrightarrow Y\,S\,s$$

$$Y \longrightarrow \lambda$$

É evidente que S e A são improdutivos, isto é, $L(S)=L(A)=\emptyset$.

Seja então:

$$A \longrightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \qquad (n > 1)$$

a regra de um símbolo não terminal recursivo à esquerda.

Uma vez que, por hipótese, $A \longrightarrow^{+} A\beta$, existe uma expansão de A, α_1 digamos, tal que $A \Rightarrow \alpha_1 \Rightarrow^{*} A\beta$. Como em particular, $A \Rightarrow \alpha_2$, podemos escrever:

$$A \Rightarrow \alpha_1 \Rightarrow^{*} A\beta \Rightarrow \alpha_2\beta\,.$$

Assim, vê-se que prim $(\alpha_2) \subset$ prim (α_1). Se prim $(\alpha_2) \neq \emptyset$, segue-se que dir $(A,\alpha_1) \cap$ dir $(A,\alpha_2) \neq \emptyset$, contrariamente à hipótese de G ser LL(1). Se prim $(\alpha_2) = \emptyset$, como G é limpa tem-se $L(\alpha_2) = \{\lambda\}$, donde em particular $\alpha_2 \Rightarrow * \lambda$. Não se pode ter prim ( ) $= \emptyset$, se não $\beta \Rightarrow * \lambda$ pela mesma razão anterior e portanto $\alpha_1 \Rightarrow * \alpha_2 \Rightarrow * \lambda$, contrariando a 2ª condição da definição da gramática LL(1). Logo prim $(\beta) \neq \emptyset$, prim $(\beta) \subset$ seg$(A) \subset$ dir(A,α_2) (porque $A \Rightarrow * A\beta$ e $\alpha_2 \Rightarrow * \lambda$) e prim $(\beta) \subset$ prim $(\alpha_1) \subset$ dir (A,α_1) (visto que $\alpha_1 \Rightarrow * \alpha_2\beta$ e $\alpha_2 \Rightarrow * \lambda$), por conseguinte dir $(A,\alpha_1) \cap$ dir $(A,\alpha_2) \neq \emptyset$. Uma vez mais se contradiz a hipótese de que a gramática dada é LL(1), e fica terminada a demonstração.

(2º) Passamos ao segundo comentário sobre a segunda condição da definição das gramáticas LL(1). Se esta condição é indispensável, como se viu, na "maior parte dos casos", se assim se pode dizer, ela é uma consequência da 1ª condição. Por exemplo, a gramática:

$$S \rightarrow A \mid B$$
$$A \rightarrow BX \mid \lambda$$
$$X \rightarrow x \mid \lambda$$
$$B \rightarrow A \mid \lambda$$

que não satisfaz a 2ª condição, já não satisfaz a primeira: vamos mostrar que para a regra $A \rightarrow BX \mid \lambda$ se tem $x \in$ dir$(A,BX) \cap$ dir(A,λ). Com efeito, $x \in$ dir(A,BX) porque $x \in$ prim(BX), visto que $BX \Rightarrow X \Rightarrow x$; por outro lado, $x \in$ dir(A,λ) porque $x \in$ seg(A), dado que $S \Rightarrow A \Rightarrow BX \Rightarrow AX \Rightarrow Ax$.

<u>Na prática, existem várias possibilidades de evitar ter de testar a 2ª condição.</u> A mais importante é porventura a de criar um novo símbolo terminal "eof" ("end of file"), um novo axioma T e uma nova produção T $\rightarrow$ S "eof", em que S é o axioma da gramática de que se parte. Sejam G a gramática inicial, G' a nova gramática.

Se G' satisfizer a condição dos conjuntos directores então G' já é LL(1), visto que a 2ª condição decorre automáticamente da primeira. Com efeito, consideremos uma regra:

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

e suponhamos que $\alpha_i \Rightarrow^* \lambda$ e $\alpha_j \Rightarrow^* \lambda$ com $i \neq j$ (para chegar a uma contradição). É fácil de ver que $seg_{G'}(A) \neq \emptyset$ em G'. Com efeito, se $seg_G(A) \neq \emptyset$ em G então $seg_{G'}(A) \neq \emptyset$ em G' visto que $seg_G(A) \subset seg_{G'}(A)$. (Isto resulta de que $S \overset{*}{\underset{G}{\Rightarrow}} \alpha$ se e só se $T \overset{*}{\underset{G'}{\Rightarrow}} \alpha$ "eof".) Se $seg_G(A) = \emptyset$ então $seg_{G'}(A) = \{$"eof"$\}$. Segue-se que

$$dir_{G'}(A, \alpha_i) \cap dir_{G'}(A, \alpha_j) \neq \emptyset$$

em qualquer dos casos, contrariando o facto de que G' satisfazia a condição dos conjuntos directores.

Podemos então concluir que G' é LL(1) se e só se satisfizer a condição dos conjuntos directores. Dado que para toda a produção $A \to \alpha$ de G se tem $dir_{G'}(A,\alpha) \subset dir_{G'}(A,\alpha)$, conclui-se que G é LL(1) se e só se G' for LL(1). Esta é uma das razões pelas quais é tão útil acrescentar a cada palavra gerada por G um terminador especial como "eof".

3. Algorítmos para a verificação das condições LL (1)

Neste parágrafo vamos apresentar os algorítmos prometidos no parágrafo anterior.

A. Símbolos não terminais que geram a palavra vazia

LAMBDA é um identificador inicializado com $\emptyset$ (conjunto vazio). Quando o algorítmo termina, LAMBDA contém todos os símbolos não terminais que geram a palavra vazia.

SIMB é um identificador auxiliar, também do tipo "conjunto de símbolos não terminais". O algorítmo é o seguinte:

(1) - Retirar da gramática todas as produções $A \to \alpha$ em que $\alpha \in V^* T V^*$, isto é, α contém pelo menos um símbolo terminal. (NOTA: Estas produções são irrelevantes para o problema em questão; só ficam produções das formas $A \to \lambda$ ou $A \to X_1 X_2 \ldots X_n$, em que $X_1, X_2, \ldots, X_n \in N$.)

(2) - Fazer LAMBDA:= SIMB:=$\emptyset$.

(3) - Seja SIMB:= conjunto dos símbolos não terminais A para os quais existe a produção $A \to \lambda$.

(4) - Se SIMB = $\emptyset$, FIM. Se não, passar a (5).

(5) - Fazer LAMBDA:= LAMBDA $\cup$ SIMB.

(6) - Eliminar as produções $A \rightarrow \alpha$ tais que $A \in$ SIMB.

(7) - Para cada uma das restantes produções $A \rightarrow X_1 X_2 \ldots X_n$, eliminar da parte direita todos os símbolos x_i que pertençam a SIMB.

(8) - Fazer SIMB:=$\emptyset$.

(9) - Repetir (3).

EXEMPLO. Seja a gramática

$$
\begin{aligned}
S &\rightarrow a\,S \mid AB \\
A &\rightarrow X\,a\,Y \mid X\,A \mid B\,Y \\
B &\rightarrow \lambda \mid S\,B \\
X &\rightarrow B\,B \mid x \\
Y &\rightarrow Y\,B \mid a
\end{aligned}
$$

Mostra-se a seguir o que acontece após a execução de cada um dos passos do algorítmo:

(1) A gramática transforma-se em:

$$
\begin{aligned}
S &\rightarrow A\,B \\
A &\rightarrow X\,A \mid B\,Y \\
B &\rightarrow \lambda \mid S\,B \\
X &\rightarrow B\,B
\end{aligned}
$$

(Note-se que Y ocorre na parte direita de uma produção, embora tenham sido eliminadas todas as produções de Y.)

(2) LAMBDA = SIMB = $\emptyset$.
(3) SIMB = $\{B\}$.
(4) SIMB $\neq \emptyset$.
(5) LAMBDA = $\{B\}$.
(6) A gramática fica:

$$
\begin{aligned}
S &\rightarrow A\,B \\
A &\rightarrow X\,A \mid B\,Y \\
X &\rightarrow B\,B
\end{aligned}
$$

(7) A gramática fica:

$$S \rightarrow A$$
$$A \rightarrow X\ A\ |\ Y$$
$$X \rightarrow \lambda$$

(8) SIMB = $\emptyset$

(3') SIMB = $\{X\}$

(4') SIMB $\neq \emptyset$

(5') LAMBDA = $\{B, X\}$

(6') A gramática fica:

$$S \rightarrow A$$
$$A \rightarrow X\ A\ |\ Y$$

(7') A gramática fica:

$$S \rightarrow A$$
$$A \rightarrow A\ |\ Y$$

(8') SIMB = $\emptyset$

(3'') SIMB = $\emptyset$

(4'') FIM.

Os símbolos não terminais que geram a palavra vazia são, por conseguinte, B e X, visto que o valor final de LAMBDA é $\{B, X\}$.

B. Determinação do conjunto dos primeiros de cada símbolo não terminal e teste da recursividade à esquerda

O algorítmo consiste em definir um grafo apropriado e em "ler" os resultados desejados do grafo.

Segue-se a construção do grafo.

(1) O conjunto dos vértices é $V = N \cup T$.

(2) Para cada produção $A \rightarrow X_1\ X_2\ \ldots\ X_n$, em que $n > 0$ e $X_1, X_2, \ldots X_n \in V$, liga-se A a X_1 por meio de um arco, se esse arco não existir ainda. Se $X_1 \Rightarrow^* \lambda$, acrescenta-se um arco de A para X_2. Se além disso $X_2 \Rightarrow^* \lambda$, traça-se um arco de A para X_3, e assim sucessivamente.

NOTA: Observe-se que nenhum $a \in T$ tem arcos incidentes para o exterior.

Os resultados pretendidos são:

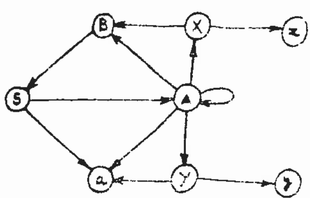
- para cada $X \in V$, prim (X) é o conjunto dos descendentes de X que são vértices terminais;
- a gramática é recursiva à esquerda se e só se algum $A \in N$ estiver contido num circuito.

EXEMPLO. - Consideremos a gramática apresentada anteriormente, que aqui se repete:

$$S \rightarrow a\ S \mid A\ B$$
$$A \rightarrow X\ a\ Y \mid X\ A \mid B\ Y$$
$$B \rightarrow \lambda \mid S\ B$$
$$X \rightarrow B\ B \mid x$$
$$Y \rightarrow y\ B \mid a$$

Recordemos que os símbolos que geram λ são B e X.

O grafo é o seguinte:



Vê-se que prim (S) = prim (A) = prim (B) = prim (X) = {a,x,y} e prim (Y) = {a,y}. Por outro lado, a gramática é recursiva à esquerda porque, por exemplo, existe o circuito (A,A).

COMENTÁRIO FINAL

Ambos os algorítmos baseiam-se numa noção alargada de "primeiro", em que o primeiro de um símbolo não terminal pode ser também um símbolo não terminal. A ideia dos algorítmos é a de definir esta noção alargada como o fecho transitivo de uma noção mais simples e, de seguida, exprimir a recursividade esquerda e a noção usual de "primeiro" em termos desta noção alargada. Logo, todo o peso dos algorítmos é transferido para o cálculo do fecho transitivo.

Mais formalmente, dados X, Y $\in$ V ponhamos

X p Y se e sõ se existem uma produção

$X \rightarrow X_1 X_2 \dots X_n$ com $n > 0$
e i, $1 \leq i \leq n$, tais que
$X_1 X_2 \dots X_{i-1} \Rightarrow^* \lambda$ e $Y = X_i$.

Note-se que isto é uma forma mais sintética de dizer que no grafo que foi construído anteriormente existe um arco de X para Y. Ponhamos

X p+ Y se e sõ se existirem $X_0, X_1, \dots, X_n$ $(n > 0)$
tais que $X = X_0$ p $X_1 \dots$ p $X_n = Y$.

(p+ é o fecho transitivo de p). O resultado inicial a ter em conta (e que não demonstraremos para não sobrecarregar a exposição) é que

X p+ Y se e sõ se existe $\beta \in V^*$ tal que $X \Rightarrow^* Y\beta$.

Com base neste resultado é fácil verificar que

- a gramática é recursiva à esquerda se e só se A p+ A para algum $A \in N$;
- dado $A \in N$, prim $(A) = \{a \in T: A$ p+ $a\}$.

Com estas observações ficam justificados os algorítmos.

C. <u>Determinação do conjunto dos seguintes de cada símbolo não terminal</u>

Este algorítmo também se baseia no cálculo do fecho transitivo de um grafo cujo conjunto de vértices é V. Antes de vermos quais são os arcos do grafo consideremos uma produção da forma

$$A \rightarrow \alpha B \beta$$

em que $B \in N$, $\alpha, \beta \in V^*$. Podemos concluir que:

1º Os primeiros de β são seguintes de B, isto é, prim $(\beta) \subset$ seg (B). Com efeito, seja $a \in$ prim (β): existe $\beta' \in V^*$ tal que $\beta \Rightarrow^* a\beta'$. Por outro lado, como a gramática é limpa, A é acessível do axioma: existem $\delta, \gamma \in V^*$ tais que $S \Rightarrow^* \delta A \gamma$. Pode-se concluir que:

$$S \Rightarrow^* \delta A \gamma \Rightarrow \delta \alpha B \beta \gamma \Rightarrow^* \delta \alpha B a \beta' \gamma$$

o que mostra de facto que $a \in$ seg (B).

2º Se $\beta \Rightarrow^* \lambda$, os seguintes de A são também seguintes de B. (Se, em particular, $\beta = \lambda$, isto é, se a produção anterior for da forma $A \rightarrow \alpha B$, também se tem seg (A) $\subset$ seg (B).) Com efeito, seja $a \in$ seg (A). Por definição de seguinte, existem $\delta, \gamma \in V^*$ tais que $S \Rightarrow^* \delta A a \gamma$.

Pode-se escrever:

$$S \Rightarrow^* \delta A a \gamma \Rightarrow \delta \alpha B \beta a \gamma \Rightarrow^* \delta \alpha B a \gamma$$

o que mostra que $a \in$ seg (B).

Em face destes resultados, não é difícil concluir que seg (B) é a reunião de todos os conjuntos da forma:

prim (β) $\cup$ <u>if</u> lambda (β) <u>then</u> seg (A) else $\emptyset$

para toda a produção $A \rightarrow \alpha B \beta$ em que B ocorra na parte direita.

Vamos agora ver como a determinação dos conjuntos dos seguintes dos símbolos não terminais pode ser feita a partir do cálculo do fecho transitivo de um certo grafo.

(1) - O conjunto dos vértices do grafo é V.

(2) - Para cada símbolo não terminal B, seleccionar todas as produções em que B ocorre na parte direita,

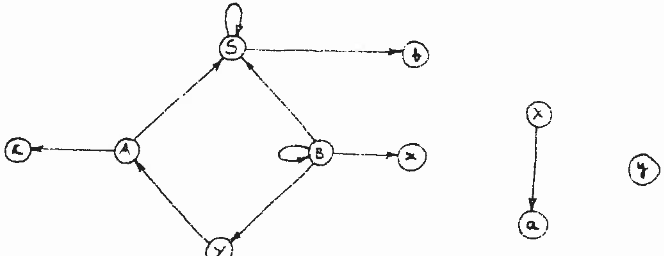
(a) - Para cada produção da forma $A \rightarrow \alpha B$, existe um arco de B para A.

(b) - Para cada produção da forma $A \rightarrow \alpha B X_1 X_2 \ldots X_n$, em que $X_1, X_2, \ldots, X_n \in V$, existe um arco de B para cada $a \in$ prim (X_1). Se $X_1 \Rightarrow^* \lambda$, existe também um arco de B para cada $a \in$ prim (X_2), e assim sucessivamente. Se também $X_n \Rightarrow^* \lambda$, existe igualmente um arco de B para A.

EXEMPLO: Seja a gramática:

$$
\begin{array}{lcl}
S & \rightarrow & a\,S \mid A\,B \\
A & \rightarrow & X\,a\,Y \mid S\,b \\
B & \rightarrow & \lambda \mid c\,B \\
X & \rightarrow & B\,x \mid x \\
Y & \rightarrow & y\,B \mid y
\end{array}
$$

O grafo (dos seguintes) associado a esta gramática é



Aplicando os algorítmos anteriores tinhamos obtido os primeiros de cada símbolo, e os símbolos que geram λ. Para justificar os arcos que partem de A notamos que a única produção em que A ocorre na parte direita é $S \rightarrow A\ B$. Dado que prim $(B) = \{c\}$ e $B \Rightarrow^* \lambda$, existe um arco de A para c e outro de A para S.

De posse de um tal grafo, o conjunto dos seguintes de $A \in N$ é o conjunto de todos os descendentes de A que são símbolos terminais.

Para o exemplo anterior temos

seg $(S) = \{b\}$, seg (Y) = seg $(A) = \{b,c\}$, seg $(B) = \{b,c,x\}$, seg $(X) = \{a\}$.

D. Verificação das condições LL (1)

Verificar se uma dada gramática G é LL (1) é agora fácil.

(1) Verifica-se se G está limpa, para o que já foram apresentados algorítmos noutro capítulo.

(2) Determina-se o conjunto dos símbolos não terminais que podem gerar λ .

(3) Verifica-se, para cada regra

$$A \rightarrow X_1 \mid X_2 \mid \ldots \mid X_n$$

se podem existir duas expansões distintas X_i e X_j tais que $X_i \Rightarrow * \lambda$ e $X_j = * \lambda$.

(4) Calcula-se prim (A) para cada $A \in N$.

(5) Calcula-se seg (A) para cada $A \in N$.

(6) Calcula-se, para cada regra

$$A \rightarrow X_1 \mid X_2 \mid \ldots \mid X_n$$

os conjuntos dir (A,X_i), e verifica-se se dir $(A,X_i) \cap$ dir $(A,X_j) = \emptyset$ para todo o par de expansões distintas X_i, X_j.

4. Programação de um algorítmo de análise de gramáticas LL (1) na forma de um conjunto de procedimentos recursivos

No sentido de tornar sistemática a apresentação deste ponto, convém considerar a seguinte gramática que gera a linguagem das gramáticas expressas em BNF.

<Gramática BNF>::=<Produção><Gramática BNF>|<Produção>
<Produção>::= "<" <identificador>">" "::="<parte direita>
<Parte direita>::=<expressão BNF>
<Expressão BNF>::=<sequência>"|"<expressão BNF>|<sequência>
<Sequência>::=<factor><sequência>|<factor>
<Factor>::="<"<identificador>">"|<terminal>|<λ>
<Identificador>::=<letra><resto>
<Resto>::=<letra><resto>|<digito><resto>|λ
<Terminal>::= """ sequência de caracteres's ≠ de """"
<λ>::= "λ"

Tomando em consideração a nomenclatura assim introduzida as funções lambda e prim podem ser definidas recursivamente da seguinte forma:

Sejam E, E_1, E_2, ..., E_n <expressões BNF> ou <identificadores> da parte esquerda de uma produção. Então:

```
lambda (E)=
    if E = <terminal> then false
    else if = <λ> then true
    else if E = <identificador>  then lambda (parte direita de E)
    else if E = E1 E2 ... En then
                lambda (E1) e lambda (E2) ... e lambda (En)
    else if E = E1 | E2 | ... | En then
                lambda (E1) ou lambda (E2) ... ou lambda (En)
    end if;

    prim (E) =
          if E = <terminal> then {<terminal>}
          else if E = <λ> then ∅
          else if E = <identificador> then prim ( parte direita de E)
          else if E = E1 | E2 | ... | En then
                      prim (E1) ∪ prim (E2) ∪ ... ∪ prim (En)
          else if E = E1 E2 ... En then
          if not lambda (E1) then prim (E1)
          else if lambda (E1) and (not lambda (E2) then
                                    prim (E1) ∪ prim (E2)
            .
            .
            .
          else if lambda (E1) and lambda (E2) ... and lambda (En-1)
                  then prim (E1) ∪ prim (E2) ∪ ... ∪ prim (En)
          end if
    end if
```

Para a construção do algorítmo consideremos então que é dada uma gramática G obedecendo às condições LL (1), onde <P_1>, ..., <P_n> são os identificadores de não terminais e <P_1> o axioma.

Introduz-se então um tipo conjunto-terminal estendido de um valor suplementar que designaremos Fim-input.

Considera-se que a palavra da entrada é "lida" pela operação:

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

o qual devolve o valor Fim-input quando se tentar "lêr" para além do fim da palavra. Finalmente o algorítmo de análise, ao

qual chamaremos Parser é definido de forma recursiva como uma função que, a uma <gramática>, uma <produção>, ou uma <expressão BNF>, faz corresponder um algorítmo.

Parser é definido pelas seguintes 3 equações:

```
Parser (G) = 'program  Parser recursivo descendente;
              type conjunto-terminal = ...;
              var t: conjunto-terminal;
              procedure ler-terminal (var t: conjunto
                                        -terminal); ...;
              procedure ERRO (N: integer);
                  Begin
                       "assinalar mensagem de erro N";
                        abortar a execução
                   end;
              Parser (P1);
              Parser (P2);
                 :
              Parser (Pn);
              begin
                  ler-terminal (t);
                  P1;
                  if t <> Fim-input then ERRO
              end.
                  '

Parser (Pi::= Ei) = 'procedure Pi;
                         begin
                              Parser (Ei)
                          end;'

Parser (E) =
  if E = <terminal> then 'if t= E then ler-terminal (t)
                               else ERRO (i) end if;'
  else if E = <λ> then ';'
      else if E = <identificador> then   'P-identifi
                              cador (* isto é Pj *);'
```

```
else if = E1 E2 ... En then 'Parser (E1);
                             Parser (E2);
                               :
                             Parser (En);'
else if E = E1 | E2 | ... | En then
        'if t in dir (E1) then Parser (E1)
         else if t in dir (E2) then Parser (E2)
          :
         else if t in dir (En) then Parser (En)
         else ERRO (K)
         end if;'
```

Para completar basta dizer que em certas circunstân cias pode suceder que dir $(E_K)=\emptyset$ (Este caso surge quando a lin guagem gera a palavra vazia) que não permite a escrita da expres são t in dir (E_K).

Neste caso particular aquela expressão escreve-se to mando como dir (E_K) o conjunto: {Fim-input}

EXEMPLO 1: dada a gramática:

```
<bloco>::=  begin  "D" ";" <X> "S" <Y> end
<X>::= "D" ";" <X> | λ
<Y>::= ";" "S" <Y> | λ
```

obtinha-se em Pascal, por exemplo:

```
procedure bloco;
     begin
        if t= "begin" then ler-terminal (t)
                                   else ERRO (1);
          if t= "D" then ler-terminal (t) else
                                           ERRO (2);
          if t= ";" then ler-terminal (t) else
                                           ERRO (3);
          X;
          if t= "S" then ler-terminal (t) else
                                           ERRO (4);
```

```
                    Y;
                    if t= "end" then ler-terminal (t) else
                                                  ERRO (5);
                end;

            procedure X;
                begin if t= "D" then
                        begin
                        if t= "D" then ler-terminal (t)
                                       else ERRO (2);
                        if t= ";" then ler-terminal (t)
                                       else ERRO (3);

                        X
                        end

                else if t= "S" then begin end
                else ERRO (6)
                end ;
```

e assim sucessivamente

Exemplo 2: dada a gramática:

<S>::= "a" <S> | λ

Obtem-se:

```
procedure S;
    begin
        if t= "a" then begin if t= "a" then ler-ter
                        minal (t) else ERRO (1); S end
        else if t= Fim-input then begin end
        else ERRO (1)
    end;
begin
     ler-terminal (t),
     S;
     if t <> Fim-input then ERRO (1) (*por completu-
                                              de *)
end;
```

Demonstra-se que se a gramática satisfizer as condições LL (1), o algorítmo definido por Parser, termina sempre para qualquer que seja a palavra de entrada, assinalando um erro sempre que esta não pertencer à linguagem gerada pela gramática.

Em particular, se a gramática for regular direita gera um algorítmo equivalente ao algorítmo recursivo apresentado para os autómatos.

5. Considerações sobre o algorítmo

A. Conjunto-terminal

Quando o conjunto-terminal não é um tipo primitivo da linguagem de programação podem-se tomar opções semelhantes às apresentadas para os autómatos.

B. Cadeia IF ELSE IF

Esta cadeia pode ser transformada num CASE quando as condições para tal exigidas, no caso do autómato, forem satisfeitas. As condições LL (1) garantem a disjunção dos ramos.

C. Forma das expressões t in Dir (E)

Como já se mostrou nos exemplos anteriores, sempre que uma expressão de uma produção tem a forma

<terminal> ....

Esta expressão toma a forma t= "terminal".

Desde que se conserve o significado matemático de

t in Dir (E)

pode ser usada uma expressão equivalente mais eficiente (por exemplo por complemento).

D. Tratamento de erros

O único tratamento de erros introduzido é abortar a execução sempre que um erro é detectado. Não se apresenta qualquer técnica de tentativa de recuperação.

No caso mais simples, ERRO pode não ter parâmetros e a mensagem ser única: 'ENCONTROU-SE UM ERRO EM: ', t.

F. Optimização do algorítmo

Uma optimização evidente no algorítmo atrás definido é a eliminação de testes redundantes sempre que uma expansão tem um terminal à cabeça, o qual é o elmento único de dir (expansão). Ou seja:

Se P::= "a" ... | "a_2" ... | "a_3" ... | ... "a_n"
Parser ("a_1") ... Parser ("a_n") pode reduzir-se a 'ler-terminal (t)'

Existem muitas outras transformações que se podem executar no algorítmo no sentido de o optimizar. Tal discussão só tem interesse num quadro mais geral que será introduzido no próximo capítulo.

6. Transformação de uma gramática não LL (1) em uma gramática LL (1)

O problema de saber se existe uma gramática LL (1) que gera a mesma linguagem que uma dada gramática não LL (1) é indecidível. Na práctica seguem-se 3 vias possíveis:

a) Tentativa de transformar a gramática.
b) Modifica-se a linguagem para encontrar uma gramática LL (1) da nova linguagem.
c) Utiliza-se a semântica.

No que toca à primeira transformação, esta faz-se, no essencial, aplicando 3 operações:[1]

- eliminação da recursividade à esquerda
- factorização de produções:

Ex: $A \longrightarrow a \mid aA$
é equivalente:

$A \longrightarrow a\ X$
$X \longrightarrow A \mid \lambda$

---

(1) - *Nos exemplos deste ponto as gramáticas são apresentadas com os não terminais em maiusculas e os terminais em minusculas.*

- Substituição, por expansão, de um símbolo não terminal, afim de aplicar em seguida a factorização:

  Ex: A → Ba | Cb
  B → aB | bC
  C → a | d

  Substituindo-se B e C em A

  A → a B a | b C a | a b | d b
  B → a B | b c
  C → a | d

  factorização:

  A → a X | b C a | d b
  X → B a | b
  B → a B | b c
  C → a | d

  substituindo B em X:

  A → a X | b C a | d b
  X → a B a | b C a | b
  ...

  factorização em X:

  A → a X | b C a | d b
  X → a B a | b y
  Y → C a | λ
  B → a B | b c
  C → a | d

A recursividade à esquerda é, em geral, introduzida para permitir repetições. A eliminação da recursividade à esquerda pode fazer-se transformando a gramática e introduzindo recursividade à direita para a obtenção das repetições. Qualquer um destes objectivos é mais fácilmente atingido introduzindo os operadores de repetição que serão apresentados num próximo capítulo.

Uma outra forma de soluccionar o problema da não verificação das condições LL (1), consiste em modificar a gramática, transformando a própria linguagem por esta gerada.

Quando o desenho de uma linguagem não está estabili

zado, ou quando pequenas modificações são irrelevantes, deve-se transformar a gramática de forma a obter fácilmente uma gramática LL (1) que gera a nova linguagem.

Finalmente, um terceiro método consiste em utilizar a própria semântica. Um exemplo típico desta solução é geralmente utilizado na implementação dos compiladores de Pascal. Em Pascal, um identificador dentro de um bloco pertence não só ao conjunto dos directores da afectação como ao conjunto dos directores de uma chamada de procedimento . No entanto, é extremamente fácil a nível da análise lexicográfica separar os terminais: "Identificador de variável" e "Identificador de procedimento". Assim, a nível da análise sintáctica o problema da não disjunção dos directores é resolvido estendendo o conjunto terminal, através da introdução de significados semânticos.

7. Exercícios

Os 2 exercícios que se seguem são retirados de: R. C. Backhouse. Syntax of Programming Languages, Prentice Hall.

1. Quais das seguintes gramáticas são LL (1):
   (Terminais em minusculas, não terminais em maiusculas)

a) $S \rightarrow a\ A$
   $A \rightarrow S \mid \lambda$

b) $S \rightarrow C \mid D$
   $C \rightarrow a\ C \mid c$
   $D \rightarrow a\ D \mid d$

c) $S \rightarrow A\ B \mid a$
   $B \rightarrow C\ D \mid a\ E$
   $C \rightarrow a\ b$
   $D \rightarrow b\ b$
   $E \rightarrow b\ b\ a$

d) $S \rightarrow a\ A\ S \mid b$
   $A \rightarrow a \mid b\ S$

e) $S \rightarrow A\ B$
   $A \rightarrow a\ A\ b \mid c$
   $B \rightarrow b\ B \mid c$

f) $S \rightarrow A\ B$
   $A \rightarrow B\ a \mid \lambda$
   $B \rightarrow C\ b \mid C$
   $C \rightarrow c \mid \lambda$

2. A seguinte gramática define expressões regulares sobre o alfabeto $\{a,b\}$. Transforme-a na forma LL (1).

<E>::=<E> "+" <T> | <T>

<T>::=<T> <F> | <T>

<F>::=<P> "*" | <P>

<P>::="∅" | "λ" | "a" | "b"

3. Transforme a gramática das gramáticas BNF apresentada neste capítulo numa gramática LL (1).

Os exercícios que se seguem são retirados da referência:

N. Wirth, Algorithms + data structures = programs, Prentice Hall, Zurich, 1976.

4. Dada a gramática:

<S>::= <A>

<A>::= <B> | "if" <A> "then" <A> "else"<A>

<B>::= <C> | <B> "+"<C> | "+"<C>

<C>::= <D> | <C> "*"<D> | "*"<D>

<D>::= "X" | "(" <A> ")" | "-" <D>

Verifique se a gramática é LL (1). Se o não for determine uma gramática LL (1) que gere a mesma linguagem.

5. Idem para a gramática:

<S>::= <A>

<A>::= <B> | "if" <C> "then" <A> | "if" <C> "then" <A> "else" <A>

<B>::= <D> "=" <C>

<C>::= "if" <C> "then" <C> "else" <C> | <D>

<D>::= "a" | "b" | "c"

NOTA: Poderá ter que suprimir algum constructo para obter aquele resultado.

6. A gramática que a seguir se apresenta descreve um sub-conjunto modificado da linguagem Pascal, e foi extraída da A. V. AHO e J. D. ULLMAN, Principles of Compiler Design, Addison-Wesley, Reading,

Mass., 1977 . Pretende-se determinar uma gramática LL (1) equivalente.
Para melhor esclarecimento, além das produções apresentam-se os símbolos terminais.

SÍMBOLOS TERMINAIS (por ordem de entrada em cena):

```
program "IDENTIFIER" "(" ")" ";" "," var ":"
integer real array "[" "CONSTANT" ".." "]"
of function result procedure begin end
if then else while do "ASSINGNOP" "RELOP"
"ADDOP" "MULOP" not "+" "-"
```

PRODUÇÕES:

```
<program>::= program "IDENTIFIER" (<identifier list>)
             ";" <declarations> <subprogram declara
             tions> <compound statement>
<identifier list>::= "IDENTIFIER" | <identifier list>
                     "," "IDENTIFIER"
<declarations>::= "var" <declaration list> |λ
<declaration list>::= <identifier list> ":" <type> ";"
                      |<declaration list><identifier
                      list> ":" <type>";"
<type>::= <standard type> | <array type>
<standard type>::= integer | real
<array type>::= array "[" "CONSTANT" ".." "CONSTANT"
                "]" of <standard type>
<subprogram declarations>::= <subprogram declarations>
                             < subprogram declaration>|λ
<subprogram declaration>::= <subprogram head><declara
                            tions><compound statement>
<subprogram head>::= function "IDENTIFIER"<arguments>
                     ":" result <standard type> ";"
                     | procedure "IDENTIFIER" <argu-
                     ments> ";"
<arguments>::= (<parameter list>)|λ
<parameter list>::= <identifier list> ":" <type>
                    |<parameter list> ";" <identifier
                    list> ":" <type>
```

```
<compound statement>::= begin <statement list> end
<statement list>::= <statement>|<statement list>
                    ";" <statement >
<statement>::= <elementary statement>| if <expression>
               then  <restricted statement>  else
               <statement>| if  <expression>  then
               <statement>| while  <expression>  do
               <statement>
<restricted statement>::= <elementary statement>
                          | if  <expression>  then
                          <restricted statement>
                           else  <restricted statement>
                          |while <expression>  do
                          <restricted statement>
<elementary statement>::= <variable> "ASSIGNOP"<expres
                          sion>|<procedure statement>
                          |<compound statement>
<variable>::= "IDENTIFIER" | "IDENTIFIER" "[" <expres-
              sion> "]"
<procedure statement>::= "IDENTIFIER" | "IDENTIFIER"
                         "(" <expression list> ")"
<expression list>::= <expression> | <expression list>
                     "," <expression>
<expression>::=<simple expression>|<simple expression>
               "RELOP" <simple expression>
<simple expression>::=<term>|<sign>|<term>|<simple ex-
                      pression> "ADDOP" <term>
<term>::= <factor>|<term> "MULOP" <factor>
<factor>::= <variable>|"CONSTANT"|"("<expression>")"
            |<function reference>| not <factor>
<function reference>::= "IDENTIFIER" | "IDENTIFIER"
                        "("<expression list>")"
<sign>::= "+" | "-"
```

8. PROGRAMAÇÃO PELA SINTAXE UTILIZANDO LINGUAGENS "CONTEXT FREE"

1. Introdução informal de uma linguagem para especificar programas pela sintaxe - SINTAX

Atrás foram introduzidas as gramáticas "context-free" assim como os formalismos BNF e Extended BNF para descrever gramáticas.

Quando essa gramática satisfaz as condições LL(1), ela pode ser vista como denotando, quer uma linguagem, quer um algorítmo (o algorítmo de reconhecimento e percurso de uma palavra dessa linguagem).

O método da programação pela sintaxe consiste, no essencial, em conceber um programa que tem por entrada uma palavra pertencente a uma dada linguagem e que através de funções semânticas inseridas convenientemente no algorítmo de reconhecimento, produz uma determinada saída, isto é, calcula o resultado da computação. Tal programa é desenhado tendo como esqueleto o algorítmo de reconhecimento de palavras geradas pela gramática que descreve a sua entrada.

A sua análise é conduzida, de preferência, introduzindo funções semânticas na própria gramática, já que a obtenção do algorítmo final é trivial.

Tendo em atenção estes aspectos é natural conceber a linguagem das gramáticas com funções semânticas como uma linguagem de especificação e/ou modelização de programas concebidos pelo método da "programação pela sintaxe".

Em seguida apresentaremos uma linguagem, a linguagem SINTAX (Programação pela Sintaxe), cuja definição informal procura fixar uma notação capaz de suportar o método da programação pela sintaxe.

2. Sintaxe da linguagem SINTAX

Os operadores +, * e ? foram introduzidos com o significado de "1 ou mais vezes", "0 ou mais vezes", "0 ou 1 vezes", respectivamente.

A partir deste capítulo passaremos a usar as notações:

+ significa alternativamente (geralmente denotada "/")

{exp} significa exp *

[exp] significa exp +

exp ? significado já conhecido.

Uma gramática das gramáticas expressas nesta nomenclatura é definida nela própria por:

```
Gramática = [Equação "."].
Equação     = Id "=" Parte direita.
Parte-direita = Exp-EBNF.
Exp-EBNF   = Sequência {"+" Sequência}.
Sequência = [Factor].
Factor      = Simples + Repetição .
Simples     = (Terminal + Id + "(" Exp-EBNF ")")"?"?.
Repetição = "{" Exp-EBNF  "}" + "[" Exp-EBNF "]".
Id          = Letra {Letra+Digito + "_"}.
Terminal    = """ {caracter} """ ou Id sublinhado.
```

Utilizando a nomenclatura assim introduzida a sintaxe de SINTAX é definida por:

```
Programa-pela-sintaxe = Modulo Id1 define Id2
                        Sobre comentário
                        Equações [Equação]"." fim Id1.
Equação     = Id "=" Exp-EBNF.
Exp-EBNF    = Sequência {"+" Sequência}.
Sequência = [Factor].
Factor      = Simples + Repetição + Semântico.
```

```
Simples           = (Id + Terminal + "("Exp-EBNF")")"?"?.
Repetição         = "[" Exp-EBNF "]" + "{" Exp-EBNF "}".
Semântico         = "/*" Corpo-semântico "*/".
Corpo-semântico   = {c}.
Id                = letra {letra + digito + "-"}.
Terminal          = """ {c} """ ou Id sublinhado.
C                 = qualquer caracter.
Letra             = "A" + ... + "Z" + "a" + ... + "z".
Digito            = "0" + ... + "9".
Id1               = Id.
Id2               = Id.
Comentário        = "(*" {c} "*)".
```

Pode-se demonstrar que este sistema tem tanta potência de denotar linguagens como as expressões regulares, as gramátiacs "context free", os automátos deterministas ou os sistemas de equações regulares. (qualquer dos formalismos anteriores munido de funções semânticas).

O identificador Id_1 que segue sempre as palavras reservadas Modulo e fim é identificador selecionado para identificar o programa. Id_2 é o identificador do axioma da gramática. O comentário que se segue a Sobre permite descrever informalmente o conjunto terminal.

O não terminal 'comentário' permite introduzir comentários no programa que, uma vez removidos, não alteram o seu significado .

O não terminal 'semântico' permite a introdução de funções semânticas no sistema de equações. Adicionalmente, fixa-se que nenhuma 'Sequência' pode ser constituída apenas por acções semânticas, tendo que conter pelo menos um não terminal: 'Simples' ou 'Repetição'.

3. Semânticas de SINTAX

A semântica de Sintax exige que a Gramática ou sistema de equações especificado verifique as condições LL(1).

As definições dos conjuntos:

prim, seg e dir

São assumidas na forma de funções que a cada 'Exp-EBNF' associam um conjunto de símbolos terminais. Também se assume a definição do predicado lambda que é uma função que a cada 'Exp-EBNF' associa o valor true ou false.

Prim (Exp-EBNF) pode ser definido pela função recursiva:

Sejam $E, E_1, E_2, \ldots, E_n$ Exp-EBNF'S

Prim (E) =

```
if E = terminal then {E}
else if E = Id then Prim (da parte direita da equa-
                     ção com Id como parte esquerda)
else if E = (E1) then Prim (E1)
else if E = [E1] then Prim (E1)
else if E = {E1} then Prim (E1)
else if E =  E1? then Prim (E1)
else if E =  E1 + E2 + ... + En then
             Prim (E1) V Prim (E2) V ... V Prim (En)
else if E =  E1 E2 E3 ... En then
       if not lambda (E1) then prim (E1)
       else if lambda (E1) and (not lambda (E2)) then
                              prim (E1) V prim (E2)
          .
          .
          .
       else if lambda (E1) and lambda (E2) and ...
               ... and (not lambda (En-1)) then
                   prim (E1) V prim (E2)V ... V prim
                                             (En-1)
       else if lambda (E1) and ... and lambda (En-1)
        then prim (E1) V prim (E2) V ... V prim (En)
```

```
    end if
end if;
```

NOTA: prim (Semântico) e prim (Comentário) não têm significado.

Lambda (Exp-EBNF) pode ser definido pela função recursiva:

Sejam E, E_1, E_2, ..., E_n Exp-EBNF'S

```
Lambda (E) =
   if E = terminal then false
   else if E = Id then lambda (da parte direita da equa
                     ção com Id como parte esquerda)
   else if E = (E1) then lambda (E1)
   else if E = {E1} ou E = E1? then true
   else if E = [E1] then lambda (E1)
   else if E = E1 + E2 + ... + En then
               lambda (E1) V ... V lambda (En)
   else if E = E1, E2, ..., En then
               lambda (E1) Λ ... Λ lambda (En)
   end if;
```

NOTA: lambda (Semântico) e lambda (Comentário) não têm significado.

Recordemos que, informalmente, as condições LL(1) se podem exprimir por:

a) A gramática expressa é "limpa", isto é:

a.1) Todas as equações são acessíveis a partir de expansões da equação do axioma.

a.2) A linguagem gerada por qualquer expansão, de qualquer equação, sobre o conjunto terminal, é não vazia.

b) Para qualquer expressão EBNF presente no sistema e da forma: $E_1 + E_2 + \ldots + E_n$.

então:

b.1) $\forall i,j$ dir $(E_i) \cap$ dir $(E_j) = \emptyset$

b.2) Se $\exists_{i,j}$: lambda $(E_i) \wedge$ lambda (E_j)
então $i = j$

c) Para qualquer expressão EBNF presente no sistema tal que:

lambda (E) então
prim (E) $\cap$ seg (E) = $\emptyset$

Assim, dado um programa em SINTAX, tal que a parte sintáctica que especifica constitua uma gramática que satisfaz as condições LL(1), é possível definir o significado semântico do programa.

Antes porem de prosseguirmos neste aspecto referiremos que cada programa em SINTAX tem um equivalente gráfico definido pelo conjunto de equações que a seguir se apresentam:

Seja P um programa em SINTAX, sintácticamente correcto.(1) Seja Pid o seu identificador e sejam $P_1, \ldots, P_n$ os identificadores das equações, com P_1 o identificador do axioma.

Grafo é uma função que a P associa uma sua representação gráfica definida pelas seguintes equações:

Grafo (P) =

' Grafo sintáctico do módulo Pid

definindo P_1 sobre comentário

Grafo (P_1)

⋮

Grafo (P_n) '

Grafo (P_i) =

' P_i ⟶ Grafo (da parte direita de que P_i é a parte esquerda) ';

(1) - *Isto é, conforme à sintaxe de SINTAX e denotando uma gramática LL(1).*

```
Grafo (Exp-EBNF) =

  if Exp-EBNF = Terminal then        '──(ter)──→ '
  else if Exp-EBNF = Id then         '──[ Id ]──→ '
  else if Exp-EBNF = Semântico then   ' /* Corpo Semântico*/ '
  else if Exp-EBNf = [E1] then        ' ── Grafo (E1) ──→ '
  else if Exp-EBNF = {E1} then        ' ─────────── '
                                       Grafo (E1)
  else if Exp-EBNF = (E1) then        ' ── Grafo (E1) ──→ '
  else if Exp-EBNF = E1? then         ' ─────────── '
                                       Grafo (E1)
  else if Exp-EBNF = E1 E2 ... En then
              '──→ Grafo (E1) ──      ──→ Grafo (En) ──→ '
  else if Exp-EBNF = E1 + E2 + ... + En then

                     ┌──→ Grafo (E1) ──┐
                     ├──→ Grafo (E1) ──┤
                   ──┤        .        ├──→
                     │        .        │
                     └──→ Grafo (En) ──┘

end if;
```

NOTA: Grafo de comentário não tem significado.

Finalmente, a semântica de um programa SINTAX obedecendo às condições LL(1) é definida pela função Parser que aplica o programa SINTAX num programa expresso, no nosso exemplo, em Pascal (por força deste aspecto as funções semânticas deverão também ser expressas em Pascal).

Parser é definido por 3 equações:

```
Parser (P) = '

    (* conjunto de definições denotadas pelo módulo
       Pid *)

    type conjunto-terminal = comentário que se segue
                                                a sobre;
    var t: conjunto-terminal;
    procedure ler-terminal (var t: conjunto-terminal);
```

```
procedure ERRO;
Parser (P1);
   :
Parser (Pn);
begin
     ler-terminal (t);
     P;
     if t≠Fim-input then ERRO
end  (*Pid*); '

Parser (Pi) = 'procedure Pi
                   begin
                     Parser (da parte direita da equação
                         cuja parte esquerda é Pi)
                   end (*Pi*);'

Sejam E, E1, ..., En Expressões EBNF

Parser (E) =

   if E = terminal then ' if t = terminal then ler-terminal
                                              (t) else ERRO;'
   else if E = Id then 'Id;'
   else if E = Comentário then 'Comentário'
   else if E = Semântico then 'begin Corpo-semântico end;'
   else if E = E1? then 'if t in prim (E1) then Parser (E1);
                         if  not (t in seg(E))then ERRO;'
   else if E = (E1) then 'begin Parser (E1) end;'
   else if E = [E1] then 'if not (t in dir (E)) then ERRO;
                          while t in prim (E1) do Parser (E1);
                          if not (t in seg (E)) then ERRO;
   else if E = {E1} then 'while t in prim (E1) do Parser (E1);
                         if not  (t in seg (E)) then ERRO
   else if E = E1, E2 ... En then 'begin Parser (E1);...;
                                          Parser (En) end;'
   else if E = E1 + E2 + ... + En then'
```

```
if t in dir (E1) then Parser (E1)
else if t in dir (E2) then Parser (E2)
   :
else if t in dir (En) then Parser (En)
else ERRO '

end if;
```

Note-se que a semântica de SINTAX segue a estratégia de testar sempre tudo e assinalar os erros tão cedo quanto o processo de análise o permita[1]

Para que o programa denotado pelo sistema, seja transformado definitivamente num programa concreto é necessário:

a) Concretizar a estratégia de tratamento de ERROS.

b) Concretizar a "interface" do "Parser" com o conjunto terminal, concretizando a operação ler-terminal.

c) Tomar em atenção que é necessário introduzir alguns parentesis begin end suplementares, para manter em Pascal o significado de Parser. Idêntico problema coloca-se em relação ao separador de instruções ";".

d) Optimizar o algoritmo evitando testes redundantes, os quais são inúteis nos casos típicos:

d.1) Se $E = E_1 + E_2 + \ldots + E_n$ e Prim (E_1) ... prim (E_n) são conjunto constituídos por 1 único terminal.

d.2) Se $E = \{E_1\}$ e Prim (E_1) é constituído por 1 único terminal.

d.3) Se $E = [E_1]$ e Prim (E_1) é constituído por 1 único terminal.

e) Estender o conjunto-terminal com o valor Fim-input, que passará a ser o único elemento de todos os conjuntos directores inicialmente vazios.

f) Concretizar alguns operadores semânticos introduzidos.

---

(1) *Reconhece-se lambda testando se na entrada está um "seguinte"*

Apresenta-se em seguida um pequeno exemplo.

Módulo reconhecedor-de-inteiros define inteiro

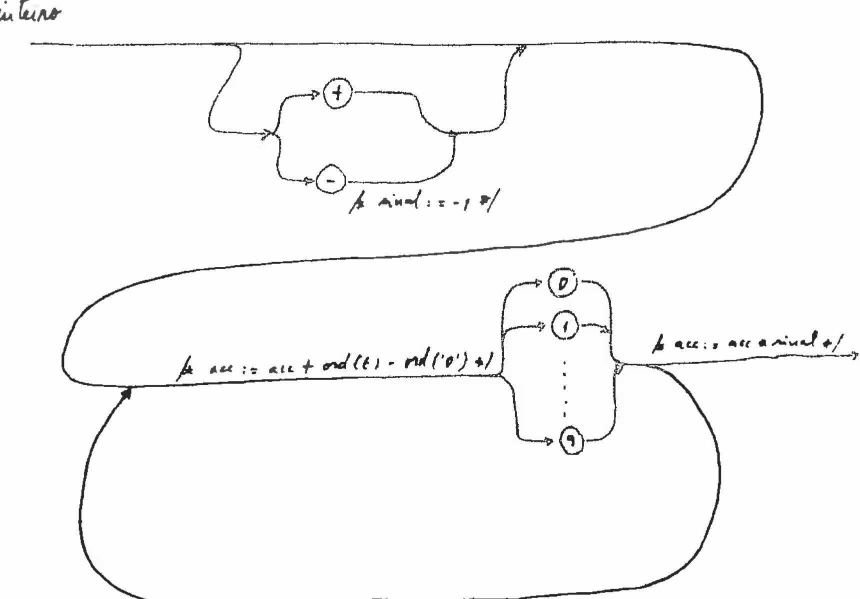
Sobre (* char *)

Equações

```
        inteiro = /* acc: = 0; sinal: = 1 */ (*iniciali
                                                 zação*)
                   ("+" + "-" /* sinal: = -1 */)?
                   [ /* acc: = acc *10 + ord (t) -ord
                                              ('0');*/
                     ("0" + "1" + "2" + "3" + "4" + "5" +
                          + "6" + "7" + "8" + "9")]
                    /* acc: = acc * sinal */.
```

fim reconhecedor-de-inteiros.

Ao qual corresponde o grafo:



E o algoritmo:

```
(* conjunto de definições denotadas pelo módulo reconhe-
                                         cedor-de-inteiros*)
type conjunto-terminal = char;
var t: conjunto-terminal;
procedure ler-terminal (var t: conjunto-terminal);
procedure ERRO;
procedure inteiro;
      begin
          acc: = 0; sinal: = 1; (*inicialização*)
          if t in [ '+', '-' ] then
                if t in ['+'] then
                             if t =['+'] then ler-terminal(t)
                             else ERRO
                else if t in [ '-'] then
                          begin
                            if t =['-'] then ler-terminal (t)
                            else ERRO; sinal: = -1
                         end
                else ERRO;
               if not t in ['0'..'9'] then ERRO;
          if not t in  '0' .. '9'  then ERRO;
          while t in  '0' .. '9'  do begin
                   acc: = acc * 10 + ord (t) -ord ('0');
                   if t in ['0'] then
                        if t = '0' then ler-terminal (t)
                                                      else ERRO
                   else if ....
                        :
                   else if t[in] '9'  then
                        if t = '9' then ler-terminal (t)
                                                      else ERRO
                   else ERRO
          end;
```

```
        if t <> fim-input then ERRO

    end (*inteiro*);

begin
     ler-terminal (t)
     inteiro;
     if t <> Fim-input then ERRO
end; (*reconhecedor de inteiros*)
```

O qual daria, depois de optimizado e com as diferentes indefinições concretizadas:

```
(*reconhecedor de inteiros optimizado*)

const Fim-input = '*';

var t: char;

procedure ler-terminal (var t: char);

     begin
         if eoln (input) then t: = Fim-input else read (t)
     end;
procedure ERRO;
     begin
          writeln ('caracter proibido');
          (*abortar execução*)
     end;
```

```
procedure inteiro;
    begin acc:=0;sinal:=1; (*inicialização*)
          if t='+' then ler-terminal (t) else
          if t='-' then begin
                              ler-terminal (t);
                              sinal:=-1
                        end;
          if t in ['0' .. '9'] then
          while t in ['0' .. '9'] do begin
                acc:= acc * 10 + ord (t) - ord ('0');
                ler-terminal (t)
             end
          else ERRO
end (*inteiro*)
```

4. Características da linguagem sintax

Ainda que a semântica de SINTAX seja apresentada de uma forma informal, esta linguagem constitui uma notação "razoávelmente" adequada à expressão da análise de programas concebidos segundo o método da programação pela sintaxe. Como tal, constitui uma linguagem de modelização da solução de problemas analisados segundo aquele método. A sua transformação num programa Pascal é um processo manual não totalmente preciso pois, as optimizações, a interface com a entrada e o tratamento de erros, não são expressos claramente em SINTAX.

Por outro lado, o facto de se dispor de um equivalente gráfico do programa SINTAX, facilita não só a análise das condições LL(1) sobre os grafos, assim como, permite, quando o sistema é regular, deduzir um autômato finito determinista com funções semânticas equivalente ao programa gerado pela função Parser.

Um programa em SINTAX tem sempre 2 interpretações possíveis: ou concebê-lo como denotando um programa concreto pela sintaxe, ou concebê-lo como denotando um grafo que representa gráficamente o mesmo programa. Qualquer das interpretações é fixada em funções que ao programa fazem corresponder a interpretação. Qualquer das funções é definida por um conjunto de equações recursivas.

5. Um exemplo de aplicação

Pretende-se realizar um programa que tem por entrada um programa Pascal sintácticamente correcto e por saída uma árvore das declarações de procedimentos e funções da forma:

```
id
  id1
     id2
     id3
  id4
     id5
         id6
  id7
  :
```

Em que id é o identificador do programa, id_1, id_4 e id_7 identificadores de procedimentos e funções declarados no programa, id_2 e id_3 identificadores de procedimentos ou funções declarados dentro de id_1, id_5 dentro de id_4, id_6 dentro de id_5, etc. (1)

Para obtermos este programa podemos, numa primeira aproximação, conceber que um determinado módulo transformaria a entrada numa palavra de uma gramática que descreve apenas a informação estrictamente essencial para desenhar aquela árvore. Uma solução, partindo deste pressuposto, é modelizada pelo seguinte programa em SINTAX:

Módulo árvore-estática define programa

sobre (* id, program, procedure, function, begin, end *)

EQUAÇÕES

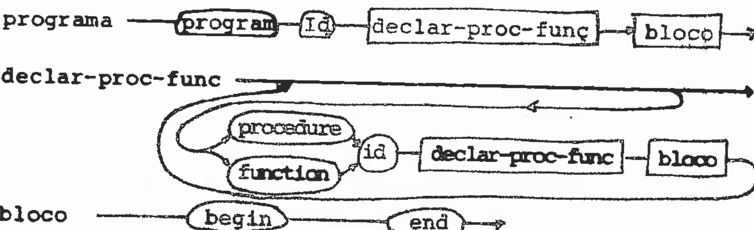
programa = program Id declar-proc-func bloco .

declarac-proc-func =

{(procedure+function) Id declar-proc-func bloco }.

bloco = begin end.

fim árvore-estática.



Que permite verificar imediatamente que se trata de um sistema LL(1).

---

(1) - *Adicionalmente fixa-se que o programa não tem declarações foward, nem parâmetros do tipo procedure ou function numa lista de parâmetros.*

De onde com funções semânticas:

```
Módulo árvore-estática define programa

sobre (* id, program, procedure, function, begin, end
                                                          *)

(* const margem-inicial = 5;
         factor = 10;
   var  pos: integer;
   procedure escrever (palavra: Id);
          begin
               write (' ': pos-1);
               writeln (palavra)
          end;
   procedure incrementar;
          begin
               pos: = pos + factor;
               if pos> 70 then ERRO
          end;
   procedure decrementar;
          begin
               pos: = pos-factor
          end;                          *)

EQUAÇÕES

programa = /* pos: = margem-inicial */ program  /* escre
  ver (t); incrementar */ Id declar-proc-func bloco

declarac-proc-func =
        {(procedure + function) /* escrever (Id); incremen
         tar */ Id declarac-proc-func bloco     /* decre-
         mentar */ }.

bloco = begin end.

fim árvore estática.
```

Prosseguindo na análise, é necessário conceber um módulo que construa a palavra apresentada à entrada do módulo árvore-estática.

Podemos por exemplo, conceber que esse módulo tem por entrada uma sequência de componentes elementares de um programa Pascal ("tokens") de onde são à partida retirados os comentários e qualquer outro "token" distinto de identificador ou palavra chave . As palavras chave que não são relevantes para o módulo são qualificadas como identificadores. O módulo constrói a palavra que será a entrada do módulo anterior pela função semântica out.

```
Módulo intermediário define programa-int
sobre  (* Id, program, begin, end, procedure, function, case
                                                          *)

(*
 var ficheiro-intermédio: file of  conjunto-terminal - do -
                                   módulo - árvore-estática ;
     contador: integer;
 procedure out (t: conjunto-terminal);
         begin
              put (ficheiro-intermédio, t) (*símbolo corrente*)
         end;
                        *)
 EQUAÇÕES

 programa-int = /*reset (ficheiro-intermédio);*/
                program /*out (program);out (t) */Id
                 term1   declarações bloco
 declarações = {         /*out (t);*/ (procedure + function)
                 /*out (t);*/ Id  term1  declarações bloco }.
 term1   = {Id}(*diferente de procedure ou function
                                                ou begin*).
 bloco   =  /*out (begin);*/ begin
            /*contador: = 1;
              while contador > 0 do
                 begin
                    if (t = begin) or (t = case) then contador
                                                  := contador+1
                    else if t = end then contador: =contador-1;
                    ler-terminal (t)
                 end;
                 out (end);
                                       */.
 fim intermediário.
```

Repare-se que no exemplo, a semântica é usada para atravessar o bloco. Trata-se de um método espedito de evitar estar a reconhecer toda a estrutura sintáctica das instruções de um bloco, pois para o nosso caso, só nos interessa atingir o end do bloco sem que este seja confundido com o end de um case ou de uma instrução composta.

Finalmente, o último módulo que transformaria o "input" do programa numa palavra sobre o alfabeto do módulo intermediário é um sistema regular ("analizador lexicográfico") que pode ser obtido fácilmente.

Para encontrar uma solução definitiva é necessário estabelecer:

a) Uma forma de acoplamento dos módulos.

b) Em função desta, uma representação para os terminais em jogo.

Para a ligação dos módulos, podemos partir das seguintes ideias:

Sendo o "analisador lexicográfico" um sistema regular, é fácil construir a partir do seu módulo em SINTAX um autómato, e a partir deste um procedimento, que passará a ser o procedimento ler-terminal do módulo intermediário.

Dado que a complexidade do problema é pequena, podem fundir-se os módulos intermediários e árvore-estática num único.

Para tal basta remover de intermediário as acções semânticas de construção do ficheiro intermédio e inserir nele, nas posições correspondentes, as acções semânticas de árvore-estática.

Finalmente o único conjunto terminal em jogo, para além dos caracteres de entrada no analisador lexicográfico, é:

{program, procedure, function, begin, end, case, Id}

o qual se representa bem em:

```
type conjunto-terminal = Record
                         classe: conjunto-classes;
                         valor : packed array [1..max] of char
                         end;
```

onde conjunto-classes =

(TPROG, TPROC, TFUNC, TID, TBEG, TEND, TCASE)

e os testes sobre o terminal corrente passarão a ser feitos sobre t.classe

Para concluir é interessante vereficar como a análise descendente em camadas de gramáticas permite obter as gramáticas mínimas relevantes para o problema, assim como os conjuntos terminais mínimos.

SINTAX revelou-se uma notação que no mínimo permite concretizar de uma forma razoávelmente cómoda, clara e segura o método da programação pela sintaxe.

ANEXO Projectos para aplicação dos métodos introduzidos

Neste anexo são apresentadas várias sugestões de projectos para aplicação dos métodos da programação pela sintaxe e de técnicas de manipulação dinâmica de dados.

Cada projecto é executado em 4 fases, que serão, outros tantos capítulos do relatório final do projecto.

- Clarificação da especificação.
- Análise da sintaxe e das funções semânticas envolvidas.
- Obtenção dos algoritmos e programas.
- Verificação de que a implementação está de acordo com a especificação.

a. Clarificação da especificação:

A clarificação da especificação deve ser o mais sucinta que possível sendo no entanto rigorosa. Gramáticas e expressões regulares poderão ser utilizadas no sentido da obtenção daquelas características.

b. Análise da sintaxe e das funções semânticas

A análise do projecto deve seguir as técnicas introduzidas. O resultado deste trabalho será a obtenção de um modelo, tão rigoroso quanto possível, de todo o programa.

O modelo deve ser expresso em termos de autómatos, expressões regulares e/ou gramáticas. As funções semânticas devem ser isoladas e clarificadas. A linguagem na qual a análise é feita deve ser clara e não esconder pressupostos, nem problemas sem resposta.

c. Obtenção dos algoritmos e programas:

Neste ponto devem-se clarificar e explicitar quais os métodos e regras segundo os quais o modelo é transformado num programa.

d. Verificação de que a implementação está de acordo com a especificação:

Neste ponto devem apresentar-se os argumentos considerados necessários e suficientes para mostrar que a implementação está de acordo com a especificação.

PROJECTOS

1 . Ârvore Geneológica

Objectivo: Construir um programa que reconheça uma linguagem de comandos que permite a definição, modificação e consulta de uma ârvore geneológica de uma dada família.

Os comandos permitem:

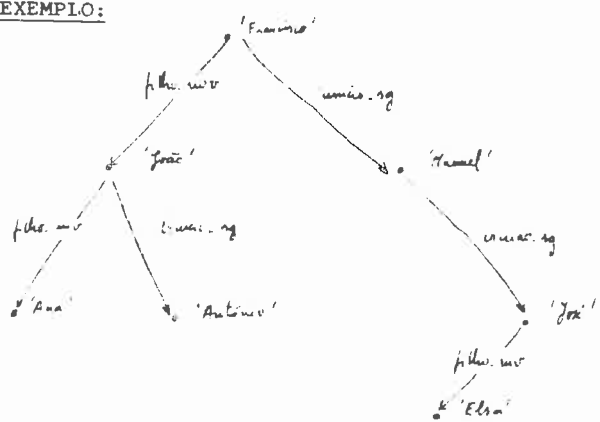
- Dar a família actualmente conhecida.
- Ampliá-la inserindo novos elementos (antepassados e descendentes).
- Verificar se um elemento pertence à família e localizá-lo.
- Contar o número de membros descendentes de um dado elemento.
- Desenhar a ârvore na impressora de linhas.
- Dado um elemento conhecer o nome dos seus parentes mais próximos (Ex: irmão, tios, pai, primos, etc.).

A estruttura de dados básica será:

```
type referência = ↑ pessoa;
     pessoa = Record
                   nome: "string";
                   filho-mv, irmão-sg: referência
              end;
```

Que permite construir a ārvore binãria seguinte:

EXEMPLO:



Onde: Francisco é pai de João e Antõnio, irmão de Manuel e José, avõ de Ana e tio de Elsa. João e Elsa são primos direitos, etc.

Para a definição da linguagem de comandos inspire-se nos exemplos apresentados nos capītulos anteriores a propósito da manipulação de uma ārvore geneológica simplificada.

2. Manipulação interactiva de ficheiros

Fazer um programa interactivo cujas funções possam ser inspiradas no seguinte (hipotético) manual de utilização.

MANUAL DE UTILIZAÇÃO

Sistema de secretariado automático è um programa que permite manipular ficheiros com informações do tipo:

Linha 1
Linha 2
Linha 3
Classificação
Dia Mês Ano

As linhas 1, 2 é 3 são sequências da caracteres quaisquer, eventualmente vazias, com um máximo de 70 caracteres.

A classificação é uma linha contendo sequências de 3 caracteres em que o primeiro destes é diferente de digito e os outros dois são digitos.

Dia, Mês e Ano são 3 inteiros com a data de inserção do item.

Em qualquer das linhas 1, 2 e 3 pode figurar um comentário, isto é uma sequência de caracteres quaisquer envolvidos em parentesis. Não há comentários dentro de comentários nem mais do que um comentário por linha.

1. TIPOS DE COMANDOS

Os comandos deste sistema são de duas espécies:

- Comandos de execução imediata:
  - Manual
  - Sair
  - Listar
  - Explicar
- Comandos de passagem a sub-estados:
  - Inserir
  - Modificar

Que permitem passar a sub-estados com o mesmo nome.

Cada sub-estado reconhece apenas os comandos que lhe são específicos (sub-comandos) e cuja descrição pormenorizada se encontra no ponto 3.

Todos os comandos, sub-comandos e switchs podem ser reconhecidos por abreviatura desde que esta não conduza a confusão.

2. CONVENÇÕES

* Emitido pelo sistema quando aguarda um comando

-- Emitido pelo sistema quando aguarda um sub-comando

(CR) Terminador de comando

3. Especificação dos comandos e dos sub-comandos

O sistema ao ser executado emite o sinal * assinalando que espera um comando do utilizador. Este pode ser alternativamente:

Manual
Explicar
Sair
Modificar
Inserir
Listar

3.1. Comando Manual

Formato: Manual (CR)
Acção: Impressão do manual do utilizador

3.2. Comando inserir

Formato: inserir (CR)
Acção: passagem ao sub-estado de inserção (permite inserir novos itens no ficheiro corrente).

Sub-comandos do estado de inserção

3.2.1. Sub-comando L_1:

Formato: L_1 = sequência de caracteres (CR)
Acção : Define ou redefine a $linha_1$ do item corrente.

3.2.2. Sub-comando L_2:

Formato: L_2 = sequência de caracteres (CR)
Acção: Define ou redefine a $linha_2$ do item corrente.

3.2.3. Sub-comando L_3:

Formato: L_3 = sequência de caracteres (CR)
Acção: Define ou redefine a $linha_3$ do item corrente.

3.2.4. Sub-comando C

Formato: C = sequência de grupos de 3 caracteres (CR)
NB: Em cada grupo o primeiro caracter é ≠ de digito e os dois restantes digitos.
Acção: Define ou redefine a classificação do item corrente.

3.2.5. Sub-comando mostrar:

Formato: Mostrar (CR)

Acção: As linhas 1, 2, 3 e a classificação do item corrente são impressas no terminal.

3.2.6. Sub-comando guardar:

Formato: Guardar (CR)

Acção: O item corrente é acrescentado ao ficheiro em modo 'append".

3.2.7. Sub-comando explicar:

Formato: Explicar (CR)

Acção: Desencadeia a escrita de mensagem:
Encontra-se no estado de inserção e pode utilizar os seguintes sub-comandos:
.L_1 para definir ou redefinir a $linha_1$
.L_2 para definir ou redefinir a $linha_2$
.L_3 para definir ou redefinir a $linha_3$
.C para definir ou redefinir a classificação
.Mostrar para imprimir o item corrente
.Guardar para acrescentar o item corrente
.Sair para regressar ao estado anterior.
.Explicar para obter este esclarecimento.

3.2.8. Sub-comando sair:

Formato: sair (CR)

Acção: O sistema volta ao estado anterior à entrada em inserção.

3.3. Comando modificar:

Formato: Modificar (CR)

Acção: Passagem ao sub-estado de modificação (permite modificar itens do ficheiro corrente).

Sub-comandos de modificar:

3.3.1. Sub-comandos L_1, L_2, L_3, C, mostrar e sair

Idênticos aos sub-comandos com o mesmo nome do estado de inserção.

3.3.2. Sub-comando mover:

Formato: Mover número (CR)

Acção: O item corrente passa a ser aquele que tiver o número interno dado. Só move em ordem ascendente.

3.3.3. Sub-comando trocar:

Formato: Trocar (CR)

Acção: O valor do item corrente no ficheiro passa a ser aquele que estiver em memória.

3.3.4. Sub-comando explicar:

Formato: Explicar (CR)

Acção: Desencadeia a escrita de mensagem:

Encontra-se no estado de modificação e pode utilizar os sub-comandos:

.L_1 para redefinir a $linha_1$
.L_2 para redefinir a $linha_2$
.L_3 para redefinir a $linha_3$
.C para redefinir a classificação
.Mostrar para imprimir o item corrente
.Mover para redefinir o item com um dado número interno.
.Trocar para redefinir o item corrente
.Suprimir para suprimir o item corrente
.Sair para regressar ao estado anterior a modificar.
.Explicar para obter este esclarecimento.

3.3.5. Sub-comando suprimir:

Formato: Suprimir (CR)

Acção: O item corrente é retirado do ficheiro corrente.

3.4. Comando listar:

Formato: Listar/Switches (CR)

Acção Permite listar o conteúdo do ficheiro corrente

Switches:

./A-DD MM AA

Só são listados os itens que tenham dado entrada antes da data indicada. Por defeito torna-se /D - 1 1 1

./D -DD MM AA

Só são listados os itens que tenham dado entrada depois da data indicada. Por defeito torna-se /D - 31 12 99

./Total

A listagem inclui número interno, data de inserção e comentários. Por defeito é não total.

./Formatado

A listagem sai centrada no papel do terminal. Por defeito não.

./Impressão

A listagem é despejada, em modo append, num ficheiro especial do sistema. Por defeito sai no terminal.

./T

A listagem sai antecedida de um título que é constituído pelo comando que lhe deu origem e sua data.

./T - sequência de caracteres

A listagem sai antecedida do título dado.

./C - classificação

Só saiem itens que verifiquem a classificação dada. Por defeito saem todos.

NOTA: Só o Switch /C pode ser repetido com o sentido de "E" lógico.

3.5. Comando explicar:

Formato: Explicar (CR)

Acção: Provoca a escrita da mensagem:

Encontra-se no estado de aceitação de comandos e pode utilizar:

.Manual para obter uma listagem do manual.

.Inserir para modificar, em modo append, os itens do ficheiro.

.Modificar para alterar qualquer item do ficheiro.

.Explicar para obter esta mensagem.

.Sair para indicar o fim de utilização.

3.6.- Comando sair:

Formato: Sair (CR)

Acção: Saída do sistema (Fim da utilização)

B.B: Qualquer comando ou sub-comando admite o switch explicar e deve ser escrito na forma:

Comando/explicar (CR)

ou

Sub-comando/explicar (CR)

Desencadeia a explicação da acção do comando ou sub-comando, bem como a escrita do respectivo formato.

3. <u>Editor interactivo</u>

Fazer um editor interactivo com a seguinte filosofia:

O editor trabalha na base da posição corrente de um ponteiro fictício. Os comandos para o editor permitem:

-Manipular a posição do ponteiro.

-Modificar o texto.

-Verificar o resultado da acção dos comandos interiores.

-Terminar a edição.

a. Manipular a posição do ponteiro:

- S sequência de caracteres (CR)

  Procurar o string que se segue a S e coloca o ponteiro no início da linha que o contém. A busca é feita para a frente.

- L (CR)

  Coloca o ponteiro no início da linha corrente.

- T (CR)

  O ponteiro é colocado no início do ficheiro.

- ± inteiro J (CR)

  Saltar um número "inteiro" de linhas. Para a frente, + pode ser omitido. Se "inteiro" for omitido toma-se o valor 1. O ponteiro fica posicionado no início da nova linha.

b. Modificar o texto:

- K (CR)

  Suprimir a linha corrente. O ponteiro fica posicionado no início da linha seguinte.

- D sequência de caracteres (CR)

  A sequência de caracteres que se segue a D é procurada, percorrendo o ficheiro ascendentemente. Se for encontrada é suprimida e o ponteiro posicionado antes do caracter que se lhe segue. Se não for encontrada o ponteiro fica posicionado no fim do ficheiro. A sequência de caracter tem de existir numa só linha.

- I sequência de caracteres $ (escape).

  A sequência de caracteres é inserida a partir da posição corrente do ponteiro e pode incluir mudanças de linha. O ponteiro fica colocado imediatamente a seguir à sequência inserida.

- R sequência de caracteres$_1$ (CR) sequência de caracteres$_2$ (CR).

  Acção equivalente a:

  D sequência de caracteres$_1$ (CR)
  I sequência de caracteres$_2$ (CR)

e. Verificação

- Q (CR)

  A linha corrente é impressa, sendo a posição corrente do ponteiro assinalada pelo caracter ↑ .

- ±V (CR)

  +V tem por efeito a impressão da linha corrente após a execução de cada comando.. -V simétrico do anterior. + pode ser omitido.

- Inteiro P (CR)

  O número de linhas especificado por "inteiro" é impressa. O ponteiro fica colocado no início da última linha impressa. Este comando anula o efeito de +V. Se "inteiro" for omitido toma-se o valor 1 por defeito.

d. Terminar a edição:

- X (CR).

4. Gestão de Pessoal

Pretende-se construir um programa que actualize um ficheiro de empregados. A especificação dos dados a manipular é dada pelas seguintes declarações:

```
meses = (JAN, FEV, MAR, ABR, MAI, JUN, JUL, AGO, SET, OUT,
         NOV, DEZ);
categ = (EMPREG, TECNIC, ADMINIST);
```

```
empregado = record
                  número: integer;
                  nome, morada: array [1..30] of char;
                  vencimento: real;
                  categoria: categ;
                  dataadmissão: record
                                    dia,ano: integer;
                                    mês: meses
                                end
            end;
```

Cada empregado é acedido pelo seu número.

Através da leitura de um ficheiro contendo os comandos e as modificações desejadas, o programa deverá realizar as operações correspondentes no caso de não ter detectado qualquer erro; senão, ficará obrigado a transmitir o tipo de erro ao utilizador, tentando sempre, desde que de uma forma segura, o tratamento da restante informação não errada.

Os comandos são dados por palavras em português e cujo significado é óbvio:

'SUPRIMIR' —> basta indicar o nº do empregado a suprimir;

'MODIFICAR' —> indicam-se os campos e os seus novos valores para o empregado com o nº apresentado;

'INSERIR' —> necessita da sequência dos valores de todos os campos do novo empregado, iniciando-se com o seu número;

'LISTAR' —> comando que origina a listagem de todos os empregados existentes no momento e por ordem numérica.

Todas as sequências de quaisquer comando terminam obrigatòriamente com 'FIM' e o ficheiro de input com 'STOP'.

O programa deverá ser indiferente ao tipo de formatação do texto do ficheiro de input, do qual se mostra um exemplo:

```
INSERIR  11379  NOME = 'LUIS COSTA'  MORADA
    = 'RUA 3Ø'  VENCIMENTO = 17 ØØØ.ØØ  CATEGORIA =
TECNIC EM 19 JUL 8Ø  FIM MODIFICAR 115  MORADA =
'AVENIDA NOVA, 14' FIM SUPRIMIR 2ØØ1 FIM STOP
```

O ficheiro dos objectos "empregado" terá existência apenas em memória principal e durante a execução do programa, sendo a sua gestão feita por um método de HASHING ou utilizando B-TREES.

No primeiro caso pretende-se o uso de um método de recorrência estrita para geração de índices secundários. A dimensão da tabela de "hash" (assim como a ordem da árvore no segundo caso) deverá figurar como constante, susceptível de ser modificada apenas na sua única declaração.

5. Formatador de texto

Pretende-se implementar um programa PASCAL que auxilie a elaboração de um relatório ou outro documento escrito, no que diz respeito à formatação do texto.

Mediante caracteres de controlo a inserir em determinados pontos do texto inicial, obter-se-á automáticamente uma versão inicial do mesmo com o aspecto gráfico pretendido, como seja por exemplo a numeração das páginas, a identação dos parágrafos e a feitura das margens.

Suponhamos que um determinado aluno pretendia elaborar um seu relatório de trabalho. Não teria mais do que, através de um editor de texto, criar um ficheiro contendo o relatório, não se preocupando muito com a sua apresentação gráfica (margens, parágrafos, etc...) desde que fornecesse os comandos necessários a formatá-lo. De seguida chamaria o programa formatador que receberia esse ficheiro e, sem o destruir, criaria um outro em que o relatório apareceria na sua versão definitiva e isento dos comandos inseridos no texto fonte.

Estes comandos deverão ser reconhecidos pelo programa formatador mediante um caracter inicial que será o ponto ortográfico ('.'), antecedido de pelo menos um espaço ou mudança

de linha. Esse caracter não poderá assim iniciar qualquer palavra que não seja comando, mas poderá estar nela contido (incluindo terminá-la).

Além disso providenciar-se-á um caracter especial ('#') não será interpretado como espaço em branco obrigatório: desta forma a escrita de '#.' não será interpretada como comando mas sim como a palavra 'ƀ.'.

Sugere-se que o nome identificador dos comandos seja sempre construído com poucos caracteres, escolhidos de maneira a servirem de memória à acção em causa. Os parâmetros, caso os haja, seguir-se-ão com um espaço em branco de intervalo.

O formatador deverá reconhecer os seguintes comandos (de que se apresentam sugestões mnemónicas) necessários às tarefas que se descrevem:

.LP

Fixação do número de linhas por página.

.RM

Comprimento da linha de texto (ie, definição da margem direita).

.LM

Número de espaços em branco à esquerda na linha de texto (ie, definição da margem esquerda).

.NP

Início da numeração das páginas ao canto superior direito (primeira linha). O número inicial é dado como parâmetro.

.P

Início de parágrafo, com indicação do número de espaços em branco em relação à margem esquerda. Tal como os comandos anteriores e caso não seja dada ordem em contrário, o parâmetro numérico manter-se-á válido até ao final do texto, devendo existir também uma opção por defeito.

.B

Salto imediato de um determinado número de linhas, deixadas em branco.

.J

Escrita de palavras sucessivas do texto inicial até se encontrar a margem direita. Não haverá palavras interrompidas por mudança de linha. Como se pretende alinhamento do texto tanto à esquerda como à direita, o formatador deverá então gerir os espaços entre palavras para que a última alinhe com a margem direita (no caso de linhas completas).

A acção produzida por este comando será tomada como opção por defeito.

.NJ

Cancelamento das acções originadas pelo comando anterior. As margens continuarão a ser respeitadas, mas as linhas de texto aparecerão tal e qual como no ficheiro inicial.

.PG

Mudança obrigatória de página.

.NNP

Não numeração das páginas seguintes. A acção correspondente será a opção por defeito. Estes 4 últimos comandos não possuem parâmetros.

.C

Escrita de texto que, não excedendo o tamanho de uma linha, se pretenda centrado(em relação às margens). O texto deverá ser parentizado.

Usar-se-á um terceiro caracter especial ('-') que obriga à escrita do caracter seguinte sem ser interpretado.

O aparecimento de incoerências que forem detectadas no ficheiro fonte não interromperá a execução do programa formatador, mas originará, no ficheiro de saída e no terminal, uma mensagem de erro adequada.

6. Árvore estática das declarações

Fazer um programa para determinar a árvore estática das declarações de procedimentos e funções de um programa Pascal sintácticamente correcto.

O programa tem como entrada um Programa Pascal, e como saída:

<u>Árvore de declarações de procedimentos e funções do programa id</u>

id
  id_{11}
  id_{12}
    id_{121}
    id_{122}
  id_{13}
    id_{131}
      id_{1311}
  id_{14}

Onde id é o identificador do programa, id_{1i} os identificadores dos procedimentos e funções declarados no seu primeiro nível. id_{12j} os identificadores dos procedimentos ou funções locais ao procedimento ou função cujo identificador é id_{12}, etc.

O programa deve permitir parametrizar, por recompilação, o número de caracteres que toma em cada identificador. O número de caracteres da margem esquerda da saída e o número de caracteres que cada nível desloca a saída para a direita.

Se por qualquer motivo a execução do programa não poder prosseguir, esse facto é assinalado no terminal do utilizador por uma mensagem adequada.

7. <u>Grafo de chamadas de procedimentos ou funções</u>

Fazer um programa com a seguinte especificação:

A entrada é constituída por um programa Pascal sintácticamente correcto, a saída tem o aspecto:

Grafo das chamadas de procedimentos ou funções de id (*-recursivo/a)

id

id_1
id_2
id_5

id_1 (*)

id_3

id_2

id_1

.
.
.

Onde id_i é o identificador de um procedimento ou função declarado no programa e id o identificador do programa. A cada id_i (e a id) seguem-se os identificadores de funções ou procedimentos chamados pelo procedimento ou função cujo identificador é id_i (ou id). Um lacete no grafo é assinalado com (*) no respectivo nó.

Em seguida calcula o fecho reflexo transitivo do grafo de chamadas e faz sair o grafo na forma:

Fecho reflexo transitivo do grafo de chamadas de id (*-recursividade imbricada potencial)

id

id_1
id_2
id_3
id_4
id_5
id_7

id_1 (*)

id_3
id_2
id_1
id_4

id_2 (*)

.
.
.
.

Se por qualquer motivo, a execução do programa não poder prosseguir, esse facto é assinalado no terminal do utilizador por uma mensagem adequada.

O programa deve também permitir parametrizar (por recompilação) todas as constantes que condicionam o formato gráfico das saídas, assim como o número máximo de nós e de arcos dos grafos. Se por qualquer motivo a execução não poder prosseguir, esse facto é assinalado no terminal do utilizador por uma mensagem adequada.

8. Tabela de referências a símbolos de um programa Pascal

Fazer um programa que tem por entrada um programa Pascal sintácticamente correcto e por saída uma tabela constituída por todos os identificadores existentes no programa inicial, listados por ordem alfabética e com indicação do (s) numero(s) da(s) linha(s) em que cada figura.

Nesta tabela de 'Cross Reference', cada ocorrência de um identificador deverá ser acompanhado também do respectivo número de 'scope' (estático) do programa: quando da declaração de determinados objectos (é o caso de procedimentos, funções e Record's), esse valor é incrementado de uma unidade, sendo decrementado de igual valor à saída dos mesmos.

O programa deve ser parametrizável de forma a permitir redefinir o conjunto de identificadores que não são listados (tal passa-se à partida para as palavras chaves).

Um outro ficheiro, a ser fornecido como saída, conterá o programa fonte com as linhas numeradas.

Se por qualquer motivo, a execução do programa não poder prosseguir, esse facto é assinalado no terminal do utilizador por uma mensagem adequada.

9. Formatador automático de programas

Fazer um programa que tem por entrada um programa Pascal sintácticamente correcto e por saída, exactamente o mesmo programa com uma disposição gráfica que ponha em evidência a sua estruturação.

Para além de pôr em evidência as estruturas de dados, de declarações e de controle, o programa põe em evidência as componentes elementares do programa ("tokens" da linguagem) separando-as por pelo menos 1 espaço.

Opcionalmente, o programa permite que o tipo de impressão usada nas palavras chave e no restante programa, possa ser distiguída. Em qualquer hipótese a estrutura dos comentários não deve ser destruida.

Por recompilação, é possível parametrizar a disposição gráfica do programa formatado, assim como o número máximo de caracteres dum identificador ou de um "token" elementar.

Se por qualquer motivo a execução do programa não poder prosseguir, esse facto é assinalado no terminal do utilizador por uma mensagem adequada.