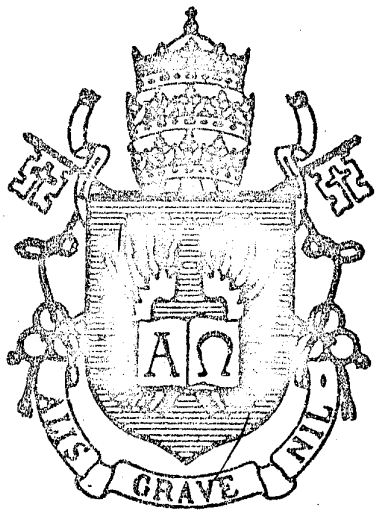


PUC

Série: Monografias em Ciência da Computação
Nº11/88

MANUAL DE OPERAÇÃO DO SISTEMA DE GERAÇÃO DE ANALISADORES SINTÁTICOS R*S SIMPLES

José Lucas Mourão Rangel Netto

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

UA MARQUÊS DE SÃO VICENTE, 225 – CEP 22453

RIO DE JANEIRO – BRASIL

PUC/RJ - Departamento de Informática

Série: Monografias em Ciência da Computação, no. 11/88
Editor: Paulo A. S. Veloso Outubro, 1988

Manual de operação do sistema de geração de analisadores sintáticos R*S simples

José Lucas Mourão Rangel Netto

- Este trabalho foi parcialmente financiado pelo MCT.

In charge of publications:

Rosane T. L. Castilho

Assessoria de Biblioteca, Documentação e Informação
PUC/RJ - Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
BRASIL

Abstract

This paper presents the necessary information for the use of a parser generator which is based on the **simple** version of the **R*S** parsing method. The generator itself has been in use for four years, and has been shown as a flexible and powerful tool for use in the construction of compilers and similar software. In order to be self-contained, this paper includes a short presentation of the theory of the method, and general description of the implementation. We have also included information on practical ways of ambiguity removal, and conflict resolution and on the attachment of semantic actions to syntax rules in **simple R*S** grammars.

A companion program which generates tables in compact form is also presented. This last program generates code which may be directly inserted in Turbo Pascal programs. Some small modifications would be required, in the case of other languages.

Keywords: parsing, parser generators, **R*S** parsing, **simple R*S** parsing, compiler construction.

Resumo:

Este trabalho apresenta a informação necessária para o uso do gerador de analisadores sintáticos baseado na versão **simples** do método de análise sintática **R*S**. O gerador propriamente dito tem sido usado há quatro anos, e se mostrou uma ferramenta flexível e poderosa para auxiliar a construção de compiladores e de "software" similar. Para que o documento seja auto-contido, incluímos uma apresentação curta da teoria do método e uma descrição sumária da implementação. Incluímos também informação sobre maneiras práticas de remoção de ambiguidades e resolução de conflitos e sobre a associação de ações semânticas às regras sintáticas em gramáticas **R*S simples**.

Apresentamos também um programa associado que gera tabelas em forma compactada. Este programa gera código que pode ser inserido diretamente em programa Turbo Pascal, ou, com pequenas alterações, em programas escritos em outras linguagens.

Palavras-chave: análise sintática, geradores de analisadores sintáticos, analisadores **R*S**, analisadores **R*S simples**, construção de compiladores.

Indice:

I. Introdução

Este manual visa possibilitar a um usuário sem prévio conhecimento do sistema R*S a construção de analisadores sintáticos R*S simples (R*Ss) através do uso dos programas gerador e compactador. Não se prevê nenhum conhecimento anterior sobre a teoria da análise sintática R*S (**[Schn87]**), mas espera-se que o usuário esteja familiarizado com os métodos de análise sintática LR, em particular o método sLR, por exemplo, como apresentado em **[AhSU86]**. Além desta introdução (**seção I**) temos a seguinte distribuição:

A **seção II** faz uma descrição breve do método de análise R*S simples, sem nenhuma preocupação de apresentação teórica precisa. Essa descrição visa apenas permitir o uso do sistema, e a necessária compreensão para a eventual alteração de uma gramática que se revele não apropriada para a aplicação ou para o método, ou ainda a alteração em alguns pontos do algoritmo básico de análise. Um outro documento deverá apresentar a teoria da versão "simples" do método R*S, que não faz parte de **[Schn87]**.

As **seções III e IV** mostram como os programas gerador e compactador podem ser utilizados, e, em particular, os formatos dos arquivos de entrada e de saída e o significado das mensagens apresentadas.

A **seção V** apresenta algumas considerações finais, entre as quais destacamos:

(a) técnicas de alteração de gramáticas para resolução de conflitos;

(b) formas de introdução de semântica nas regras simples, nos raros casos em que isto se torna necessário;

(c) algumas limitações da implementação e as formas de contorná-las, sem perder a eficiência do programa gerado.

A conclusão da seção se faz com a apresentação de um exemplo de porte semelhante aos encontrados durante a prática de implementação de compiladores: demonstramos a construção de um analisador sintático, para um dialeto de Pascal.

Este sistema foi construído pelo autor, com a colaboração de seus alunos, entre os quais citamos em particular Sérgio de Mello Schneider, Paulo Eduardo Laurenz Buchsbaum e Oliver Barreira Ponte. Agradecemos também a colaboração de Clovis Torres Fernandes, e dos usuários das versões anteriores, em particular dos alunos do curso Compiladores I do Departamento de Informática da PUC-RJ.

II. Breve descrição do método R*S simples

II.1 Introdução

O método R*Ss foi desenvolvido com a intenção de, ao mesmo tempo,

- permitir uma aceleração do processo de análise, eliminando o tempo gasto nas reduções pelas chamadas regras simples,
- tornar possível uma recuperação de erro mais simples e mais precisa,
- manter o tamanho das tabelas da mesma ordem de grandeza de métodos considerados práticos, como sLR(1) e laLR(1),
- permitir que as gramáticas de linguagens de programação sejam usadas praticamente sem alterações, facilitando a tarefa de programação da "semântica".

Para que estes objetivos fossem atingidos, foi necessário alterar a informação necessária para decidir pela propriedade de uma ação de redução: essa decisão é tomada consultando, além do estado no topo da pilha, e do símbolo à frente ("lookahead"), o estado a ser descoberto se o proposto "handle" for removido. Por essa razão, a construção das tabelas R*Ss é mais demorada do que no caso dos métodos citados acima.

Para demonstrar o funcionamento do método de análise R*S simples, e a maneira pela qual as tabelas são construídas, vamos utilizar a gramática exemplo **Gs**, apresentada a seguir. Essa gramática já é a gramática aumentada, com a regra adicional que serve, nos métodos LR e semelhantes, para indicar a parada. O leitor notará que a regra inicial é ligeiramente distinta daquela usada no método sLR(1): o símbolo **$** representa aqui início e fim de arquivo, e aparece explícitamente na regra inicial. O símbolo **!** é usado para separar as diversas alternativas, e é usualmente lido "ou".

Gramática Gs

```
 0. S'= $ C $
 1. C = [ L ]
 2.   ! V := E
 3. L = L ; C
 4.   ! C
 5. E = E + T
 6.   ! T
 7. T = T * F
 8.   ! F
 9. F = ( E )
10.   ! V
11. V = a
```

II.2 Construção dos estados do analisador

O estado inicial é formado pelo fechamento ("closure") de um conjunto de ítens que contém apenas o ítem inicial, que no nosso

caso é **S'= $.C $.** O fechamento é feito pelo mesmo processo usado no caso sLR: para cada símbolo não-terminal que apareça precedido pelo ponto em algum ítem do estado, todos os ítens iniciais correspondentes às regras do mesmo não-terminal devem ser acrescentados.

A construção do estado inicial, no caso do exemplo, prossegue como abaixo:

- o ítem inicial **S'= $.C $.** é incluído.
- como nesse ítem existe um ponto antes de uma ocorrência do não-terminal **C**, os ítens iniciais das regras de C devem ser acrescentados: **C = .[L]** e **C = .V := E** , ou de forma abreviada,

```
C = .[ L ]
  ! .V := E
```

- temos agora o ponto na frente do não-terminal **V**. O único ítem inicial de **V**, **V = .a**, deve ser acrescentado, portanto.

A composição final do estado inicial (Estado **0**) é então:

```
0. S'= $.C $
   C = .[ L ]
     ! .V := E
   V = .a
```

Devemos agora construir a classe de estados acessíveis a partir do estado **0**. Isso será feito de forma semelhante à forma usada no caso sLR. A alteração é devida ao fato de que o método R*S ignora as reduções por regras simples, ou seja, as regras da forma **A = B**, onde **A** e **B** são não-terminais. Por essa razão, se o processo de cálculo da função de transição entre os estados nos levar a um ítem simples completo, (isto é um ítem de regra simples com o ponto na última posição) esse ítem não será incluído no novo estado.

Esse processo de construção da coleção dos estados pode ser aplicado ao exemplo. Veremos aqui apenas alguns casos. Por exemplo, a partir do Estado **0**, com o símbolo **[**, atingiremos um estado (Estado **1**) cuja composição, após o correspondente fechamento é:

```
1. C = [.L ]
   L = .L ; C
     ! .C
   C = .[ L ]
     ! .V := E
   V = .a
```

A partir deste estado são atingidos os Estados **5**, **4**, **2**, com os símbolos **L**, **V**, **a**. Com o símbolo **[** o estado atingido é novamente o estado **1**; note que o símbolo **C** não leva a nenhum estado, uma vez que o ítem **L = .C** nos levaria, através do símbolo **C** a **L = C.**, um ítem simples completo.

```
5. C = [ L.]          4. C = V.:=E          2. V = a.
   L = L.; C
```

Observação. A numeração dos estados é irrelevante. Entretanto, para facilidade de referência futura, a numeração dos estados usada acima é a mesma usada pelo gerador.

Se a construção acima for aplicada em todos os casos, até que nenhum outro estado novo possa ser criado, teremos o conjunto de estados R*S do analisador correspondente à gramática considerada. O resultado final é o seguinte:

Estados:

```
0. S'= $.C $
   C = .[ L ]
     ! .V := E
   V = .a

1. C = [.L ]
   L = .L ; C
     ! .C
   C = .[ L ]
     ! .V := E
   V = .a

2. V = a.

3. S'= $ C.$

4. C = V.:= E

5. C = [ L.]
   L = L.; C

6. S'= $ C $.

7. C = V :=.E
   E = .E + T
     ! .T
   T = .T * F
     ! .F
   F = .( E )
     ! V
   V = .a

8. C = [ L ].

9. L = L ;.C
   C = .[ L ]
     ! .V := E
   V = .a

10. F = (.E )
    E = .E + T
      ! .T
    T = .T * F
      ! .F
    F = .( E )
      ! .V
    V = .a

11. C = V := E.
    E = E.+ T

12. T = T.* F

13. L = L ; C.

14. E = E.+ T
    F = ( E.)

15. E = E +.T
    T = .T * F
    T = .F
    F = .( E )
      ! .V
    V = .a

16. T = T *.F
    F = .( E )
      ! .V
    V = .a

17. F = ( E ).

18. E = E + T.
    T = T.* F

19. T = T * F.
```

As transições entre estados estão indicadas na tabela abaixo, que define a função de transição; os casos em que esta função tem o valor indefinido são indicados por "-". O acesso a um desses valores indefinidos indica a presença de um erro, se, adicionalmente, uma ação de redução não for aplicável ao caso.

A última coluna da mesma tabela indica as ações de redução possíveis em cada estado, reconhecida pela presença de itens (não-simples) completos.

Tabela de transições

Est.	Transições $	[	]	:=	;	+	*	(	)	a	C	L	E	T	F	V	Reduções
0	–	1	–	–	–	–	–	–	–	2	3	–	–	–	–	4	–
1	–	1	–	–	–	–	–	–	–	2	–	5	–	–	–	4	–
2	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	red 11
3	6	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
4	–	–	–	7	–	–	–	–	–	–	–	–	–	–	–	–	–
5	–	–	8	–	9	–	–	–	–	–	–	–	–	–	–	–	–
6	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	red 0
7	–	–	–	–	–	–	–	10	–	2	–	–	11	12	–	–	–
8	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	red 1
9	–	1	–	–	–	–	–	–	–	2	13	–	–	–	–	4	–
10	–	–	–	–	–	–	–	10	–	2	–	–	14	12	–	–	–
11	–	–	–	–	–	15	–	–	–	–	–	–	–	–	–	–	red 2
12	–	–	–	–	–	–	16	–	–	–	–	–	–	–	–	–	–
13	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	red 3
14	–	–	–	–	–	15	–	–	17	–	–	–	–	–	–	–	–
15	–	–	–	–	–	–	–	10	–	2	–	–	–	18	–	–	–
16	–	–	–	–	–	–	–	10	–	2	–	–	–	–	19	–	–
17	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	red 9
18	–	–	–	–	–	–	16	–	–	–	–	–	–	–	–	–	red 5
19	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	red 7

II.3 Tratamento das ações de empilhamento de terminais

Como o empilhamento de não-termirais é feito automaticamente, no método R*Ss, como parte da ação de redução, não há necessidade do uso de uma tabela de transições com símbolos não-terminais. Para o caso dos terminais, as transições são as indicadas na tabela de empilhamento, mostrada abaixo para a gramática Gs do exemplo.

Tabela de empilhamento

Estado	Terminal $	[	]	:=	;	+	*	(	)	a
0,1,9	–	1	–	–	–	–	–	–	–	2
3	6	–	–	–	–	–	–	–	–	–
4	–	–	–	7	–	–	–	–	–	–
5	–	–	8	–	9	–	–	–	–	–
7,10,15,16	–	–	–	–	–	–	–	10	–	2
11	–	–	–	–	–	15	–	–	–	–
12,16	–	–	–	–	–	–	16	–	–	–
14	–	–	–	–	–	15	–	–	17	–

A tabela de empilhamento pode ser armazenada de forma compactada, uma vez que é bastante esparsa, tipicamente. Nessa ocasião podemos levar em consideração as linhas iguais, e as

linhas com elementos em comum. A forma pela qual esta compactação é feita está discutida na **seção IV**, onde se discute o uso do programa compactador.

II.4 Tratamento das ações de redução

As ações de redução devem ser tentadas quando o estado **q** no topo da pilha contém um ítem (não-simples) completo, por exemplo, **B = beta.**. Inicialmente, verificamos qual é o estado **p** descoberto pela retirada do "handle" **beta** da pilha, ou melhor, pela retirada dos estados correspondentes aos símbolos de **beta**, em número igual ao seu comprimento. De posse dos estados **q**, **p**, e do símbolo de "lookahead" **s**, decidimos qual será o estado **r** a ser empilhado. Neste caso, o estado **r** refletirá o empilhamento de um não-terminal **A**, obtido a partir de **beta** por redução não-simples para **B**, seguida de zero ou mais reduções simples, de **B** para **A**.

Vamos mostrar a construção da tabela de redução, que fornece o valor de **r** a partir dos valores de **q**, **s**, e **p**, no caso da gramática **Gs** apresentada anteriormente. Uma vez que a tabela de redução tem três entradas, representaremos os valores da tabela, à medida que forem sendo encontrados, através de quádruplas da forma **[q, s, p, r]**.

No exemplo, os estados que permitem reduções são **2**, **8**, **11**, **13**, **17**, **18**, e **19**, sendo as reduções, respectivamente, pelas regras **11**, **1**, **2**, **3**, **9**, **5**, **7**. Em cada estado, neste exemplo, temos apenas a possibilidade de redução por uma única regra; o caso em que há reduções múltiplas é discutido entre os "tópicos avançados" da **seção V**, pela pouca freqüência com que ocorre em casos práticos.

Note-se que não consideramos o estado **6** um estado de redução, uma vez que a regra correspondente, a regra **0**, é a regra acrescentada para sinalizar o término do processo de análise, reconhecido através do empilhamento do estado **6**.

Iniciando então com o caso q = 2, [illegible] regra **V = a**. Verificamos, então, que o não-terminal V pode ser reduzido, através de zero ou mais aplicações das regras simples, para V, F, T, ou E.

A determinação dos valores possíveis de **p** [illegible] tabela [illegible] de transição no sentido inverso: os estados **p** a partir dos quais se pode alcançar o estado **q = 2** com **beta = a** são os estados **p = 0**, **1**, **7**, **9**, **10**, **15**, e **16**.

Examinamos inicialmente o caso **p = 0**. O estado **0** admite transições com os não-terminais **C** e **V**. **C** não nos interessa, porque não pode gerar **V**, o não-terminal do lado esquerdo da regra considerada, através de regras simples, de forma que o único caso a considerar é o caso **V**, com o qual o estado **r = 4** pode ser alcançado. Como **r = 4** não é um estado de redução, a única continuação possível é com o empilhamento do símbolo **s = :=**. Assim, anotamos que para **q = 2**, **s = :=**, **p = 0**, a ação correta é o empilhamento de **r = 4**, refletindo a redução de **a** para **V**. Isto

pode ser indicado pela quádrupla [2, :=, 0, 4].

A situação do caso p = 0 se repete para p = 1, e p = 9, sendo [2, :=, 1, 4] e [2, :=, 9, 4] as quádruplas correspondentes. Para o caso p = 7, temos transições definidas com os não-terminais E e T, sendo r = 11 e r = 12 as opções a considerar. O caso r = 12 é mais simples, uma vez que não é um estado de redução, e o único símbolo s possível é * (quádrupla [2, *, 7, 12]). No caso do estado 11, temos de considerar o símbolo + para empilhamento, e os símbolos que permitam a redução indicada no primeiro ítem de 11. Usaremos aqui, de forma semelhante ao que se faz no método sLR, o conjunto follow[C] de símbolos seguidores de C como uma estimativa do conjunto de símbolos para os quais a ação de redução será apropriada. Assim, como follow[C] = { $,], ; }, escolheremos r = 11 para s = +, $,], ;. As quádruplas correspondentes são [2, +, 7, 11], [2, $, 7, 11], [2,], 7, 11], e [2, ;, 7, 11].

A construção das demais entradas da tabela de redução é semelhante. Note que aceitamos uma ação de redução quando após a redução considerada,

ou o símbolo s pode ser empilhado

ou o símbolo s é um símbolo apropriado para uma redução adicional (verificado através do conjunto follow correspondente).

O resultado da construção completa pode ser representado através das quádruplas abaixo:

Quádruplas de redução [q, s, p, r] para a gramática Gs

[2, $, 7, 11]	[2, $, 15, 18]	[2, $, 16, 19]
[2,], 7, 11]	[2,], 15, 18]	[2,], 16, 19]
[2, ;, 7, 11]	[2, ;, 15, 18]	[2, ;, 16, 19]
[2, :=, 0, 4]	[2, :=, 1, 4]	[2, :=, 9, 4]
[2, +, 7, 11]	[2, +, 10, 14]	[2, +, 15, 18]
[2, +, 16, 19]	[2, *, 7, 12]	[2, *, 10, 12]
[2, *, 15, 18]	[2, *, 16, 19]	[2,), 10, 14]
[2,), 15, 18]	[2,), 16, 19]	
[8, $, 0, 3]	[8,], 1, 5]	[8,], 9, 13]
[8, ;, 1, 5]	[8, ;, 9, 13]	
[11, $, 0, 3]	[11,], 1, 5]	[11,], 9, 13]
[11, ;, 1, 5]	[11, ;, 9, 13]	
[13,], 1, 5]	[13,], 9, 13]	[13, ;, 1, 5]
[13, ;, 9, 13]		
[17, $, 7, 11]	[17, $, 15, 18]	[17, $, 16, 19]
[17,], 7, 11]	[17,], 15, 18]	[17,], 16, 19]
[17, ;, 7, 11]	[17, ;, 15, 18]	[17, ;, 16, 19]
[17, +, 7, 11]	[17, +, 10, 14]	[17, +, 15, 18]
[17, +, 16, 19]	[17, *, 7, 12]	[17, *, 10, 12]
[17, *, 15, 18]	[17, *, 16, 19]	[17,), 10, 14]
[17,), 15, 18]	[17,), 16, 19]	
[18, $, 7, 11]	[18, $, 15, 18]	[18, $, 16, 19]
[18,], 7, 11]	[18,], 15, 18]	[18,], 16, 19]

```
[ 18,  ;,  7, 11 ]  [ 18,  ;, 15, 18 ]  [ 18,  ;, 16, 19 ]
[ 18, :=,  0,  4 ]  [ 18, :=,  1,  4 ]  [ 18, :=,  9,  4 ]
[ 18,  +,  7, 11 ]  [ 18,  +, 10, 14 ]  [ 18,  +, 15, 18 ]
[ 18,  +, 16, 19 ]  [ 18,  *,  7, 12 ]  [ 18,  *, 10, 12 ]
[ 18,  *, 15, 18 ]  [ 18,  *, 16, 19 ]  [ 18,  ), 10, 14 ]
[ 18,  ), 15, 18 ]  [ 18,  ), 16, 19 ]
[ 19,  $,  7, 11 ]  [ 19,  $, 15, 18 ]  [ 19,  $, 16, 19 ]
[ 19,  ],  7, 11 ]  [ 19,  ], 15, 18 ]  [ 19,  ], 16, 19 ]
[ 19,  ;,  7, 11 ]  [ 19,  ;, 15, 18 ]  [ 19,  ;, 16, 19 ]
[ 19, :=,  0,  4 ]  [ 19, :=,  1,  4 ]  [ 19, :=,  9,  4 ]
[ 19,  +,  7, 11 ]  [ 19,  +, 10, 14 ]  [ 19,  +, 15, 18 ]
[ 19,  +, 16, 19 ]  [ 19,  *,  7, 12 ]  [ 19,  *, 10, 12 ]
[ 19,  *, 15, 18 ]  [ 19,  *, 16, 19 ]  [ 19,  ), 10, 14 ]
[ 19,  ), 15, 18 ]  [ 19,  ), 16, 19 ]
```

Alternativamente, podemos representar a mesma informação, de forma mais legível, através de uma tabela que fornece o valor de **r** a partir das três entradas **q**, **s**, e **p**. Para diminuir o tamanho ocupado no texto pela tabela e facilitar sua consulta, em alguns casos, as entradas com valores repetidos foram reunidas. Esta tabela aparece na página seguinte.

A tabela de redução pode também ser armazenada de forma compactada, uma vez que é bastante esparsa, como tipicamente ocorre. Uma das formas pela qual esta compactação pode ser feita está discutida na **seção IV**, onde se discute o método usado pelo programa compactador.

II.5 Tabelas adicionais

Para a obtenção do valor de **p**, é necessário conhecer o comprimento do lado direito da regra pela qual seria feita a redução não-simples; no caso estudado aqui, este valor é determinado apenas pelo estado **q**, e é chamado de salto. Uma tabela, que fornece para cada estado de redução o valor do salto correspondente, deve ser acrescentada às tabelas já vistas para que a análise sintática possa ser realizada.

Uma outra tabela adicional não é estritamente necessária para a análise sintática, mas é fundamental para qualquer aplicação prática do analisador: a tabela que fornece para cada estado de redução o número da regra não-simples pela qual a redução é feita. Esta informação permite a chamada da rotina semântica associada à regra, e é a forma pela qual um analisador sintático controla um processo de compilação ou de tradução em geral.

Tabela de redução para a gramática Gs

q	s	p	r
2,18,19	$,],;	7	11
		15	18
		16	19
	:=	0,1,9	4
	+	7	11
		10	14
		15	18
		16	19
	*	7,10	12
		15	18
		16	19
	)	10	14
		15	18
		16	19
8,11	$	0	3
	],;	1	5
		9	13
13	],;	1	5
		9	13
17	$,],;	7	11
		15	18
		16	19
	+	7	11
		10	14
		15	18
		16	19
	*	7,10	12
		15	18
		16	19
	)	10	14
		15	18
		16	19

Para o nosso exemplo as duas tabelas que indicam o salto e a regra pela qual se faz a redução estão apresentadas na página seguinte. Note que o estado 6, o estado que indica o fim da análise, não foi incluído nessas tabelas, pela razão já vista: a redução pela regra 0 nunca chega a ser realizada.

Estado	Salto
2	1
8	3
11	3
13	3
17	3
18	3
19	3

Estado	Regra
2	11
8	1
11	2
13	3
17	9
18	5
19	7

II.6 Um exemplo de análise

Mostraremos a seguir como funciona o método de análise R*Ss. O algoritmo básico de análise envolve uma pilha, onde são guardados os estados correspondentes ao prefixo viável já obtido. Essa pilha se encontra inicialmente vazia, e pode ser manipulada pelos procedimentos **push**, **pop** e **top**. Suporemos que **push(q)** empilha o estado q no topo da pilha, **pop(n)** retira **n** estados do topo da pilha, e **top** é uma função que devolve o valor do estado correntemente no topo da pilha. As constantes **estinicial** e **estfinal** representam os estados inicial e final, respectivamente. O procedimento **scan** atualiza a variável **s** com o valor do próximo símbolo da entrada, ou com o valor **$** se a entrada estiver terminada.

O algoritmo fundamental tem a forma abaixo:

```
push(estinicial);
scan;
repeat
  q:=top;
  if   "tabela empilha indica transição com s de q para p"
  then begin
    push(p);
    scan;
  end else begin
    pop(salto(q));
    p:=top;
    if   "tabela de redução contem quádrupla [q, s, p, r]"
    then begin
      push(r);
      { incluir aqui ação semântica associada à regra(q) }
    end else
      erro; { anunciar ou reparar erro sintático }
  end;
until top=estfinal;
```

As posições em que a ação semântica ou a ação de tratamento de erro sintático podem ser tomadas se encontram indicadas no algoritmo. Os trechos que representam consulta às tabelas do analisador serão discutidos em conjunção com as técnicas de compactação utilizadas (**seção IV**).

Vamos agora mostrar, passo a passo, a análise da sequência de entrada **$ [a := a + a ; a := (a * a) + a] $.** O funcionamento do analisador está indicado através do conteúdo da pilha a cada instante, e da porção ainda não tratada da entrada. Como é habitual, símbolos terminais e não-terminais na pilha indicam o prefixo viável construído. A última coluna indica, no caso de redução, a quádrupla correspondente, sendo as demais ações de empilhamento. Isto vale inclusive para a última linha, em que o "fim-de-arquivo" **$** é simbolicamente empilhado, sinalizando a conclusão com sucesso do processo de análise.

```
$ 0                          [ a := a + a ; a := ( a * a ) + a ] $
$ 0 [ 1                        a := a + a ; a := ( a * a ) + a ] $
$ 0 [ 1 a 2                      := a + a ; a := ( a * a ) + a ] $ (1)
$ 0 [ 1 V 4                      := a + a ; a := ( a * a ) + a ] $
$ 0 [ 1 V 4 := 7                    a + a ; a := ( a * a ) + a ] $
$ 0 [ 1 V 4 := 7 a 2                  + a ; a := ( a * a ) + a ] $ (2)
$ 0 [ 1 V 4 := 7 E 11                 + a ; a := ( a * a ) + a ] $
$ 0 [ 1 V 4 := 7 E 11 + 15              a ; a := ( a * a ) + a ] $
$ 0 [ 1 V 4 := 7 E 11 + 15 a 2            ; a := ( a * a ) + a ] $ (3)
$ 0 [ 1 V 4 := 7 E 11 + 15 T 18           ; a := ( a * a ) + a ] $ (4)
$ 0 [ 1 V 4 := 7 E 11                     ; a := ( a * a ) + a ] $ (5)
$ 0 [ 1 L 5                               ; a := ( a * a ) + a ] $
$ 0 [ 1 L 5 ; 9                             a := ( a * a ) + a ] $
$ 0 [ 1 L 5 ; 9 a 2                            := ( a * a ) + a ] $ (6)
$ 0 [ 1 L 5 ; 9 V 4                            := ( a * a ) + a ] $
$ 0 [ 1 L 5 ; 9 V 4 := 7                          ( a * a ) + a ] $
$ 0 [ 1 L 5 ; 9 V 4 := 7 ( 10                       a * a ) + a ] $
$ 0 [ 1 L 5 ; 9 V 4 := 7 ( 10 a 2                     * a ) + a ] $ (7)
$ 0 [ 1 L 5 ; 9 V 4 := 7 ( 10 T 12                    * a ) + a ] $
$ 0 [ 1 L 5 ; 9 V 4 := 7 ( 10 T 12 * 16                 a ) + a ] $
$ 0 [ 1 L 5 ; 9 V 4 := 7 ( 10 T 12 * 16 a 2               ) + a ] $ (8)
$ 0 [ 1 L 5 ; 9 V 4 := 7 ( 10 T 12 * 16 F 19              ) + a ] $ (9)
$ 0 [ 1 L 5 ; 9 V 4 := 7 ( 10 E 14                        ) + a ] $
$ 0 [ 1 L 5 ; 9 V 4 := 7 ( 10 E 14 ) 17                     + a ] $ (10)
$ 0 [ 1 L 5 ; 9 V 4 := 7 E 11                               + a ] $
$ 0 [ 1 L 5 ; 9 V 4 := 7 E 11 + 15                            a ] $
$ 0 [ 1 L 5 ; 9 V 4 := 7 E 11 + 15 a 2                          ] $ (11)
$ 0 [ 1 L 5 ; 9 V 4 := 7 E 11 + 15 T 18                         ] $ (12)
$ 0 [ 1 L 5 ; 9 V 4 := 7 E 11                                   ] $ (13)
$ 0 [ 1 L 5 ; 9 C 13                                            ] $ (14)
$ 0 [ 1 L 5                                                      ] $
$ 0 [ 1 L 5 ] 8                                                    $ (15)
$ 0 C 3                                                            $
$ 0 C 3 $ 6
```

As quádruplas usadas para as reduções estão indicadas abaixo pelos números entre parênteses. A seguir, vamos examinar alguns casos com um pouco mais de detalhe.

Na primeira linha, temos no topo da pilha o estado **q = 0**, o estado inicial, e temos como símbolo a ser considerado o símbolo **s = [.** Pela tabela de empilhamento, vista na **seção II.3**, é possível o empilhamento, e o estado resultante é o estado **1.**

Na segunda linha, portanto, o estado do topo é **q = 1**, e o novo símbolo a ser considerado é **s = a**, e novamente a ação é de empilhamento, e o estado resultante é o estado 2.

Na terceira linha, com **q = 2**, e **s = :=**, nenhum estado está indicado na tabela de empilhamento, e devemos, portanto, verificar se uma ação de redução é apropriada. Para isso verificamos na tabela de salto da **seção II.5** que o valor associado ao estado **2** é **1**, de forma que o estado **p = 1** é obtido retirando um estado da pilha. (Lembramos que os símbolos apenas aparecem como comentários, e não precisam ser contados nas ações de desempilhamento). A quádrupla de redução a ser procurada tem como primeiros elementos **2**, **:=**, e **1**. Procurando na lista de quádruplas encontramos então **[2, :=, 1, 4]**, indicada pelo número **(1)** na lista abaixo. Assim, o estado correspondente ao empilhamento do não-terminal resultante da redução é o estado **4**, empilhado após o estado **1**. Naturalmente, este não-terminal não precisa ser identificado durante o processo efetivo de análise. Neste caso, entretanto, podemos verificar que se trata do não-terminal **V**, através do exame da tabela de transições entre estados vista na **seção II.2**. O valor **4** pode ser encontrado mais facilmente entrando-se na tabela de redução com os valores dados. Assim, a quarta linha representa a situação após a redução de **a** para **V**; o terminal **:=** é empilhado a seguir, obtendo-se como resultado aquele indicado na quinta linha.

A análise prossegue de forma semelhante, nos demais passos, correspondendo as ações de redução efetuadas às quádruplas constantes da lista a seguir.

```
 (1) [  2, :=,  1,  4 ]
 (2) [  2,  +,  7, 11 ]
 (3) [  2,  ;, 15, 18 ]
 (4) [ 18,  ;,  7, 11 ]
 (5) [ 11,  ;,  1,  5 ]
 (6) [  2, :=,  9,  4 ]
 (7) [  2,  *, 10, 12 ]
 (8) [  2,  ), 16, 19 ]
 (9) [ 19,  ), 10, 14 ]
(10) [ 17,  +,  7, 11 ]
(11) [  2,  ], 15, 18 ]
(12) [ 18,  ],  7, 11 ]
(13) [ 11,  ],  9, 13 ]
(14) [ 13,  ],  1,  5 ]
(15) [  8,  $,  0,  3 ]
```

Atingida a linha indicada por **(11)**, temos uma situação semelhante à da linha indicada por **(1)**, isto é, uma redução. Neste caso, entretanto, o símbolo **]**, que serve de "lookahead" para a redução, não é imediatamente empilhado, e seu empilhamento só acontece após três reduções adicionais (linhas **(12)**, **(13)**, e **(14)**).

As reduções simples, como previsto, são realizadas de forma

explícita, e podemos, portanto, supor que a análise da sequência acima se deu de acordo com a árvore indicada na Figura 2, e não através da árvore de derivação indicada na Figura 1. Mesmo nesta gramática exemplo relativamente pequena, pode-se notar uma significativa diferença no número de nós das duas árvores.

A tabela que associa regras aos estados, que foi vista na **seção II.5**, seria usada para determinar que ações semânticas devem ser executadas, à medida que a estrutura sintática da sequência de entrada vai sendo descrita através das ações de redução.

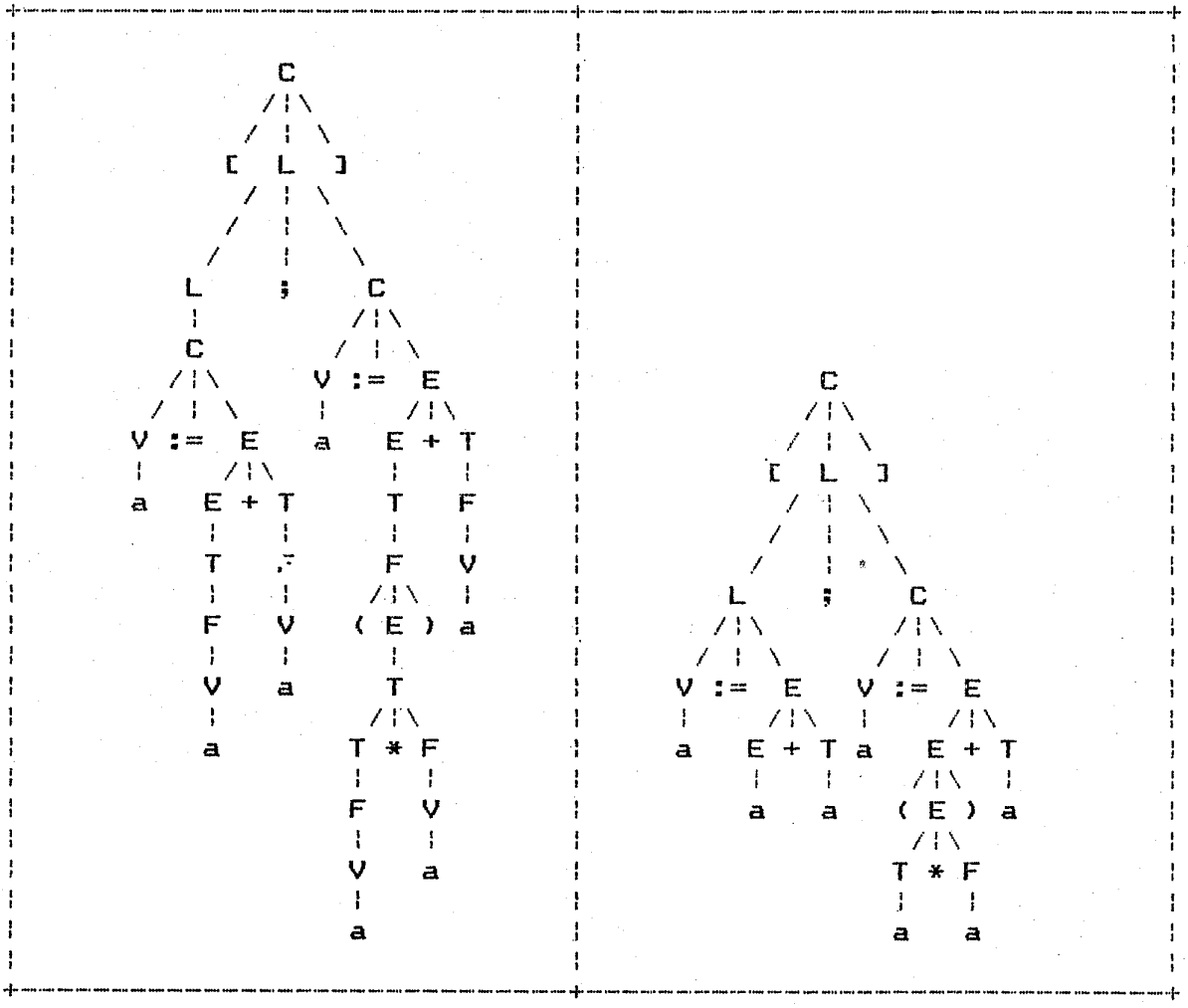


Figura 1
Arvore de derivação

Figura 2
Arvore R*S

II.7 Considerações sobre o método R*Ss

O exemplo apresentado na seção anterior mostra as duas principais diferenças entre o método R*Ss e os métodos mais conhecidos da família LR: a eliminação das regras simples durante a construção da coleção de estados, e a utilização de mais

informação para a decisão sobre uma possível redução. Temos menos estados que no caso dos analisadores que usam como base os estados LR(0), o que tende a facilitar a consulta às tabelas, e além disso, temos menos operações de redução a executar. As operações de redução tem execução mais simples, uma vez que se torna possível obter diretamente o estado que caracteriza o empilhamento do não-terminal resultante da redução, dispensando outras operações adicionais que visam obter esse estado.

O preço a pagar por essas facilidades é a complexidade da tabela de redução, que deve descrever as quádruplas de redução da gramática. Esta é uma tabela de três entradas, e portanto ocupa um espaço muito grande, se guardada em sua forma bruta. Para a gramática **Gs** do exemplo, temos 20 estados x 10 símbolos x 20 estados, ou seja, um total possível de 4000 entradas; para uma linguagem real, como Pascal, teríamos algo como dez vezes mais estados e dez vezes mais símbolos, e portanto uma tabela bruta mil vezes maior que a de **Gs**. O exame da estrutura dessa tabela, entretanto, leva à conclusão que a tabela é extremamente esparsa, e com elementos muito semelhantes, e portanto se presta bem à compactação. Métodos de compactação desenvolvidos especificamente para este tipo de tabelas se encontram implementados no programa compactador, cujo funcionamento e utilização são discutidos na **seção IV**, mas o próprio gerador já se encarrega de parte do trabalho de compactação, ainda durante o processo de geração das tabelas.

III. Utilização do programa gerador

III.1 Introdução

O programa gerador recebe como entrada uma gramática descrita no formato apropriado, mostrado abaixo, e fornece como saída dois arquivos: o arquivo de listagem e o arquivo de trabalho. O arquivo de listagem deve ser examinado para verificação da correção da gramática, e para obtenção de informações necessárias à sua alteração, caso isso se torne necessário. O arquivo de trabalho contém, em formato numérico, essencialmente as mesmas informações apresentadas no arquivo de listagem, e, em fase posterior, serve como entrada para o programa compactador.

O programa gerador solicita a indicação da gramática a ser examinada através do nome do arquivo correspondente. A desinência "default" para arquivos de gramáticas é **.grm**. O nome do arquivo da gramática é também usado para a construção dos nomes do arquivo de listagem, e do arquivo de trabalho. A desinência é substituída por **.lst** ou **.trb**, respectivamente. Assim, no caso do exemplo, os nomes dos três arquivos são **gs.grm**, **gs.lst**, e **gs.trb**.

III.2 O formato de entrada da gramática

O formato de entrada da gramática se caracteriza por:

- reunir todas as regras de um mesmo não-terminal em um único grupo, terminado por "**;**".
- o não-terminal correspondente aparece separado da primeira alternativa por "**=**".
- as alternativas (lados direitos de regras do mesmo não-terminal) aparecem separadas por "**!**".
- não-terminais são representados por identificadores formados de letras, números e do caracter "**_**" (sublinhado), devendo um identificador ser sempre iniciado por uma letra.
- terminais são representados por uma sequência qualquer entre caracteres "**'**" (aspas simples ou "plicas").
- comentários são precedidos de "**#**" e se estendem até o fim da linha.
- o não-terminal inicial da gramática é o não-terminal do lado esquerdo da primeira regra.

Abaixo temos uma listagem do arquivo **gs.grm**, que contém a gramática **Gs** usada anteriormente como exemplo.

```
C = '[' L ']'        # gramatica exemplo
  ! V ':=' E ;       # para o manual
L = L ';' C
  ! C ;
E = E '+' T
  ! T ;
T = T '*' F
  ! F ;
F = '(' E ')'
  ! V ;
V = 'a' ;
```

III.3 O arquivo de listagem

A organização do arquivo de listagem é mostrada a seguir, dividindo-se para a discussão o arquivo nas seguintes partes:

1 - arquivo fonte (gramática) e data da execução
2 - número de regras e de símbolos encontrados
3 - numeração dos terminais e não-terminais
4 - listagem das regras da gramática
5 - **geraepsilon:** conjunto dos não-terminais que geram a sequência vazia.
6 - **first:** conjunto first de cada não-terminal
7 - **follow:** conjunto follow de cada não-terminal
8 - **simples:** para cada não-terminal, o conjunto de não-terminais que geram aquele não terminal através apenas de regras simples
9 - listagem dos estados R*Ss, da função de transição, e das reduções associadas a cada estado
10 - tabela de saltos e de regras associadas a cada estado
11 - **empilha** e **acesso:** tabelas que descrevem as transições com terminais
12 - **alfa**, **beta** e **betaq:** tabelas que descrevem as quádruplas de redução

Vamos agora analisar cada uma destas partes do arquivo por sua vez, descrevendo seu conteúdo com mais detalhes. Para cada parte, apresentaremos também o trecho correspondente do arquivo **gs.lst**. Em alguns casos pequena re-arrumação do texto foi feita para facilidade de inclusão neste trabalho.

1 - Nome do arquivo fonte, para identificação, e data da execução do programa gerador, para documentação

```
fonte: gs.grm  [ 30/6/1988 - 12:26 ]
```

2 - Como indicado, o número total de regras, o número de regras não-simples, o número de símbolos não-terminais, e de símbolos terminais.

```
+-------------------------------------------------------------------------+
|    TOTAL:                                                               |
|       12 REGRAS                                                         |
|        8 REGRAS NAO SIMPLES                                             |
|        7 NAOTERMINAIS                                                   |
|       10 TERMINAIS                                                      |
+-------------------------------------------------------------------------+
```

3 - Numeração dos terminais e dos não-terminais, para permitir a interpretação da informação contida nas demais partes da listagem. A numeração dos terminais deve ser mantida para permitir a consulta às tabelas, compactadas ou não. Em particular, pode ser usada diretamente para definir os códigos numéricos dos "tokens" na construção do analisador léxico ("scanner").

```
+-------------------------------------------------------------------------+
|    TERMINAIS                                NAOTERMINAIS                |
|        0:'$'                                 1000: S'                   |
|        1:'['                                 1001: C                    |
|        2:']'                                 1002: L                    |
|        3:':='                                1003: E                    |
|        4:';'                                 1004: T                    |
|        5:'+'                                 1005: F                    |
|        6:'*'                                 1006: V                    |
|        7:'('                                                            |
|        8:')'                                                            |
|        9:'a'                                                            |
+-------------------------------------------------------------------------+
```

4 - Regras da gramática, com as duas numerações: a que inclui todas as regras, e é usada internamente no gerador, e a que inclui apenas as regras não-simples, e é utilizada na construção das tabelas compactadas. Esta última numeração é a utilizada na construção de compiladores, identificando as "ações semânticas".

```
+-----------------------------------------------------------------------+
|    REGRAS                                                             |
|                    S' =                                               |
|         0 /     0   '$' C '$'  ;                                      |
|                    C =                                                |
|         1 /     1   '[' L ']'                                         |
|         2 /     2 ! V ':=' E  ;                                       |
|                    L =                                                |
|         3 /     3   L ';' C                                           |
|                 4 ! C  ;                                              |
|                    E =                                                |
|         4 /     5   E '+' T                                           |
|                 6 ! T  ;                                              |
|                    T =                                                |
|         5 /     7   T '*' F                                           |
|                 8 ! F  ;                                              |
|                    F =                                                |
|         6 /     9   '(' E ')'                                         |
|                10 ! V  ;                                              |
|                    V =                                                |
|         7 /    11   'a'  ;                                            |
+-----------------------------------------------------------------------+
```

5 - Esta seção apresenta a lista dos não-terminais da gramática que geram a sequência vazia **epsilon**. No caso de **Gs**, essa lista é vazia.

```
+-----------------------------------------------------------------------+
|    GERAEPSILON                                                        |
+-----------------------------------------------------------------------+
```

6 - Para cada não-terminal, estão listados (nome e número) os terminais que constituem o conjunto **first** do não-terminal, isto é, os terminais que podem aparecer na primeira posição de sequências derivadas do não-terminal.

```
+-----------------------------------------------------------------------+
|    FIRST                                                              |
|    1000:S' >>   0:'$'                                                 |
|    1001:C  >>   1:'['    9:'a'                                        |
|    1002:L  >>   1:'['    9:'a'                                        |
|    1003:E  >>   7:'('    9:'a'                                        |
|    1004:T  >>   7:'('    9:'a'                                        |
|    1005:F  >>   7:'('    9:'a'                                        |
|    1006:V  >>   9:'a'                                                 |
+-----------------------------------------------------------------------+
```

7 - Para cada não-terminal, estão listados (nome e número) os terminais que constituem o conjunto **follow** do não-terminal, isto é, os terminais que podem aparecer na primeira posição após o não-terminal, em sequências derivadas do não-terminal inicial da gramática.

```
+-------------------------------------------------------------------------------+
|    FOLLOW                                                                     |
|    1001:C >>  0:'$'   2:']'   4:';'                                           |
|    1002:L >>  2:']'   4:';'                                                   |
|    1003:E >>  0:'$'   2:']'   4:';'   5:'+'   8:')'                           |
|    1004:T >>  0:'$'   2:']'   4:';'   5:'+'   6:'*'   8:')'                   |
|    1005:F >>  0:'$'   2:']'   4:';'   5:'+'   6:'*'   8:')'                   |
|    1006:V >>  0:'$'   2:']'   3:':='  4:';'   5:'+'   6:'*'                   |
|               8:')'                                                           |
+-------------------------------------------------------------------------------+
```

8 - Para cada não-terminal estão listados (nome e número) os não-terminais que derivam, através de regras **simples**, o não-terminal considerado. Estes não-terminais são os que podem substituir o não-terminal considerado, uma vez que as regras simples não são consideradas neste método.

```
+-------------------------------------------------------------------------------+
|    SIMPLES                                                                    |
|    1001 C >>                                                                  |
|    1002:L                                                                     |
|    1004 T >>                                                                  |
|    1003:E                                                                     |
|    1005 F >>                                                                  |
|    1003:E 1004:T                                                              |
|    1006 V >>                                                                  |
|    1003:E 1004:T 1005:F                                                       |
+-------------------------------------------------------------------------------+
```

9 - Aparecem aqui listados todos os estados R*Ss, com a seguinte informação associada:

- ítens principais do estado, no formato **[r / p : s]**, onde **r** é o número da regra, **p** é a posição do ponto, e **s** é o símbolo após o ponto. No caso de ítens completos, **s** não é definido, e esse fato se encontra indicado por **----**.
- transições a partir do estado, no formato **[s > p]**, onde **s** é o símbolo, e **p** o estado alvo da transição.
- reduções no estado, indicadas por **RED n**, onde **n** é o número da regra (de acordo com a numeração que inclui todas as regras, simples e não-simples). No caso de regras com lado direito vazio ("regras-epsilon"), para as quais o ítem completo não é um ítem principal, a ocorrência é marcada adicionalmente por **EPS**.

```
Estado:   0:[   0/  1:1001]
[   1>  1][   9>  2][1001>  3][1006>  4]
Estado:   1:[ .  1/  1:1002]
[   1>  1][   9>  2][1002>  5][1006>  4]
Estado:   2:[  11/  1:-----]
RED    :  11
Estado:   3:[   0/  2:   0]
[   0>  6]
Estado:   4:[   2/  1:   3]
[   3>  7]
Estado:   5:[   1/  2:   2][   3/  1:   4]
[   2>  8][   4>  9]
Estado:   6:[   0/  3:-----]
RED    :   0
Estado:   7:[   2/  2:1003]
[   7> 10][   9>  2][1003> 11][1004> 12]
Estado:   8:[   1/  3:-----]
RED    :   1
Estado:   9:[   3/  2:1001]
[   1>  1][   9>  2][1001> 13][1006>  4]
Estado:  10:[   9/  1:1003]
[   7> 10][   9>  2][1003> 14][1004> 12]
Estado:  11:[   2/  3:-----][   5/  1:   5]
RED    :   2
[   5> 15]
Estado:  12:[   7/  1:   6]
[   6> 16]
Estado:  13:[ *  3/  3:-----]
RED    :   3
Estado:  14:[   5/  1:   5][   9/  2:   8]
[   5> 15][   8> 17]
Estado:  15:[   5/  2:1004]
[   7> 10][   9>  2][1004> 18]
Estado:  16:[   7/  2:1005]
[   7> 10][   9>  2][1005> 19]
Estado:  17:[   9/  3:-----]
RED    :   9
Estado:  18:[   5/  3:-----][   7/  1:   6]
RED    :   5
[   6> 16]
Estado:  19:[   7/  3:-----]
RED    :   7

TOTAL:20 ESTADOS
```

10 - Esta tabela indica, para cada estado, os números (pelas duas numerações) das regras para as quais existem ítens completos no estado, e o comprimento dos lados direitos das regras, na coluna Salto Original. A última coluna indica o não-terminal do lado esquerdo da regra citada, e pode ser usada para identificar o follow que será usado no desempate entre as diversas regras, no caso de reduções múltiplas.

A coluna Salto Alterado se destina a uma implementação especial do analisador em que um não-terminal nunca é empilhado, se não houver certeza de que a ação seguinte é de empilhamento de terminal. O valor do Salto Alterado é o mesmo do Salto Original, exceto se a regra tem como último símbolo um não-terminal, caso em que se subtrai 1 do valor original. Sua aplicação não é muito frequente, e não será discutida neste documento.

Estado	Salto Original	Salto Alterado	Regra nao Simples	Regra Original	Nao Terminal
2	1	(1)	7	(11)	1006
6	3	(3)	0	(0)	1000
8	3	(3)	1	(1)	1001
11	3	(2)	2	(2)	1001
13	3	(2)	3	(3)	1002
17	3	(3)	6	(9)	1005
18	3	(2)	4	(5)	1003
19	3	(2)	5	(7)	1004

11 - As tabelas **empilha** e **acesso** tem a finalidade de descrever a tabela de transições com terminais, denominada de tabela de empilhamento na seção anterior. Levando em consideração que cada estado somente pode ser atingido por transições com um único símbolo, aquele que aparece depois do ponto em seus ítens principais, indicamos na tabela empilha apenas a lista de estados acessíveis através de transições com terminais a partir de cada estado considerado. Para determinar qual a transição correta com um determinado símbolo, procuramos na lista correspondente da tabela empilha o estado cujo símbolo de acesso (obtido na tabela acesso) é o desejado. Cada entrada da tabela acesso é da forma **[q, s]**, onde **s** é o símbolo de acesso do estado **q**.

As possibilidades de compactação da tabela empilha decorrem da presença de linhas iguais, ou com composição semelhante.

```
EMPILHA
    0 >   1   2
    1 >   1   2
    3 >   6
    4 >   7
    5 >   8   9
    7 >  10   2
    9 >   1   2
   10 >  10   2
   11 >  15
   12 >  16
   14 >  15  17
   15 >  10   2
   16 >  10   2
   18 >  16
```

```
ACESSO
[  0,   0][  1,   1][  2,   9][  3,1001][  4,1006][  5,1002]
[  6,   0][  7,   3][  8,   2][  9,   4][ 10,   7][ 11,1003]
[ 12,1004][ 13,1001][ 14,1003][ 15,   5][ 16,   6][ 17,   8]
[ 18,1004][ 19,1005]
```

12 - Como observado anteriormente, a tabela de redução é, nos casos de interesse prático, grande demais para que possa ser armazenada e tratada em sua forma bruta. Assim, durante a própria geração da tabela, ela é decomposta em tabelas menores, implementadas através de listas. Assim, para obter o valor de **r** em função de **q, s, p**, procedemos da forma abaixo:

- uma primeira lista (**betaq**), é consultada para se obter o valor de **beta** em função do valor de **q**; **beta** é o número da lista que contém as informações relativas a **q**. Essa lista representa a reunião de várias listas semelhantes correspondentes a vários estados.

- uma segunda lista associada a **beta** é consultada para determinar um valor **alfa** correspondente a **s**; **alfa** é o número da lista que contém as informações relativas ao par **q, s**. Como acima, essa lista representa a fusão de várias listas semelhantes correspondentes a vários pares **q, s**.

- finalmente, uma terceira lista associada a **alfa** é consultada para determinar o valor de **r** correspondente a **p**; o resultado é o valor de **r** correspondente a **q, s, p**.

Naturalmente, se **q, s, p** não representam uma situação em que uma redução é possível, caracterizando, portanto, uma situação de erro. Nenhum valor será encontrado para **r** pelo processo acima: se **q** não é um estado de redução, a busca de **beta** falha na primeira tabela; se nenhuma redução é possível no estado **q** com o símbolo **s**, a busca de **alfa** falha na segunda; e finalmente, se **p** não é um

estado que permita, em combinação com **q** e **s** uma ação de redução, a busca de **r** falhará na terceira tabela.

Durante a construção de todas as listas, a fusão de duas listas só é permitida se não impedir a identificação tanto das situações corretas, como dos erros. Para isto verificamos se há possibilidade de que uma determinada configuração seja atingida, Se a configuração é tal que nunca será atingida (porque o erro seria descoberto mais cedo durante a análise), não precisa ser tratada como uma situação de erro, e esse fato pode ser usado para facilitar a fusão de listas semelhantes. Mesmo assim, a compactação restante, a ser realizada pelo programa compactador, em fase posterior,é ainda considerável.

As listas mencionadas acima são apresentadas no arquivo de listagem. No caso do exemplo **Gs** temos as listas abaixo:

```
+-------------------------------------------------------------------------+
|     ALFA     1[   0    3]                                               |
|     ALFA     4[   0    4][  1    4][   9    4]                          |
|     ALFA     5[   7   11][ 15   18][  16   19]                          |
|     ALFA     6[   7   11][ 10   14][  15   18][  16   19]               |
|     ALFA     7[   7   12][ 10   12][  15   18][  16   19]               |
|     ALFA     9[  10   14][ 15   18][  16   19]                          |
|     ALFA    10[   1    5][  9   13]                                      |
+-------------------------------------------------------------------------+

+-------------------------------------------------------------------------+
|     BETA     0[   0    5][  2    5][  3    4][  4    5]                 |
|              [   5    6][  6    7][  8    9]                            |
|     BETA     1[   0    1][  2   10][  4   10]                           |
|     BETA     2[   2   10][  4   10]                                     |
|     BETA     3[   0    5][  2    5][  4    5][  5    6][  6    7]      |
|              [   8    9]                                                |
+-------------------------------------------------------------------------+

+-------------------------------------------------------------------------+
|     BETAQ    [ 2   0][ 8   1][11   1][13   2][17   3][18   0]           |
|              [19   0]                                                   |
+-------------------------------------------------------------------------+
```

Por exemplo, vejamos como encontrar a quádrupla assinalada por (1) no exemplo da **seção II.6**, a quádrupla **[2, :=, 1, 4]**. Note que neste caso, **s** tem o valor numérico correspondente a **:=**, ou seja, **3**. (As entradas consultadas são as assinaladas em negrito.) O valor **r = 4** pode ser determinado da forma abaixo:

- procuramos na lista indicada por **BETAQ** o valor de **beta** associado a **2**: o par **[2 0]** indica que **beta = 0.**
- procuramos na lista indicada por **BETA 0** o valor de **alfa** correspondente a **3** (código numérico do símbolo **:=**): o par **[3 4]** indica que **alfa = 4.**
- procuramos na lista indicada por **ALFA 4** o valor de **r** correspondente a **1**: o par **[1 4]** indica que **r = 4.**

13 - O estado final, ou de parada é indicado nesta seção; naturalmente, o estado inicial é, por construção, o estado **0**. Durante a fase posterior de compactação, a numeração destes (e dos demais estados) será revista, para facilitar a compactação, procurando evitar os claros nas tabelas.

```
+-------------------------------------------------------------------------------+
!    STOP   6                                                                   !
+-------------------------------------------------------------------------------+
```

III.4 Considerações finais

A maior parte da informação acima não precisa normalmente ser consultada; é fornecida para que o tratamento de conflitos, reduções múltiplas e de outras possíveis alterações possa ser feito. Se não tivermos conflitos, será suficiente utilizar o arquivo de trabalho como entrada para o compactador.

O gerador pode apresentar mensagens de erro nas diversas fases de tratamento de uma gramática; essas mensagens se referem a erros léxicos ou sintáticos na definição da gramática, a erros semânticos como um não-terminal sem produções ou uma estrutura recursiva de definição de não-terminais em função de outros que torna impossível o cálculo dos não-terminais que geram a sequência vazia, ou ainda os conflitos. Estes últimos correspondem a situações em que duas decisões podem ser tomadas, e normalmente indicam erros ou ambiguidades na definição da gramática. O tratamento dos conflitos está apresentado na seção VI.

Quando um erro é detetado numa fase do processo de geração, a execução é interrompida na passagem para a fase seguinte. A presença de conflitos é indicada na tela, durante a execução, quando é detetada, e ao final da execução, com a mensagem **"Problemas encontrados!"**. A mesma mensagem se repete ao final do arquivo de listagem, para permitir a rápida determinação da presença de erros.

Muitas vezes, o tratamento dos conflitos não será necessário. O exemplo clássico desta situação é o conflito empilha-reduz em que a preferência pelo empilhamento torna a redução não alcançável, como é o caso do conflito entre as construções **if-then** e **if-then-else** de Algol ou Pascal.

IV. Utilização do programa compactador

IV.1 Introdução

Como vimos na seção anterior, o programa gerador fornece como saída os arquivos de listagem (**.lst**) e de trabalho (**.trb**). A organização do arquivo de trabalho é semelhante à organização das partes **10** a **12** do arquivo de listagem, exceto, naturalmente, pelo formato numérico do arquivo de trabalho. A partir destes valores, as tabelas que descrevem as ações de empilhamento e de redução passam a ser construídas. Essas tabelas são organizadas na forma de constantes estruturadas da linguagem Turbo Pascal **[Bor187].**

IV.2 Compactação da tabela empilha

A representação da tabela **empilha** é feita através das listas de estados que podem ser atingidos, via transições com terminais, a partir de cada estado. Repetimos aqui a tabela, como vista no arquivo **gs.lst** para a gramática exemplo **Gs.**

```
+-------------------------------------------------------------------------+
|    EMPILHA                                                              |
|       0 >    1    2                                                     |
|       1 >    1    2                                                     |
|       3 >    6                                                          |
|       4 >    7                                                          |
|       5 >    8    9                                                     |
|       7 >   10    2                                                     |
|       9 >    1    2                                                     |
|      10 >   10    2                                                     |
|      11 >   15                                                          |
|      12 >   16                                                          |
|      14 >   15   17                                                     |
|      15 >   10    2                                                     |
|      16 >   10    2                                                     |
|      18 >   16                                                          |
+-------------------------------------------------------------------------+
```

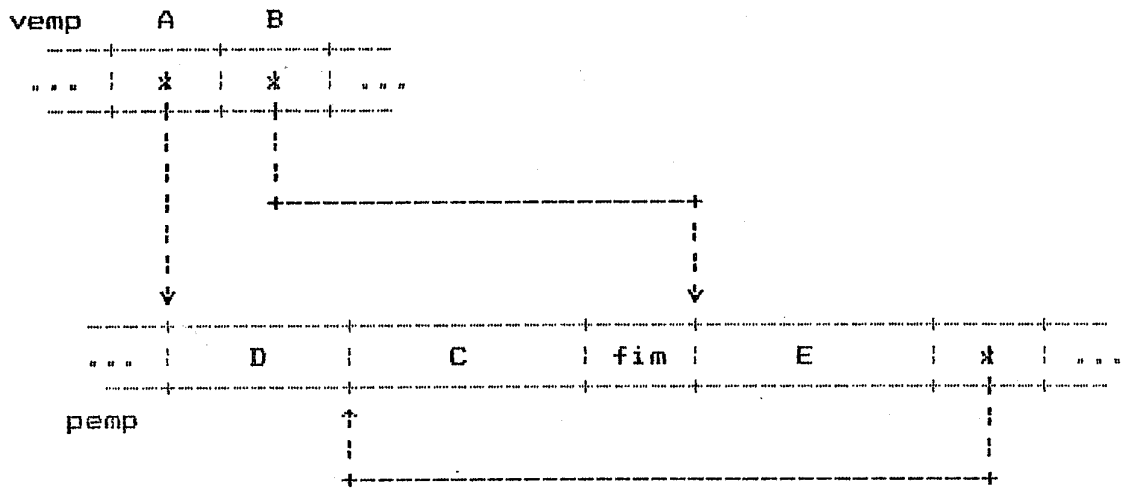
Como se pode ver, a tabela contém muitas listas iguais. Se identificarmos todas as listas distintas no caso acima, teremos apenas:

[1,2], [6], [7], [8,9], [10,2], [15], [16], [15,17]

Estas listas so precisam ser representadas uma vez. Notamos ainda que há interseções entre alguns pares de listas:

[1,2] e **[10,2]**
[15] e **[15,17]**

No caso geral, se tivermos **A** e **B** com interseção **C**, podemos sempre escrever **A = D + C** e **B = E + C**, e representar os elementos de **C** apenas uma vez:



O ponteiro ao final de **E** se distingue dos outros elementos por ser um número negativo, ou através de alguma outra convenção apropriada. Este tipo de compactação não traz vantagens se a lista **C** tem apenas um elemento, como é o caso do exemplo aqui considerado, mas traz considerável vantagem em situações reais. A representação das listas é feita através de dois vetores, denominados **pemp** e **vemp**. No caso de **Gs**, o resultado desta etapa é:

	pemp
0	1
1	1
2	fim
3	6
4	8
5	10
6	fim
7	4
8	fim
9	1
10	4
11	14
12	16
13	fim
14	13
15	4
16	4
17	fim
18	16
19	fim

	vemp
1	1
2	2
3	fim
4	10
5	-2
6	6
7	fim
8	7
9	fim
10	8
11	9
12	fim
13	17
14	15
15	fim
16	16
17	fim

Em **vemp**, um valor negativo indica um ponto de desvio; assim, um valor **p** = **-q** indica que a busca deve ser continuada a partir da posição **q** de **vemp**. Este é, então, o caso de **vemp[5]** = **-2**, que indica o desvio para a componente **vemp[2]**, cuja aplicação será vista em um exemplo apresentado mais abaixo.

Em ambas as tabelas, **fim** indica um valor inválido. Em **pemp**, esse valor inválido indica que a lista de estados alcançáveis é vazia, e portanto nenhum símbolo terminal pode ser empilhado; em **vemp**, indica que o fim da lista foi alcançado, e portanto, que aquele símbolo específico não pode ser empilhado. Naturalmente, a representação numérica desse valor é arbitrária, da mesma forma que é arbitrária a escolha dos números negativos para indicar os desvios.

Os símbolos que permitem acesso aos diversos estados são indicados em outro vetor. Assim, para o caso de **Gs**, teremos um vetor com os valores:

	acesso
0	0
1	1
2	9
3	fim
4	fim
5	fim
6	0
7	3
8	2
9	4
10	7
11	fim
12	fim
13	fim
14	fim
15	5
16	6
17	8
18	fim
19	fim

como se pode ver no arquivo de listagem. As entradas indicadas por **fim** são as correspondentes a não-terminais. Como nunca serão consultadas, seus valores são irrelevantes e podem ser escolhidos arbitrariamente.

Para se obter o estado alcançado a partir do estado **15** com o símbolo **9**, procedemos da seguinte forma:

- obtemos o valor **pemp[15] = 4**, que indica o início em **vemp** da lista de estados acessíveis a partir de **15**.

- como **vemp[4] = 10**, e **acesso[10] = 7**, **10** não é ainda o estado procurado.

- tentamos então a continuação da lista, em **vemp[5]**, que tem o valor **-2**, e encontramos um número negativo, indicando que a

busca continua da posição 2 de **vemp**.

- como **vemp[2]** = 2, e como **acesso[2]** = 9, 9 é o estado procurado.

Se, durante a busca, encontrássemos a marca **fim**, sem encontrar um estado com o valor correto de **acesso**, concluíríamos a impossibilidade do empilhamento.

IV.3 Compactação da tabela de redução

A forma de compactação da tabela de redução é semelhante à da tabela de empilhamento, exceto pelo fato de que, pela estrutura da tabela, é agora necessário tratar com listas de pares de valores numéricos.

Durante a geração da tabela, uma estrutura mais geral é necessária do que a estrutura que utilizaremos para a tabela compactada. Por essa razão, apenas três vetores se tornam necessários, e as referências aos números das listas **alfa** e **beta** são retiradas, sendo substituídas por ponteiros para os seus primeiros elementos.

A forma compactada da tabela é mostrada na página seguinte. Por exemplo, para obter **r** quando temos **q** = 2, **s** = 3, (ou seja, **:=**), e **p** = 1, seguimos o seguinte caminho:

- obtemos inicialmente **betaq[q]** = 1, de maneira que passamos para a posição 1 de **vbeta**;
- encontramos a partir de **vbeta[1]**, os pares [3 4], [0 13], etc. O par que tem o valor desejado de **s** é o primeiro, de maneira que passamos para a posição 4 de **valfa**;
- encontramos a partir de **valfa[4]**, sucessivamente, os pares [0 4], [1 4], [9 4]. O par que tem o valor desejado de **p** é o segundo, de forma que concluímos que **r** = 4.

Se tivéssemos **q** = 2, **s** = 6, e **p** = 16, o caminho a ser seguido seria mais longo:

- com **betaq[2]** = 1, começamos em **valfa[1]**;
- a partir dessa posição, encontramos diversos pares [3 4], [0 13], [? 13], [4 13], [5 11] e [6 20], sendo o último o que nos interessa; continuamos então a partir de **vbeta[20]**;
- a partir daí, encontramos os pares [7 12], [10 12], e a indicação -15; em vbeta[15] encontramos [15 18] e [16 19]. O par que nos interessa é o último, e **r** = 19.

Se a qualquer momento da busca encontrarmos alguma marca **fim**, a busca pode ser encerrada sem sucesso: a redução não é possível.

Tabelas correspondentes à tabela de redução

	betaq
0	fim
1	fim
2	1
3	fim
4	fim
5	fim
6	fim
7	fim
8	16
9	fim
10	fim
11	16
12	fim
13	18
14	fim
15	fim
16	fim
17	3
18	1
19	1

	vbeta
1	3
2	4
3	0
4	13
5	2
6	13
7	4
8	13
9	5
10	11
11	6
12	20
13	8
14	25
15	fim
16	0
17	1
18	2
19	28
20	4
21	28
22	fim

	valfa
1	0
2	3
3	fim
4	0
5	4
6	1
7	4
8	9
9	4
10	fim
11	10
12	14
13	7
14	11
15	15
16	18
17	16
18	19
19	fim
20	7
21	12
22	10
23	12
24	-15
25	10
26	14
27	-15
28	1
29	5
30	9
31	13
32	fim

IV.4 A renumeração dos estados

Para limitar ao mínimo necessário a ocorrência de entradas com valor **fim** nas tabelas, e evitar os intervalos vazios nas tabelas **pemp**, **reduz**, **salto** e **betaq**, os estados são renumerados, de forma que estados de redução, e estados de empilhamento tenham, tanto quanto possível numerações contíguas. Essa renumeração fica documentada num arquivo com desinência **.ren**, para alterações que se deseje fazer nas tabelas compactadas. O conteúdo do arquivo **gs.ren**, correspondente à gramática exemplo **Gs**, está listado abaixo:

gramatica:gs.grm [30/6/1988 - 12:26]

estado	novo	estado	velho
0	8	0	13
1	9	1	19
2	2	2	2
3	15	3	6
4	16	4	8
5	17	5	17
6	3	6	11
7	10	7	18
8	4	8	0
9	11	9	1
10	12	10	7
11	6	11	9
12	18	12	10
13	0	13	15
14	19	14	16
15	13	15	3
16	14	16	4
17	5	17	5
18	7	18	12
19	1	19	14

Pode-se ver que, nesse arquivo, a renumeração fica indicada nos dois sentidos, para maior facilidade de consulta.

IV.5 As tabelas compactadas

O resultado principal da compactação é o arquivo de tabelas, indicado pela desinencia **.tab**; as tabelas incluídas nesse arquivo são:

- **pemp**, **vemp** e **acesso**, que representam as ações de empilhamento de terminais;

- **reduz**, que indica para cada estado de redução o número da regra correspondente; naturalmente, usamos aqui a numeração das regras que exclui as regras simples.

- **salto**, que indica quantos estados devem ser retirados da pilha em uma ação de redução;

- **betaq**, **vbeta**, e **valfa** que representam as ações de redução;

Em todas as tabelas, a numeração empregada pelos estados é a descrita na seção anterior; temos os limites para cada faixa definidos pelas constantes **0**, **inipemp**, **fimreduz** e **fimpemp**: os estados de redução são numerados de 0 a **fimreduz**; os que tem ações de empilhamento de **inipemp** a **fimpemp**; os estados que pertençam simultaneamente às duas categorias ficam na interseção dos dois intervalos, isto é, entre **inipemp** e **fimreduz**.

As constantes **estinicial**, e **estfinal** correspondem aos novos valores dos estados inicial e final, após a renumeração. A constante **fim**, como anteriormente mencionado tem seu valor arbitrariamente escolhido, desde que não haja perigo de confusão com outros valores constantes das mesmas tabelas.

Para a gramática exemplo **Gs**, temos o arquivo **gs.tab** abaixo:

```
{gramatica: gs.grm [30/6/1988 - 12:26 ]
}
 const
   estinicial=8;
   estfinal=3;
   fim=-8888;
   inipemp=6;
   fimpemp=19;
   pemp:array[6..19] of integer=(
     16,   13,    1,    1,   11,    1,   11,   11,   11,    4,
      6,    8,   13,   15);
   vemp:array[1..17] of integer=(
      9,    2,  fim,    3,  fim,   10,  fim,    4,   11,  fim,
     12,   -2,   14,  fim,    5,   13,  fim);
   acesso:array[2..14] of integer=(
      9,    0,    2,    8, 1003, 1004,    0,    1,    3,    4,
      7,    5,    6);
   fimreduz=7;
   reduz:array[0..7] of integer=(
      3,    5,    7,    0,    1,    6,    2,    4);
   salto:array[0..7] of integer=(
      3,    3,    1,    3,    3,    3,    3,    3);
   betaq:array[0..7] of integer=(
     18,    1,    1,   -1,   16,    3,   16,    1);
   vbeta:array[1..22] of integer=(
      3,    4,    0,   13,    2,   13,    4,   13,    5,   11,
      6,   20,    8,   25,  fim,    0,    1,    2,   28,    4,
     28,  fim);
   valfa:array[1..32] of integer=(
      8,   15,  fim,    8,   16,    9,   16,   11,   16,  fim,
     12,   19,   10,    6,   13,    7,   14,    1,  fim,   10,
     18,   12,   18,  -15,   12,   19,  -15,    9,   17,   11,
      0,  fim);
```

O arquivo **.tab** contém um trecho sintática e semanticamente correto de Turbo Pascal. Para formar uma unidade de compilação ("unit") de Turbo Pascal 4.0, é necessário acrescentar duas linhas no início e duas linhas no fim do arquivo:

```
unit tabelas; { ou outro nome qualquer }
interface
     { incluir aqui o arquivo .tab }
implementation
end.
```

Esta forma de implementação dispensa a necessidade de recompilação das tabelas a cada vez que se queira compilar o programa que delas faz uso. Se não se deseja fazer uso da facilidade de compilação em separado do Turbo Pascal 4.0, a diretiva **$I**, de inclusão de arquivos, pode ser usada diretamente.

Podemos reduzir adicionalmente o tamanho efetivo das tabelas pelo uso do tipo **byte** em vez do tipo **integer**, onde possível.

Para utilizar as tabelas compactadas em outras linguagens, a forma mais simples de trabalhar é através de uma ferramenta q.e, a partir do arquivo de tabelas, gere outro arquivo no formato apropriado. As referências na outra linguagem serão então feitas a esse arquivo.

Assim, se a linguagem não permite a definição de constantes de tipo array, a melhor solução pode ser a utilização de um arquivo com os valores numéricos das constantes, e de variáveis tipo array, que serão tratadas como constantes. Durante a fase inicial do programa, o arquivo numérico será lido, e as variáveis que fazem o papel de constantes receberão seus valores corretos.

V. Considerações finais

V.1 Exemplos de gramáticas com conflitos

Apresentamos aqui exemplos de gramáticas com conflitos, através dos quais discutimos as formas de alteração de gramáticas que permitam a geração de um analisador sintático adequado. Quando a gramática apresenta conflitos, restam ao construtor do analisador sintático diversas possibilidades:

- re-escrever a gramática, obtendo outra **equivalente** que não tenha os conflitos apresentados pela primeira, e que permita a chamada de ações semânticas em pontos adequados;

- re-escrever a gramática, obtendo outra **não-equivalente**, que não tenha os conflitos apresentados pela primeira, e permita a chamada de ações semânticas em pontos adequados; neste caso, a linguagem da segunda gramática deve conter a linguagem da primeira, e algumas ações semânticas especiais serão usadas para verificar que a sintaxe definida pela primeira linguagem é respeitada.

- utilizar a gramática com os conflitos, verificando em cada ponto qual a escolha feita pela implementação, que é alterada diretamente quando a escolha feita não coincide com a desejada.

A determinação da melhor forma de atuação depende do caso a ser considerado, e da aplicação que se tem em vista.

V.1.1 Primeiro Exemplo

O primeiro exemplo de gramática com conflitos sinalizados pelo gerador é baseado na ambiguidade já mencionada da construção **if-then/if-then-else** de linguagens como Algol60 ou Pascal. A gramática **G1** abaixo apresenta o conflito mencionado.

```
G1: 0. S' = $ C $
    1. C = if E then C
         ! if E then C else C
         ! c
    2. E = a
```

Antes de examinar as mensagens de erro produzidas pelo gerador, vejamos a composição dos estados. Apenas os itens principais de cada estado estão listados.

```
0. S' = $.C $

1. C = if.E th C
     ! if.E th C el C

2. C = c.

3. S' = $ C.$

4. E = a.

5. C = if E.th C
     ! if E.th C el C

6. S' = $ C $.

7. C = if E th.C
     ! if E th.C el C

8. C = if E th C.
     ! if E th C.el C

9. C = if E th C el.C

10. C = if E th C el C.
```

Neste caso, a tabela de transicões é a seguinte.

Est.	Transições $	if	then	else	c	a	C	E	Reduções
0	–	1	–	–	2	–	3	–	–
1	–	–	–	–	–	4	–	5	–
2	–	–	–	–	–	–	–	–	red 3
3	6	–	–	–	–	–	–	–	–
4	–	–	–	–	–	–	–	–	red 4
5	–	–	7	–	–	–	–	–	–
6	–	–	–	–	–	–	–	–	red 0
7	–	1	–	–	2	–	8	–	–
8	–	–	–	9	–	–	–	–	red 1
9	–	1	–	–	2	–	10	–	–
10	–	–	–	–	–	–	–	–	red 2

O arquivo de listagem correspondente contem as mensagens abaixo:

```
|    Conflito  Reduz / Empilha ! [   8    3 ][  8,  3,  7,  8]   |
|    Conflito  Reduz / Empilha ! [   8    3 ][  8,  3,  7,  8]   |
|    Conflito  Reduz / Empilha ! [   8    3 ][  8,  3,  9, 10]   |
```

Como era de se esperar, os conflitos anunciados são todos do tipo empilha/reduz, e em todos eles o símbolo 3 (**else**) é o símbolo s envolvido. Cumpre observar que, por vezes, a mensagem correspondente a um conflito pode aparecer mais de uma vez, devido ao fato de que a mesma situação é encontrada mais de uma vez durante o processo de construção das tabelas e verificação dos estados. No caso acima, a quádrupla [8, 3, 7, 8] é gerada duas vezes, devido ao próprio conflito (o estado **r = 8** aceita as duas ações com **else**: empilhamento e redução). A cada vez, o conflito com o empilhamento de **else** é reconhecido e assinalado.

Estes conflitos se devem à ambiguidade existente na gramática, e não podem ser removidos sem sua alteração. Esta ambigui-

dade é comumente resolvida através de uma regra que manda "anexar o **else** ao último **if** aberto". Naturalmente, isto significa preferir o empilhamento à redução, e como presumimos que sempre o algoritmo tentará empilhar antes de reduzir, a redução constante da tabela de redução é inócua, uma vez que nunca será alcançada.

Poderíamos, é claro, atingir o mesmo resultado pela alteração da gramática. A gramática **Gx1** abaixo é equivalente a **G1** (gera a mesma linguagem) mas não é ambígua (tem apenas uma árvore de derivação para cada sequência da linguagem).

```
Gx1:  0. S' = $ C $
      1. C  = if E then C
      2.    ! if E then Ce else C
      3.    ! c
      4. Ce = if E then Ce else Ce
      5.    ! c
      6. E  = a
```

Os estados e a tabela de transição de **Gx1** são como se segue:

```
0. S'= $.C $

1. C = if.E then C
     ! if.E then Ce else C

2. C = c.

3. S'= $ C.$

4. E = a.

5. C = if E.then C
     ! if E.then Ce else C

6. S'= $ C $.

7. C = if E then.C
     ! if E then.Ce else C

8. C = if.E then C
     ! if.E then Ce else C
   Ce= if.E then Ce else Ce

9. C = c.
   Ce= c.

10. C = if E then C.

11. C = if E then Ce. else C

12. C = if E.then C
      ! if E.then Ce else C
    Ce= if E.then Ce else Ce

13. C = if E then Ce else.C

14. C = if E then.C
      ! if E then.Ce else C
    Ce= if E then.Ce else Ce

15. C = if E then Ce else C.

16. C = if E then Ce.else C
    Ce= if E then Ce.else Ce

17. C = if E then Ce else.C
    Ce= if E then Ce else.Ce

18. Ce= if E then Ce else Ce.
```

O não-terminal **Ce** acrescentado à gramática se caracteriza por ser uma forma de comando restrito: ou é um comando simples ou um comando if-then-else, estando excluída a possibilidade de um **if-then**, sem o **else**. Esta forma de comando restrito ocorre exclusivamente antes de um terminal **else**, de forma que nenhum novo conflito é introduzido, e a ambiguidade original é retirada, uma vez que ela se deve à possibilidade, aqui excluída, de um comando **if-then** precedendo um **else**.

Est.	Transições $	if	then	else	c	a	C	Ce	E	Reduções
0	-	1	-	-	2	-	3	-	-	-
1	-	-	-	-	-	4	-	-	5	-
2	-	-	-	-	-	-	-	-	-	red 3
3	6	-	-	-	-	-	-	-	-	-
4	-	-	-	-	-	-	-	-	-	red 6
5	-	-	7	-	-	-	-	-	-	-
6	-	-	-	-	-	-	-	-	-	red 0
7	-	8	-	-	9	-	10	11	-	-
8	-	-	-	-	-	4	-	-	12	red 1
9	-	-	-	-	-	-	-	-	-	red 3,
										red 5
10	-	-	-	-	-	-	-	-	-	red 1
11	-	-	-	13	-	-	-	-	-	-
12	-	-	14	-	-	-	-	-	-	-
13	-	1	-	-	2	-	15	-	-	-
14	-	8	-	-	9	-	-	16	-	-
15	-	-	-	-	-	-	-	-	-	red 2
16	-	-	-	17	-	-	-	-	-	-
17	-	8	-	-	9	-	15	18	-	-
18	-	-	-	-	-	-	-	-	-	red 4

Como se pode notar, há mais regras e mais estados nesta gramática do que na gramática anterior. Há ainda a duplicação adicional de duas reduções no mesmo estado, que não é tratada pelo compactador. Apesar do consenso quase unânime dos programadores de compiladores pela não alteração da gramática em casos como este, é plenamente justificada, e é independente do método de análise sintática utilizado.

A presença de reduções múltiplas é assinalada na descrição do estado correspondente, na tabela que indica as reduções em cada estado, e em uma mensagem específica destinada a chamar a atenção para o fato. No caso de **Gx1**, temos a seguinte mensagem:

```
[ reducoes multiplas no estado     9 ! ]
```

chamando a atenção para o estado 9, cuja composição é

```
9. C = c.
   Ce= c.
```

Como os lados direitos das duas regras tem comprimento 1, não há necessidade de alterar o salto; apenas as ações semânticas a serem tomadas devem ser as associadas à regra apropriada. O teste pode ser feito em função do símbolo de "lookahead", por exemplo, executando-se a ação semântica associada à regra 5 apenas no caso em que este símbolo é **else**. A decisão se faz

levando em consideração os conjuntos **follow[C]** = { **$** } e **follow[Ce]** = { **else** }.

Em casos semelhantes a este, pode até acontecer, na prática, que as ações semânticas associadas às duas regras sejam exatamente as mesmas, e neste caso, o teste será dispensável.

O gerador, nos casos de reduções múltiplas, inclui apenas a primeira redução no arquivo de trabalho, de forma que as demais reduções não são consideradas pelo compactador, cabendo ao usuário o seu tratamento. A frequência relativamente pequena de ocorrência de estados com reduções múltiplas faz com que seu tratamento não se justifique no caso geral. Note-se que a nível de arquivo de listagem, e a nível de teste de conflitos, todas as reduções são consideradas.

V.1.2 Segundo exemplo

O segundo exemplo leva em consideracão problemas relacionados com a precedência de operadores em expressões:

```
G2: 0. S'= $ E $
    1. E = E + E
    2.   ! E * E
    3.   ! ( E )
    4.   ! a
```

Para essa gramática temos os estados e a tabela de transição abaixo:

```
0. S'= $.E $
1. E = (.E )
2. E = a.
3. S'= $ E.$
   E = E.+ E
   E = E.* E
4. E = ( E.)
   E = E.+ E
   E = E.* E
5. S'= $ E $.
6. E = E +.E
7. E = E *.E
8. E = ( E ).
9. E = E + E.
   E = E.+ E
   E = E.* E
10. E = E * E.
    E = E.+ E
    E = E.* E
```

Est.	Transições $	+	*	(	)	a	E	Reduções
0	-	-	-	1	-	2	3	-
1	-	-	-	1	-	2	4	-
2	-	-	-	-	-	-	-	red 4
3	5	6	7	-	-	-	-	-
4	-	6	7	-	8	-	-	-
5	-	-	-	-	-	-	-	red 0
6	-	-	-	1	-	2	9	-
7	-	-	-	1	-	2	10	-
8	-	-	-	-	-	-	-	red 3
9	-	6	7	-	-	-	-	red 1
10	-	6	7	-	-	-	-	red 2

Neste caso, a ambiguidade da gramática leva a uma longa lista de conflitos:

```
!    Conflito  Reduz / Empilha ! [   9   1 ][  9,  1,  0,  3] !
!    Conflito  Reduz / Empilha ! [   9   2 ][  9,  2,  0,  3] !
!    Conflito  Reduz / Empilha ! [   9   1 ][  9,  1,  1,  4] !
!    Conflito  Reduz / Empilha ! [   9   2 ][  9,  2,  1,  4] !
!    Conflito  Reduz / Empilha ! [   9   1 ][  9,  1,  6,  9] !
!    Conflito  Reduz / Empilha ! [   9   1 ][  9,  1,  6,  9] !
!    Conflito  Reduz / Empilha ! [   9   2 ][  9,  2,  6,  9] !
!    Conflito  Reduz / Empilha ! [   9   2 ][  9,  2,  6,  9] !
!    Conflito  Reduz / Empilha ! [   9   1 ][  9,  1,  7, 10] !
!    Conflito  Reduz / Empilha ! [   9   1 ][  9,  1,  7, 10] !
!    Conflito  Reduz / Empilha ! [   9   2 ][  9,  2,  7, 10] !
!    Conflito  Reduz / Empilha ! [   9   2 ][  9,  2,  7, 10] !
!    Conflito  Reduz / Empilha ! [  10   1 ][ 10,  1,  0,  3] !
!    Conflito  Reduz / Empilha ! [  10   2 ][ 10,  2,  0,  3] !
!    Conflito  Reduz / Empilha ! [  10   1 ][ 10,  1,  1,  4] !
!    Conflito  Reduz / Empilha ! [  10   2 ][ 10,  2,  1,  4] !
!    Conflito  Reduz / Empilha ! [  10   1 ][ 10,  1,  6,  9] !
!    Conflito  Reduz / Empilha ! [  10   1 ][ 10,  1,  6,  9] !
!    Conflito  Reduz / Empilha ! [  10   2 ][ 10,  2,  6,  9] !
!    Conflito  Reduz / Empilha ! [  10   2 ][ 10,  2,  6,  9] !
!    Conflito  Reduz / Empilha ! [  10   1 ][ 10,  1,  7, 10] !
!    Conflito  Reduz / Empilha ! [  10   1 ][ 10,  1,  7, 10] !
!    Conflito  Reduz / Empilha ! [  10   2 ][ 10,  2,  7, 10] !
!    Conflito  Reduz / Empilha ! [  10   2 ][ 10,  2,  7, 10] !
```

Da mesma forma anterior, encontramos alguma repetição nas mensagens dos conflitos. Examinando os estados correspondentes, veremos que os conflitos se referem aos estados **9** e **10**, e aos símbolos **1** e **2**, ou seja, respectivamente, **+** e *****. No estado **9**, a redução é por **E = E + E**, e a preferência é pelo empilhamento, no caso de *****, e pela redução, no caso de **+**. No estado **10**, a preferência é pela redução, nos dois casos. A tabela de empilhamento deve ser alterada de forma a impedir o empilhamento

nos três casos em que preferimos a redução: estado 9, com + e estado 10, com + e *.

A tabela de empilhamento original é

	$	+	*	(	)	a
0	-	-	-	1	-	2
1	-	-	-	1	-	2
3	5	6	7	-	-	-
4	-	6	7	-	8	-
6	-	-	-	1	-	2
7	-	-	-	1	-	2
9	-	6	7	-	-	-
10	-	6	7	-	-	-

e deve ser substituída por

	$	+	*	(	)	a
0	-	-	-	1	-	2
1	-	-	-	1	-	2
3	5	6	7	-	-	-
4	-	6	7	-	8	-
6	-	-	-	1	-	2
7	-	-	-	1	-	2
9	-	-	7	-	-	-

A alteração não pode ser feita simplesmente pela retirada dos valores indesejáveis da tabela compactada, porque as listas de estados alcançáveis compartilham esses valores pela compactação.

Um procedimento que pode ser aplicado em geral é preceder a consulta à tabela por um teste, e não consultar a tabela nos casos em que houve alteração. O código adicional incluído, entretanto, encarece o processamento para todos os casos, e não apenas aqueles em que houve alteração, e, se possível, este procedimento deve ser evitado.

A outra forma de trabalho é a alteração direta das tabelas **vemp** e **pemp**, o que deve ser feito caso a caso. Para nosso exemplo, consultamos o arquivo de renumeração **g2.ren**, e observamos que os estados 6, 7, 9 e 10 tem agora os números 7, 8, 3 e 4, respectivamente. A numeração dos símbolos não se altera, de forma que devemos alterar a tabela de forma equivalente à retirada das entradas [3, 1], [4, 1], [4, 2]; entretanto, para a entrada [3, 2], o valor 8 deve ser mantido.

A forma original encontrada é

```
const
  pemp:array[3..10] of integer=(
    5,    5,    1,    1,    1,    1,    4,    8);
  vemp:array[1..9] of integer=(
    6,    0,  fim,    1,    7,    8,  fim,    2,   -5);
```

e as alterações podem ser feitas substituindo as duas primeiras entradas de **pemp** por **6** (para acesso apenas a 8), e **fim**. Os novos valores de **pemp** e de **vemp** ficam sendo então

```
const
  pemp:array[3..10] of integer=(
    6,  fim,    1,    1,    1,    1,    4,    8);
  vemp:array[1..9] of integer=(
    6,    0,  fim,    1,    7,    8,  fim,    2,   -5);
```

Em qualquer caso, se fatores como velocidade de acesso ou espaço ocupado forem críticos, pode ser necessário reformular mais profundamente as tabelas e/ou o algoritmo de consulta a elas. Em geral, entretanto, as alterações ficam restritas apenas às entradas correspondentes às ações de empilhamento que se deseja suprimir.

V.1.3 Terceiro exemplo

Este terceiro exemplo (G3) apresenta um conflito reduz/reduz, ao contrário dos dois exemplos anteriores, que apenas apresentaram conflitos empilha/reduz.

```
G3: D = L : T
      ! I : T := C
    L = L , I
      ! I
    I = a
    T = t
    C = c
```

Podemos interpretar as sentenças geradas pela gramática G3 como declarações de duas formas: várias variáveis, sem valor inicial, ou uma única, com o correspondente valor inicial.

Os estados correspondentes a G3 são os seguintes:

```
0. S'= $.D $                    8. D = I :.T := C

1. I = a.                       9. T = t.

2. S'= $ D.$                   10. D = L : T.

3. D = L.: T                   11. L = L , I.
   L = L., T
```

4. D = I.: T := C

5. S'= $ D $.

6. D = L :.T

7. L = L ,.I

12. D = I : T.:= C

13. D = I : T :=.C

14. C = c.

15. D = I : T := C.

A tabela de transição correspondente a G3 é a mostrada na página seguinte.

A mensagem de conflito para esta gramática é a seguinte:

```
+-------------------------------------------------------------------+
|    Conflito ! [  1,  1,  0,  3][  1,  1,  0,  4]                 |
+-------------------------------------------------------------------+
```

Trata-se de um conflito do tipo reduz/reduz. Em ambos os casos, temos **q = 1**, **s = 1 (:)**, e **p = 0**; a redução não-simples a ser efetuada é pela regra 5. O não-terminal a ser empilhado é que não está determinado: pode ser o não-terminal L, levando ao estado 3, ou o não-terminal I, levando ao estado 4. Naturalmente, a diferença entre um e outro caso é a de uma redução pela regra 4, uma regra simples.

Tabela de transição de G3

Est.	Transições $	:	:=	,	a	t	c	D	L	I	T	C	Red.
0	–	–	–	–	1	–	–	2	3	4	–	–	–
1	–	–	–	–	–	–	–	–	–	–	–	–	r5
2	5	–	–	–	–	–	–	–	–	–	–	–	–
3	–	6	–	7	–	–	–	–	–	–	–	–	–
4	–	8	–	–	–	–	–	–	–	–	–	–	–
5	–	–	–	–	–	–	–	–	–	–	–	–	r0
6	–	–	–	–	–	9	–	–	–	–	10	–	–
7	–	–	–	–	1	–	–	–	–	11	–	–	–
8	–	–	–	–	–	9	–	–	–	–	12	–	–
9	–	–	–	–	–	–	–	–	–	–	–	–	r6
10	–	–	–	–	–	–	–	–	–	–	–	–	r1
11	–	–	–	–	–	–	–	–	–	–	–	–	r3
12	–	–	13	–	–	–	–	–	–	–	–	–	–
13	–	–	–	–	–	–	14	–	–	–	–	15	–
14	–	–	–	–	–	–	–	–	–	–	–	–	r7
15	–	–	–	–	–	–	–	–	–	–	–	–	r2

A solução mais indicada neste caso é claramente a de transferir para uma ação semântica o ônus de verificar que apenas uma variável está sendo declarada no caso da inicialização. Assim, alteramos a regra 2 para **D = L : T := C**, e a gramática assim obtida não terá o conflito.

Outra possibilidade reside na construção de uma gramática equivalente que não oferece o conflito. Uma vez que o conflito está entre **I** e a lista **L** com exatamente uma ocorrência de **I**, excluímos de **L** esta possibilidade, passando **L** a denotar as listas de duas ou mais ocorrências de **I**:

```
D = L : T
  ! I : T
  ! I : T := C
L = L , I
  ! I , I
```

A nova gramática assim obtida tem mais regras e mais estados que a original, mas gera exatamente a mesma linguagem. Normalmente, transformações como estas acabam criando muitos casos particulares e sobrecarregando o código encarregado do tratamento semântico, além de tornar o programa menos legível.

V.2 Semântica em regras simples

Como já observado neste trabalho, os analisadores sintáticos R*S não sinalizam a presença de reduções por regras simples. Entretanto, pode ser conveniente, ou até mesmo necessário, associar ações semânticas a essas regras. Neste caso, devemos alterar a gramática, obtendo uma gramática equivalente, em que a regra correspondente é uma regra não-simples. Isso pode ser feito pela adição de um não-terminal especial, que deriva exclusivamente a sequência vazia, e que pode ser acrescencado ao lado direito da regra simples. Assim, substituímos a regra simples **A = B** por uma regra **A = B X**, não-simples, e acrescentamos uma regra que leva o não-terminal novo **X** apenas à sequência vazia epsilon. Esta regra é indicada no arquivo fonte por

```
X = ;
```

A transformação acima pode ser usada, no caso geral, uma vez que não introduz novos conflitos, mas não é, naturalmente, a única transformação possível que soluciona nosso problema. Por exemplo, se tivermos uma regra simples **A = B** e uma regra (usualmente não-simples) **B = beta**, podemos substituir o efeito combinado das duas por uma regra **A = beta**. A essa nova regra, devemos associar uma combinação das ações semânticas das duas regras anteriores. Usando judiciosamente esta transformação, podemos efetuar a remoção de uma regra simples não desejada.

V.3 Reduções múltiplas

Durante a discussão sobre os conflitos encontrados em gramáticas e a forma de solucionar os problemas criados por eles, observamos que reduções múltiplas não são levadas em consideração pelo programa gerador, na construção do arquivo de listagem. Portanto, esas reduções não podem ser levadas em consideração pelo programa compactador, na construção das tabelas compactadas. A decisão de projeto que levou a essa situação decorreu do fato

de que a ocorrência das reduções múltiplas é pouco frequente. Com efeito, raras vezes as gramáticas de linguagens de programação contêm regras que levem a esta situação, e quando a situaçao ocorre, apenas em um ou dois (entre as duas ou três centenas de estados, que são encontrados em casos reais) a situação pode ser identificada.

Por essas razões, o programa gerador constrói o arquivo de listagem incluindo nele apenas a primeira das diversas regras com ítens completos no estado, e o comprimento do lado direito dessa regra. Havendo mais de uma regra, a determinação da regra correta deve ser feita verificando a qual dos conjuntos **follow** pertence o símbolo de lookahead **s**. Note que a análise feita pelo gerador leva em conta todos os ítens completos do estado, e todas as possibilidades de conflito.

Para tornar a discussão mais simples, suporemos que há apenas duas reduções no estado **q** considerado, digamos **A = alfa.** e **B = beta.**. A extensão para o caso mais geral é imediata. Observemos que, pela forma de construção dos estados, vale uma das três igualdades: ou **alfa = beta**, ou **alfa = gama beta**, ou ainda **beta = gama alfa**.

Para decidir qual a regra correta, consultamos os conjuntos **follow[A]** e **follow[B]**, e verificamos a qual dos dois pertence o símbolo de "lookahead" **s**. (Se **s** não pertence a nenhum dos dois conjuntos, temos um erro.) Como a primeira regra já é considerada, a alteração deve ser feita apenas no caso em que **s** pertence a **follow[B]**. Neste caso, o comprimento de **beta** deve ser usado para a obtenção de **p**, em vez do comprimento de **alfa**, que é o constante da tabela **salto**, e a ação semântica a ser executada deve ser a associada à regra **B = beta**, e não a associada à regra **A = alfa**, cujo número consta da tabela **reduz**. Naturalmente, se tivermos o caso **alfa = beta**, teremos a igualdade dos dois comprimentos, e não haverá necessidade de tratamento especial do salto.

Todas as informações necessárias para as alterações mencionadas acima estão disponíveis no arquivo de listagem e no arquivo de renúmeração.

A situação mencionada é encontrada em gramáticas como a do exemplo seguinte:

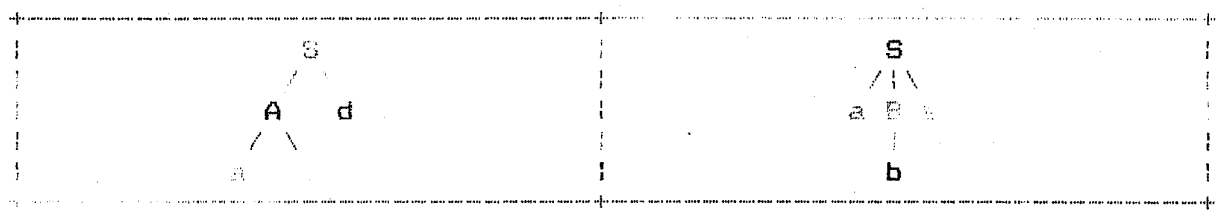
```
G4: S = A d
      ! a B e
    A = a b
    B = b
```

A gramática **G4** gera apena duas sequências, as sequências **a b d** e **a b e**, de acordo com as derivações

S => A d => a b d

S => a B e => a b e.

As correspondentes árvores de derivação estão representadas a seguir:



O analisador construído para esta gramática contem um estado com os ítens

A = a b.
B = b.

sem que entretanto, o conflito se caracterize, uma vez que os conjuntos **follow[A]** = { **d** }, e **follow[B]** = { **e** } são disjuntos. O único problema, como mencionado acima, é que apenas uma das duas reduções estará incluída na tabela correspondente e será necessário verificar se é a correta para cada situação.

A solução mais provável de ser utilizada, entretanto, não é esta, uma vez que raramente temos construções semelhantes, como as encontradas acima, sem que a sintaxe seja a mesma nos dois casos. Assim, uma uniformização da sintaxe seria provavelmente a solução preferida.

Caso a solução preferida seja a manutenção da gramática original, todas as informações necessárias para as alterações mencionadas acima estão disponíveis no arquivo de listagem e no arquivo de renumeração.

V.4 Um exemplo anotado

Para mostrar a forma de utilização do gerador em uma situação real, apresentamos aqui um exemplo, um subconjunto de Pascal, definido apenas com essa finalidade. Para esse subconjunto, construímos um analisador sintático, ao qual se acrescentaram as rotinas necessárias para a construção de árvores sintáticas, da forma comumente usada com o gerador R*S. Vamos relatar aqui o processo de construção e e suas características principais. Os arquivos a que fazemos referência estão incluídos no disco de distribuição do gerador.

Inicialmente, a gramática que define o subconjunto foi escrita, em um arquivo **pasx.grm**. Esse arquivo foi então usado como fonte para a gerador, que forneceu como resposta os arquivos **pasx.lst** e **pasx.trb**. No arquivo de listagem, encontramos informações sobre os conflitos da gramática. Com efeito, a gramática não poderia ser R*S simples, uma vez que apresenta a ambiguidade no comando **if**, que já foi discutida anteriormente neste trabalho.

As mensagens de conflitos encontradas são:

```
+-------------------------------------------------------------------------+
|    Conflito  Reduz / Empilha ! [ 144   28 ][144, 28, 79,129]            |
|    Conflito  Reduz / Empilha ! [ 144   28 ][144, 28,101,144]            |
|    Conflito  Reduz / Empilha ! [ 144   28 ][144, 28,101,144]            |
|    Conflito  Reduz / Empilha ! [ 144   28 ][144, 28,121,151]            |
|    Conflito  Reduz / Empilha ! [ 144   28 ][144, 28,165,179]            |
|    Conflito  Reduz / Empilha ! [ 144   28 ][144, 28,186,188]            |
+-------------------------------------------------------------------------+
```

Para analisar estas mensagens, é necessário examinar o conteúdo dos estados mencionados, que pode ser encontrado no arquivo de listagem. Em particular, o estado **144** é descrito por seus ítens principais

```
+-------------------------------------------------------------------------+
|    Estado: 144:[  74/  4:-----][  75/  4:  28]                          |
|    RED    :  74                                                         |
|    [  28>165]                                                           |
+-------------------------------------------------------------------------+
```

ou equivalentemente

```
if_stat = 'If' expr 'Then' stat .
        ! 'If' expr 'Then' stat . 'Else' stat
```

verificando-se quais as regras relevantes, pela numeração original. Quanto aos símbolos, apenas precisamos consultar o arquivo para verificar que o símbolo **28** é o simbolo **else**. O exame da situação mostra que se trata realmente do conflito devido à forma de construção do comando **if**, isto é, um conflito empilha-reduz em que se dá preferência ao empilhamento. Assim, nenhuma ação corretora precisa ser tomada.

Outras informações constantes do mesmo arquivo dão algumas das dimensões que caracterizam o problema: temos 132 regras, das quais quase um terço constituído de regras simples; temos 65 símbolos não-terminais, 54 terminais e um total de 189 estados.

Encontramos também no mesmo arquivo a numeração dos símbolos terminais, que deve ser usada para consulta às tabelas e portanto na saída do analisador léxico.

O arquivo **pasx.trb** é então submetido ao programa compactador, sendo o resultado o constante dos arquivos **pasx.ren** e **pasx.tab**. O primeiro destes arquivos não será considerado aqui, uma vez que não há necessidade de alteração das tabelas. O arquivo **tabx.pas** foi construído a partir de **pasx.tab**, pela formação de uma **unit** de Turbo Pascal 4.0, da forma descrita anteriormente. A compilação dessa unidade gerou o arquivo **tabx.tpu**, com 3632 bytes.

Essas tabelas são consultadas pelo analisador sintático através de rotinas como as listadas abaixo. Este trecho de código

é incluído aqui apenas como exemplo, e deve ser revisto para cada aplicação.

```
program pascalX;
  uses     { incluir aqui                  }
    tabX,  { as tabelas do parser          }
    lexX,  { o analisador léxico scan      }
    semX;  { as ações semânticas rotsem    }
  var estp,estq:byte;
  const maxsin=200 ; { ou outro valor apropriado }
  var psin:array[1..maxsin]of byte;
    tsin:integer;
  function empilhou:boolean;
    var i:integer;
  begin { empilhou }
    empilhou:=false;
    if ((estq>=inipemp) and (estq<=fimpemp)) then begin
      i:=pemp[estq];
      repeat
        if acesso[vemp[i]]=ord(simb) then begin
          estq:=vemp[i];
          inc(tsin);
          psin[tsin]:=estq;
          scan; { atualiza  simb }
          empilhou:=true;
          exit;
        end else begin
          inc(i);
          if vemp[i]<0 then
            i:=-vemp[i];
        end;
      until i=-fim;
    end;
  end; { empilhou }
  function reduziu:boolean;
    var i,j:integer;
  begin { reduziu }
    reduziu:=false;
    if estq <= fimreduz then begin
      prod:=reduz[estq];
      tsin:=tsin-salto[estq];
      estp:=psin[tsin];
      i:=betaq[estq];
      repeat
        if vbeta[i]=ord(simb) then begin
          j:=vbeta[i+1];
          repeat
            if valfa[j]=estp then begin
              estq:=valfa[j+1];
              inc(tsin);
              psin[tsin]:=estq;
              rotsem(prod);
              reduziu:=true;
              exit;
            end else begin
```

```
                j:=j+2;
                if valfa[j]<0 then
                  j:=-valfa[j];
              end
            until j=-fim;
            i:=-fim;
          end else begin
            i:=i+2;
            if vbeta[i]<0 then
              i:=-vbeta[i];
          end
        until i=-fim;
      end else
        estp:=255;
  end; { reduziu }
  procedure errosin;
  begin
    writeln('Erro sintatico);
    halt;
  end; { errosin }
begin { pascalX }
  tsin:=1;
  psin[1]:=estinicial;
  estq:=estinicial;
  scan; { inicializa simb }
  repeat
    if not empilhou then
        if not reduziu then
          errosin
  until psin[tsin]=estfinal;
end. { pascalX }
```

V.5 Extensões previstas

Ao sistema descrito neste manual deverão ser acrescentados brevemente outros módulos, destinados a permitir a construção automática de avaliadores de atributos, que funcionarão acoplados com os analisadores construídos da forma aqui descrita **[Sant88]**.

Bibliografia

[AhSU86] Alfred V.Aho, Ravi Sethi, Jeffrey D. Ullman
Compilers, Principles, Techniques, and Tools
Addison Wesley, 1986

[Bor187] Borland International
Turbo Pascal Owner's Handbook

[Sant88] Jeferson S. Santana
Um construtor de avaliadores de atributos
Tese de Mestrado, DI/PUC-RJ, a ser apresentada

[Schn87] Sergio M. Schneider
Gramáticas e analisadores R*S
Tese de Doutorado,
Progr. Eng. Sistemas e Computação,COPPE/UFRJ,
setembro 1987