**Análise Sintática e
Recuperação de Erros
em Linguagens Determinísticas**

Alan M. Durham

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA OBTENÇÃO DO GRAU DE MESTRE
EM
MATEMÁTICA APLICADA

**Área de Concentração: Ciência da Computação
Orientador: Prof. Dr. Valdemar W. Setzer**

-SÃO PAULO, Julho de 1987-

Dedico este trabalho aos três professores que me levaram a escolher a carreira acadêmica, e pelos quais tenho toda a admiração: meu avô, meu pai, e minha mãe.

Agradecimentos

Gostaria de agradecer ao prof. Valdemar W Setzer pela sua orientação e ao departamento de Matemática Aplicada pelo apoio material. Gostaria de agradecer também aos meus alunos Marcia, Gisela e Ricardo pela colaboração na programação do sistema desenvolvido em paralelo com este trabalho. Gostaria de agradecer à Inês pela leitura dos originais. Gostaria de agradecer ao apoio especial de minha colega, Denise. Finalmente gostaria de fazer um agradecimento à minha mulher, Regina, que me ajudou a enfrentar os momentos difíceis. Obrigado.

Índice

Prefácio	i
Capítulo 1: Conceitos Fundamentais e Notações	1
1.1 Linguagens e Gramáticas	1
1.2 Operações sobre Cadeias	6
Capítulo 2: Analisadores do Tipo LR	8
2.1 Analisando uma Linguagem LR(1)	8
2.2 A Construção do An. de uma Gramática	12
2.3 Algoritmos de Construção do Analisador e das Tabelas	19
2.4 Analisadores SLR e LALR	22
2.5 Otimização de analisadores LR	24
2.6 Recuperação de Erros Sintáticos	26

Capítulo 3: Recuperação de Erros Local	29
3.1 Panic Mode	31
3.1.1 O Panic Mode Estático	31
3.1.2 O Panic Mode de Wirth	32
3.2 Os Métodos de Custo Mínimo	35
3.2.1 O Método de Custos Designados	35
3.2.2 O Método de Complementação	45
 Capítulo 4: Recuperação de Erros Regional	 57
4.1 A Recuperação por Padrões de Troca	60
4.2 A Recuperação por Redução de Cadeias	70
4.3 A Recuperação com Backtracking	78
4.3.1 O Método de Mikunas e Modry	82
4.3.2 O Método de Pennello e DeRemer	89
4.3.3 A Recuperação Não Corretiva	97
4.3.4 Métodos sem Condensação	98
 Capítulo 5: Recuperação de Erros Global	 107
5.1 A Recuperação Implícita	108
5.2 A Recuperação de Contexto Global	114
 Capítulo 6: Final	 119
6.1 A Recuperação em Analisadores de Produção	119
6.2 A Recuperação em Sistemas de Geração de Analisadores	122
 Bibliografia	 124
 Apêndice A	 129
 Apêndice B	 131

Prefácio

A construção de compiladores é muito simplificada pelo uso de analisadores sintáticos.

Uma classe muito conhecida de analisadores sintáticos é a dos analisadores LR. Uma das principais vantagens desta classe de analisadores é a sua capacidade de analisar qualquer linguagem determinística, desde que adequadamente descrita. O primeiro analisador LR foi desenvolvido por Knuth [4]. Baseados no algoritmo de Knuth surgiram vários outros analisadores extentendo ([6] [38] [39] [40] [41] [42] [43] [44] [45]) ou restringindo ([3] [5]) a capacidade de análise.

Um bom esquema de análise sintática deve contar com um algoritmo de recuperação de erros para que o programador possa, com apenas uma execução do analisador, detectar a maioria de seus eventuais erros sintáticos na descrição do programa.

O objetivo deste trabalho é fornecer um panorama sobre aos métodos de recuperação de erros em analisadores sintáticos do tipo LR, a saber, os analisadores LR, SLR e LALR.

No capítulo um são introduzidos os conceitos fundamentais sobre

gramáticas, linguagens e cadeias, utilizados no restante do texto.

No capítulo dois introduzimos o conceito de análise sintática, descrevemos a construção e o funcionamento de um analisador LR, definimos as classes de analisadores e linguagens SLR e LALR e introduzimos o problema da recuperação de erros, além de mencionarmos brevemente algumas otimizações que podem ser realizadas na implementação de analisadores do tipo LR.

A bibliografia existente sobre recuperação de erros em analisadores do tipo LR é extensa. Por este motivo decidimos dividir os métodos de recuperação em categorias, baseados na divisão feita por Hammond e Raymond Smith ref50. Esta divisão particiona os métodos de recuperação em três categorias: os métodos locais, descritos no cap.3, os métodos regionais, descritos no cap.4, e os métodos globais, descritos no cap.5.

Partindo do artigo dos autores acima citados, estendemos a bibliografia consultada e realizamos uma descrição detalhada dos métodos que consideramos mais significativos. Cada categoria de métodos de recuperação foi por nós dividida em sub-categorias, visando maior clareza na exposição.

Possivelmente a maior contribuição deste trabalho seja a uniformização da notação utilizada para a descrição dos algoritmos. Para isto nos baseamos parcialmente na notação utilizada por Pennello e DeRemer [35]. Os algoritmos são descritos em detalhe utilizando uma linguagem muito próxima ao PASCAL, mas com comandos em português. Procuramos, na medida do possível, não nos utilizarmos de comandos de nível muito alto, de maneira a facilitar uma possível implementação. Os comandos de nossa linguagem para os quais não existe contrapartida em linguagens de terceira geração (ALGOL, PASCAL, etc.) são em sua grande maioria aqueles que envolvem uma opção por uma representação específica de determinada estrutura de dados. Nestes casos utilizaremos uma estrutura de nível de abstração maior do que o das referidas linguagens. Desta maneira, uma implementação de nossos algoritmos envolveria antes a definição da representação das estruturas de Lista e Conjunto, juntamente com o funcionamento das primitivas de união, intersecção, pertinência, concatenação, etc.

Para que fosse possível a uniformização da notação, quase todos os algoritmos descritos sofreram modificações. Muitos deles não estavam nem ao menos descritos em uma linguagem formal (como por exemplo os métodos de Burke e Fisher [19], Mikunas e Modry [11] e de Sippu e

Soisalon-Soininen [25]).

No capítulo final tecemos algumas considerações sobre como deve ser na prática um esquema de recuperação de erros sintáticos para analisadores LR.

Este trabalho, devido à extensão do assunto, não pretende ser completo. Bibliografia adicional sobre análise sintática e recuperação de erros pode ser encontrado em [27] e [66]. Nem todos os itens bibliográficos da bibliografia do final do trabalho foram explicitamente citados no texto, tendo servido como leitura de apoio.

Paralelamente a este trabalho está sendo desenvolvida uma implementação de um gerador de analisadores LALR(1). Os analisadores gerados possuem um esquema de recuperação, denominado **recuperação não corretiva** cuja descrição pode ser encontrada em 4.3.3.

Capítulo 1

Conceitos Fundamentais e Notações

Neste capítulo descreveremos os conceitos fundamentais sobre gramáticas e linguagens que serão utilizados nos próximos capítulos. Um estudo mais aprofundado destes conceitos pode ser encontrado em [53] [55] [57].

1.1 Linguagens e Gramáticas

Seja Σ um conjunto, denominado alfabeto. Os elementos deste conjunto são chamados símbolos, sendo o conjunto de seqüências finitas de símbolos de Σ denotado por Σ^* . Os elementos de Σ^* são chamados palavras sobre o alfabeto Σ . A seqüência vazia será denotada por λ .

def 1.1.1 Uma linguagem sobre Σ é um subconjunto de Σ^* .

def 1.1.2 Uma gramática G é uma quádrupla (V_n, V_t, P, S) , onde:

a) V_n e V_t são alfabetos, denominados respectivamente de alfabeto dos símbolos não-terminais e alfabeto dos símbolos terminais. O conjunto $V = V_n \cup V_t$ por V , é o conjunto dos símbolos

b) $P \subseteq (V_n V^*) \times V^*$ é o conjunto das produções. Cada elemento $p = (\alpha, \beta)$ de P será denotado por $\alpha \rightarrow \beta$, onde α é o lado esquerdo da produção e β é o lado direito, denotados, respectivamente, por $LE(p)$ e $LD(p)$.

c) $S \subseteq V_n$ é o símbolo inicial

def 1.1.3 Dada uma gramática $G = (V_n, V_t, P, S)$ definimos uma relação $\Rightarrow$ denominada derivação direta na gramática G definida por:

$$\Rightarrow = \{ (\alpha u \beta, \alpha v \beta \mid \alpha, \beta \in V^* \text{ e } (u \rightarrow v) \in P \}$$

$\Rightarrow^*$ é o fecho transitivo e reflexivo de $\Rightarrow$

$\Rightarrow^+$ é o fecho transitivo de $\Rightarrow$

se $(u, v) \in \Rightarrow^+ \text{ ou } \Rightarrow^*$, denotaremos o fato por $u \Rightarrow^+ v$ ($u \Rightarrow^* v$).

def 1.1.4 $\alpha \beta \Rightarrow^+ \gamma v \beta$ é dita uma derivação mais à direita se $\beta \in V_t^*$.

O mesmo vale para $\Rightarrow^*$ e $\Rightarrow$. Denotaremos o fato por $\Rightarrow^+(rm)$ ($\Rightarrow^*(rm)$, $\Rightarrow(rm)$).

def 1.1.5 O conjunto das formas sentenciais de uma gramática $G = (V_n, V_t, P, S)$, denotado por $FS(G)$ é definido por:

$$FS(G) = \{ \alpha \in V^* \mid S \Rightarrow^+ \alpha \}$$

def 1.1.6 A linguagem gerada pela gramática $G = (V_n, V_t, P, S)$ denotada por $L(G)$ é dada por:

$$L(G) = FS(G) \cup V_t$$

chamaremos sentenças aos elementos de $L(G)$.

def 1.1.7 Uma gramática livre de contexto (GLC) é uma gramática $G = (V_n, V_t, P, S)$ t.q.

$$\forall (\alpha \rightarrow \beta) \in P, \alpha \in V_n.$$

def 1.1.8 Uma linguagem L é dita livre de contexto se existe uma GLC G tal que $L(G) = L$.

def 1.1.9 Seja $GG=(V_n, V_t, P, S)$ uma gramática, definimos, $\forall \alpha \in V^*$:

$$First_k(\alpha) = \{ \gamma \in V_t^* / (\alpha \xRightarrow{*} \gamma \text{ e } |\gamma| \leq k) \text{ ou } (\alpha \xRightarrow{*} \gamma\beta \text{ e } |\gamma| = k, \beta \in V_t^*) \}$$

def 1.1.10 Seja $G = (V_n, V_t, P, S)$ uma GLC onde não é verdade que $S \xRightarrow{+} S$. G é uma gramática LR(k) se:

$$\forall \omega, \omega', \mu \in V_t^*,$$

$$\forall \gamma, \alpha, \alpha', \beta, \beta' \in V^*,$$

$$\forall A, A' \in V_n, \text{ se}$$

$$i) S \xRightarrow{*} (rm) \alpha A \omega \Rightarrow (rm) \alpha \beta \omega = \gamma \omega$$

$$ii) S \xRightarrow{*} (rm) \alpha' A \mu \Rightarrow (rm) \alpha' \beta \omega' = \gamma \omega'$$

$$iii) First_k(\omega) = First_k(\omega')$$

então

$$i) A = A', \beta = \beta' \text{ (isto é, } A \rightarrow \beta \text{ e } A' \rightarrow \beta' \text{ são a mesma produção)}$$

$$ii) |\alpha\beta| = |\alpha'\beta'| \text{ (isto é, } \alpha = \alpha')$$

def 1.1.11 Uma GLC $G=(V_n, V_t, P, S)$ é determinística se, para qualquer palavra de $L(G)$ existe apenas uma sequência de derivações mais à direita a partir S terminando na palavra. Em outras palavras:

$$\forall \gamma \in L(G)$$

$$\text{se } S \Rightarrow (rm) \alpha_1 \Rightarrow (rm) \dots \Rightarrow (rm) \alpha_n = \gamma \text{ e}$$

$$S \Rightarrow (rm) \beta_1 \Rightarrow (rm) \dots \Rightarrow (rm) \beta_m = \gamma$$

$$\text{então } n=m, \alpha_1=\beta_1, \dots, \alpha_n=\beta_m.$$

def 1.1.12 Uma GLC $G=(V_n, V_t, P, S)$ é reduzida se $\forall A \in V_n$, $\exists \alpha, \beta \in V^*$, $\gamma \in L(G)$ t.q $S \xRightarrow{*} \alpha A \beta \xRightarrow{*} \gamma$.

def 1.1.13 Uma GLC $G=(V_n, V_t, P, S)$ é aumentada se S não aparece do lado direito de nenhuma produção e se a única produção onde aparece do lado esquerdo é uma produção do tipo:

$$S \rightarrow Z\#,$$

onde '#' é um símbolo terminal que não aparece em nenhuma outra produção, símbolo este denominado de símbolo de fim de cadeia.

def 1.1.14 Uma GLC é ambígua se ela não é determinística.

def 1.1.15 Uma linguagem L livre de contexto é determinística se existe uma GLC G determinística tal que $L = L(G)$.

Teorema 1.1

Toda a gramática $LR(k)$ é determinística. ([51] pg. 505)

Teorema 1.2

Toda a linguagem determinística L tem uma gramática G $LR(1)$ tal que $L = L(G)$. ([51] pg. 512)

def 1.1.16 Um automato a pilha é uma 7-upla

$A_p = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ onde

- a) Q é um conjunto de estados representando um controle finito;
- b) Σ é um alfabeto de entrada;
- c) Γ é um alfabeto de trabalho;
- d) δ é um mapeamento de $Q \times \Gamma^* \times (\Sigma \cup \{\lambda\})$ nos subconjuntos finitos de $\Gamma^* \times \Sigma^*$;
- e) $q_0 \in Q$ é o estado inicial;
- f) $z_0 \in \Gamma$ é o símbolo que aparece inicialmente na pilha;
- g) $F \subseteq Q$ é o conjunto de estados finais.

def 1.1.17 Uma configuração de A é uma tripla (q, α, β) pertencente a $Q \times \Gamma^* \times \Sigma^*$ onde:

- a) q representa o estado atual do controle finito;
- b) α representa o conteúdo da pilha (Se $\alpha = \alpha'z$, onde $z \in \Gamma$, dizemos que z é o topo da pilha).
- c) β representa a porção não utilizada da cadeia de entrada;

Para representarmos o funcionamento de um autômato a pilha utilizaremos a relação ' $<<$ ' entre configurações.

def 1.1.18 Definimos uma transição de um autômato finito como uma relação denotada por ' $<<$ ' e definida por:

$$(q, \alpha z, \beta) << (q', \alpha \gamma, \beta) \text{ sse } (q', \gamma) \in \delta(q, z, \beta),$$

onde $q \in Q$, $b \in \Sigma \cup \{\lambda\}$, $\beta \in \Sigma^*$, $z \in \Gamma$, $\alpha, \gamma \in \Gamma^*$

Consideremos $<<^*$ e $<< +$ como os fechos reflexivo-transitivo e transitivo de $<<$, respectivamente.

Uma configuração inicial é uma configuração da forma (q_0, z_0, β) , onde $\beta \in \Sigma^*$.

Uma configuração final é uma configuração do tipo (q, α, λ) onde $q \in F$ e $\alpha \in \Gamma^*$.

Uma cadeia de símbolos β é aceita por um autômato a pilha A_p se $(q_0, z_0, \beta) <<^* (q, \alpha, \lambda)$ para algum $q \in F$ e algum $\alpha \in \Gamma^*$.

A linguagem reconhecida por um autômato a pilha A_p , denotada por $L(A_p)$, é o conjunto das cadeias aceitas por A_p .

def 1.1.19 Um autômato a pilha $A_p = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ é um autômato a pilha determinístico (APD) se, $\forall q \in Q$ e $\forall z \in \Gamma$:

a) $\delta(q, z, b)$ contém, no máximo, um elemento para cada $b \in \Sigma$ e $\delta(q, z, \lambda) = \emptyset$

ou

b) $\delta(q, z, b) = \emptyset \forall b \in \Sigma$ e $\delta(q, z, \lambda)$ contém no máximo um elemento.

def 1.1.20 Um autômato a pilha estendido é uma 7-upla $A_{pe} = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ onde:

a) δ é um mapeamento de um subconjunto finito de

$Q \times \Gamma^* \times (\Sigma \cup \{\lambda\})$ nos subconjuntos finitos de $Q \times \Gamma^*$

e

b) os outros símbolos são como na definição de autômato a pilha

obs: As definições de configuração, transição e linguagem aceita por um autômato a pilha estendido são análogas às aquelas feitas para autômato a pilha.

Teorema 1.3

Seja $G = (N, \Sigma, P, S)$ uma GLC. Pode-se construir um AP estendido R t.q. $L(R) = L(G)$. ([55] pg. 180)

Como pudemos observar pelo teorema acima, qualquer que seja a GLC G , é possível construir um autômato a pilha estendido que reconhece se determinada cadeia de símbolos é uma sentença da gramática.

No contexto das linguagens de programação, existe particular interesse em linguagens livres de contexto determinísticas. Por esta razão, estaremos interessados em reconhecedores de linguagens geradas por gramáticas determinísticas.

No próximo capítulo descreveremos um algoritmo de construção de um autômato a pilha determinístico estendido a partir de uma gramática LR(1), de maneira que a linguagem reconhecida pelo autômato seja igual a linguagem gerada pela gramática. Em seguida serão descritas duas restrições a este algoritmo, cujo objetivo é tornar o autômato mais compacto.

1.2 Operações sobre cadeias

Definiremos agora uma série de operações e funções sobre cadeias que utilizaremos ao longo deste trabalho.

Seja Σ um conjunto finito e sejam $\alpha = a_1 a_2 \dots a_n$ e $\beta = b_1 b_2 \dots b_m$ duas cadeias onde $a_1, \dots, a_n, b_1, \dots, b_m \in \Sigma, n \geq 0, m \geq 0$, então:

def 1.2.1 $\text{Prim}(\alpha) = a_1$ se $n > 0$
 λ se $n = 0$

def 1.2.2 $\text{Inicio}(\alpha) = a_1 a_2 \dots a_{n-1}$ se $n > 0$
 λ se $n = 0$

def 1.2.3 $\text{Ult}(\alpha) = a_n$ se $n > 0$
 λ se $n = 0$

def 1.2.4 $\text{Final}(\alpha) = a_2 a_3 \dots a_n$ se $n > 0$
 λ se $n = 0$

def 1.2.5 $\alpha \text{ cat } \beta = a_1 a_2 \dots a_n b_1 b_2 \dots b_m$

def 1.2.6 A norma de uma cadeia γ , denotada por $|\gamma|$, é o número de símbolos que a compõe, desta maneira :

$$|\alpha| = n \text{ e } |\beta| = m$$

def 1.2.7 Uma operação de edição sobre a cadeia α é um mapeamento $Oe(\alpha)$ t.q.:

$Oe(\alpha) = a_1 \dots a_{i-1} a_{i+1} \dots a_n$ para algum $1 \leq i \leq n$

ou

$Oe(\alpha) = a_1 \dots a_{i-1} c a_{i+1} \dots a_n$ para algum $1 \leq i \leq n$
e algum $c \in \Sigma$

ou

$Oe(\alpha) = a_1 \dots a_{i-1} c a_i \dots a_n$ para algum $1 \leq i \leq n$
e algum $c \in \Sigma$

no primeiro caso denominamos a operação de edição de eliminação, no segundo de troca e no terceiro de inserção.

def 1.2.8 A distância de Hamming entre α e β , denotada por $d_h(\alpha, \beta)$ é o número mínimo de operações de edição que deve ser aplicado à cadeia α para obter a cadeia β .

Capítulo 2

Analísadores do tipo LR

O processo de construção de todas as formas sentenciais intermediárias entre o símbolo inicial de uma gramática G e uma sentença de $L(G)$ é chamado **análise sintática**.

Neste capítulo descreveremos inicialmente um algoritmo de geração de analisadores sintáticos para linguagens $LR(1)$ e em seguida definiremos algumas restrições e extensões às gramáticas $LR(k)$, juntamente com metodologias de geração de analisadores para estas novas classes de gramáticas.

2.1 Análise uma linguagem $LR(1)$

Os algoritmos que descreveremos a seguir foram desenvolvidos para gerar analisadores para gramáticas $LR(k)$, qualquer que seja o número k , uma vez fornecido um algoritmo para se computar, para qualquer símbolo

não terminal, o conjunto das cadeias de terminais de comprimento k que podem sucedê-lo numa forma sentencial (chamaremos este conjunto de 'look-ahead', seguindo a terminologia inglesa).

Como dissemos no capítulo anterior (teorema 1.2), toda a linguagem livre de contexto determinística tem uma gramática LR(1) que a gera. Desta maneira, para simplificar as descrições, utilizaremos sempre $k=1$.

No chamados analisadores sintáticos do tipo LR a análise é feita reconhecendo as formas sentenciais utilizadas na derivação mais à direita da cadeia de terminais a partir do símbolo inicial da gramática. As formas sentenciais são reconstruídas de "trás para diante", isto é, a partir da cadeia de símbolos vamos reconstruindo as formas sentenciais até obtermos a forma sentencial que contém apenas o símbolo inicial da gramática. Quando isso for possível reconhecemos a cadeia como pertencente à linguagem gerada pela gramática dada.

Para reconstruirmos as formas sentenciais vamos ler a cadeia de símbolos da esquerda para a direita, armazenando os símbolos lidos numa estrutura de pilha. Quando algum sufixo da pilha corresponde a um lado direito de uma produção da gramática, então este sufixo é substituído pelo lado ^{esquerda} direito da referida produção se a pilha resultante da operação, concatenada com o restante da cadeia, formarem uma forma sentencial da linguagem. Se a gramática que gera a referida linguagem for LR(1) isto sempre pode ser decidido investigando-se a pilha e o primeiro elemento da porção não analisada da cadeia (ver def 1.1.10).

Ex: Seja a gramática:

$$S \rightarrow aAb \mid aCd$$

$$A \rightarrow aA \mid d$$

$$C \rightarrow cA \mid cc$$

A análise da cadeia 'acaadb' ficaria :

Pilha	Cadeia
	'acaadb'
'a'	'caadb'
'ac'	'adb'
'aca'	'db'
'acaa'	'b'
'acaad'	

'acaaA'	'b'
'acaA'	'b'
'acA'	'b'
'aC'	'b'
'aCb'	"
S	"

obs: os sufixos da pilha que são lados direitos de produções estão grifados.

Descreveremos agora um analisador de linguagens LR(1). Este analisador foi inicialmente desenvolvido por Knuth [3], porém, para seu relato nos basearemos naquele feito por Harrison [51].

Um analisador sintático LR de uma gramática reduzida aumentada $G = (V_n, V_t, P, S)$ é um autômato a pilha extendido $AN = (Q, V_t, (Q \times V), \delta, Z, e_f)$. Os elementos da pilha deste autômato são duplas compostas de um símbolo da gramática e da representação de um estado do controle finito. Para maior clareza trataremos a pilha do autômato como duas pilhas paralelas que denominaremos **pilha sintática** (contendo os símbolos) e **pilha do analisador** (contendo a representação dos estados).

A ação do analisador será determinada por duas funções, que chamaremos de Atitude e Destino, cujos parâmetros serão o primeiro símbolo da cadeia de entrada e o estado do topo da pilha do analisador. São quatro os possíveis valores da função Atitude: redução, aceitação, avanço e erro. A aceitação corresponde ao reconhecimento da produção relativa ao símbolo inicial da gramática e indica que a cadeia de símbolos da fita de entrada era uma sentença de $L(G)$, onde G é a gramática em questão. O avanço corresponde a detecção de que o símbolo de entrada, concatenado ao final da pilha sintática, forma com ela o prefixo de uma forma sentencial da linguagem. Redução corresponde à identificação de um sufixo da pilha sintática como o lado direito de uma produção. Finalmente, erro é uma ação especial que coloca o analisador em um estado de rejeição, o que indica que a cadeia analisada não pertence à linguagem, ou se preferirmos, que os símbolos da pilha sintática, concatenados com o primeiro símbolo da cadeia de entrada não constituem um prefixo de uma forma sentencial da gramática.

ilustração de analisador

O mapeamento δ do autômato a pilha estendido no qual se constitui o analisador sintático de uma linguagem LR(1) não será definido, ficando porém implícita sua forma pela definição da relação ' $\ll$ ' definida mais abaixo.

Num analisador LR(1) o estado atual do controle finito é sempre aquele que está representado no topo da pilha do analisador. Desta maneira, definiremos uma configuração de um analisador LR de maneira ligeiramente diversa daquela de um autômato a pilha genérico.

def 2.1.1 Uma configuração (CA) de um analisador LR é uma tripla (α, γ, β) onde:

α é o conteúdo da pilha do analisador

γ é o conteúdo da pilha sintática

β é o restante da cadeia de entrada

def 2.1.2 A configuração inicial de um analisador LR é definida por:

(e_0, λ, ω) , onde e_0 é o estado inicial do analisador, e ω a cadeia a ser analisada.

O funcionamento do analisador é definido pela relação de transição ' $\ll$ ' em configurações do analisador. Conforme já foi dito esta relação será determinada pelo conteúdo do topo da pilha do analisador e pelo próximo símbolo da cadeia de entrada. Para descrevermos a relação acima precisamos definir duas funções:

def 2.1.3 Seja $A_n = (Q, V_t, V_n \cup V_t, q_0, (q_0, \lambda), \{q_f\})$ um analisador sintático

gramática $G = (V_n, V_t, P, S)$ reduzida extendida. Definimos então as seguintes funções:

- 1) Atitude : $\Gamma \times V_t \rightarrow \{ \text{avanço, erro} \} \cup \{ \text{reduz } \pi / \pi \in P \}$
- 2) Destino : $\Gamma \times V \rightarrow \Gamma \cup \{ '? \}$

def 2.14 Seja (α, γ, β) uma CA então:

- a) se $\text{Atitude}(\text{Ult}(\alpha), \text{Prim}(\beta)) = \text{avanço}$ então:
 $(\alpha, \gamma, \beta) << (\alpha \text{ cat } \text{Destino}(\text{Ult}(\alpha), \text{Prim}(\beta)), \gamma \text{ cat } \text{Prim}(\beta), \text{Final}(\beta))$
- b) se $\text{Atitude}(\text{Ult}(\alpha), \text{Prim}(\beta)) = \text{reduza } A \rightarrow \mu$ então:
 seja $\gamma = \gamma' \gamma''$, e $\alpha = \alpha' \alpha''$, onde $/\gamma''/ = /\alpha''/ = /\mu/$, temos:
  - b.1) se $\alpha' = \epsilon_0$ e $\beta = \lambda$ então $(\alpha, \gamma, \beta) << (\alpha', \gamma', \beta) << (\text{aceita})$
 senão
  - b.2) se $\text{Atitude}(\text{Destino}(\text{Ult}(\alpha'), A), \text{Prim}(\beta)) = \text{erro}$
 então $(\alpha, \gamma, \beta) << (\text{erro})$
 senão
  - b.3) $(\alpha, \gamma, \beta) << (\alpha', \gamma', A \text{ cat } \beta) << (\alpha' \text{ cat } (\text{Destino}(\text{Ult}(\alpha', A)), \gamma' \text{ cat } A, \beta)$

2.2 A Construção do Analisador de uma gramática

Para construirmos os estados do analisador e calcularmos as funções Atitude e Destino, utilizaremos alguns conjuntos construídos a partir de produções da gramática reduzida aumentada $G = (V_n, V_t, S, T)$ aos quais denominaremos conjuntos de itens.

def 2.2.1 Dada uma gramática $G = (V_n, V_t, P, S)$, definimos, $\forall a \in V$ os seguintes conjuntos:

$$\text{Follow}_k(a) = \{ \delta \in V_t^* / \exists \alpha, \beta \in V^* \text{ t.q. } \alpha a \delta \in \text{FS}(G) \text{ e } /\delta/ = k \}$$

$$\text{Follow}(a) = \text{Follow}_1(a)$$

def 2.2.2 Um item LR(1) de uma GLC $G = (V_n, V_t, P, S)$ é uma dupla da forma:

$$(A \rightarrow \gamma \cdot \delta; u), \text{ onde } \gamma, \delta \in V^*, (A \rightarrow \gamma \delta) \in P, \text{ e } u \subseteq \text{Follow}(A).$$

A definição de item vem da necessidade de representar o reconhecimento parcial de um lado direito de uma produção. Desta maneira utilizaremos o item $(A \rightarrow \gamma.\delta; u)$ para representar o fato de que γ é um sufixo da pilha sintática, indicando o reconhecimento parcial da produção em questão. Além disto o conjunto u , denominado **conjunto de lookahead** indicará quais os símbolos que podem suceder a A caso o lado direito da produção venha a ser completamente reconhecido.

def 2.2.3 Um item redução é um item da forma $(A \rightarrow \mu. ; u)$

def 2.2.4 Um estado LR(1) de uma gramática é um conjunto finito de itens LR(1) desta mesma gramática.

A definição de estado LR(1) vem de encontro à necessidade de representar, em um dado ponto da análise de uma cadeia de entrada, quais as produções que podem vir a ter seu lado direito reconhecido, e em que ponto está o reconhecimento destas produções.

def 2.2.5 Um item LR(1) válido de uma gramática $G(V_n, V_t, P, S)$ é um item LR(1) da forma $(A \rightarrow \gamma.\delta; u)$ para o qual existe uma forma sentencial α tal que $S \xRightarrow{*} (rm)\alpha$ e $\alpha = \alpha'\gamma\delta a\alpha''$, onde $\alpha', \alpha'' \in V^*$, e $a \in u$.

Os itens LR(1) válidos de uma gramática são aqueles que correspondem a um possível reconhecimento parcial do lado direito de uma produção durante a análise de uma sentença de G .

def 2.2.6 Um estado LR(1) de uma gramática $G = (V_n, V_t, P, S)$, composto dos itens $I_1, \dots, I_n$ é válido se existem formas sentenciais $\alpha_1 \dots \alpha_n$ de G tais que, para $1 \leq i \leq n$:

- a) $S \xRightarrow{*} (rm)\alpha_i$
- b) $\alpha_i = \mu\alpha_i'$
- c) considerando $I_i = (A_i \rightarrow \gamma_i.\delta_i ; u_i)$:
  - c.1) $\mu = \mu_i\gamma_i$
  - c.2) $\alpha_i' = \delta_i a_i \alpha_i''$, onde $a_i \in u_i$

Um estado é válido se os seus itens são todos válidos e se podem corresponder a um mesmo instante de uma análise LR(1); ele representa um possível momento da análise sintática, indicando quais produções podem vir a ter seus lados direitos reconhecidos.

O conjunto de produções, bem como o tamanho do lado direito de cada uma delas é finito. O conjunto de símbolos terminais é finito e logo

também o número de possíveis conjuntos de lookahead. Desta maneira, o número de possíveis itens LR(1) e portanto de possíveis estados LR(1) é finito para uma determinada GLC. Assim é possível, na prática, o cálculo do conjunto de estados LR(1) de uma dada gramática. Veremos agora um algoritmo que calcula todos os estados LR(1) válidos para uma GLC reduzida estendida LR(1) qualquer.

Para calcularmos os estados válidos utilizaremos a seguinte técnica: Inicialmente construímos o estado válido no instante inicial da análise, quando a pilha sintática está vazia. A partir deste estado teremos uma idéia de quais lados direitos podem vir a ser reconhecidos. O reconhecimento de qualquer um destes lados direitos implica que o seu primeiro símbolo deve ser reconhecido. A partir então de cada símbolo à direita do ponto em algum item do estado inicial podemos construir um estado válido correspondente ao momento da análise sintática onde a pilha conterá apenas o referido símbolo. Repetimos este processo para os estados recém criados criando os estados válidos correspondentes a momentos da análise onde a pilha sintática contem dois elementos, e assim por diante. Como o número de estados válidos é finito este processo termina quando todos os estados válidos tiverem sido calculados. Vejamos um exemplo:

ex

Seja a GLC reduzida aumentada G definida por:

$S \rightarrow Z\#$
 $Z \rightarrow Ck$
 $Z \rightarrow aAd$
 $Z \rightarrow bAB$
 $A \rightarrow cA$
 $A \rightarrow c$
 $B \rightarrow d$
 $C \rightarrow e$

Para maior clareza chamaremos de $V(\gamma), \gamma \in V^*$ o estado válido referente à pilha sintática contendo γ .

Inicialmente calcularemos $V(\lambda)$. Um item que certamente pode ser incluído neste estado é aquele que indica que o lado direito da produção de Z é esperado. Desta maneira inicialmente temos:

$$V(\lambda) = \{(S \rightarrow .Z\# ; \{ \})\}$$

o conjunto de look-ahead é vazio porque nenhum símbolo pode suceder ao indicador de final de cadeia.

Porém se o próximo símbolo pode ser o não terminal Z é igualmente verdade que podemos esperar os símbolos iniciais dos lados direitos das produções onde Z é lado esquerdo. Analogamente, sempre que houver um item de um estado com o ponto imediatamente antes de um símbolo não-terminal, devemos acrescentar a este estado itens relativos às produções deste não terminal, indicando que seu reconhecimento é esperado mas ainda não foi iniciado. Chamaremos a isto de **ação de complementação**. Os itens de um estado que não são gerados pela ação de complementação são chamados de **itens base** daquele estado. Vejamos o aspecto final de $V(\lambda)$:

$$V(\lambda) = \left\{ \begin{array}{l} (S \rightarrow .Z\# ; \{ \}) \\ (Z \rightarrow .Ck ; \{ \# \}) \\ (Z \rightarrow .aAd ; \{ \# \}) \\ (Z \rightarrow .bAB ; \{ \# \}) \\ (C \rightarrow .e ; \{ k \}) \end{array} \right\}$$

Para exemplificar o cálculo dos conjuntos de lookahead vejamos o caso do item $(C \rightarrow .e ; \{ k \})$. Utilizamos o fato de que seu item antecessor era $(Z \rightarrow .Ck ; \{ \# \})$, indicando que, caso o lado direito de C seja reconhecido, só k pode ser o próximo símbolo terminal. A partir de $V(\lambda)$ verificamos que o conjunto de prefixos de tamanho 1 de formas sentenciais da gramática G é $\{ Z, C, a, b, e \}$, pois como indica o ponto nos diversos itens de $V(\lambda)$, estes são os próximos símbolos que podem ocorrer a partir do prefixo de forma sentencial ao qual este estado está associado, no caso o prefixo de comprimento zero. Deste modo estas são as possíveis configurações da pilha sintática quando ela tiver tamanho 1. Por isto construiremos em seguida os estados válidos $V(Z)$, $V(C)$, $V(a)$, $V(b)$ e $V(e)$:

$$\begin{aligned} V(Z) &= \{ (S \rightarrow Z.\# ; \{ \}) \} \\ V(C) &= \{ (Z \rightarrow C.k ; \{ \# \}) \} \\ V(a) &= \left\{ \begin{array}{l} (Z \rightarrow a.Ad ; \{ \# \}) \\ (A \rightarrow .cA ; \{ d \}) \\ (A \rightarrow .c ; \{ d \}) \end{array} \right\} \\ V(b) &= \left\{ \begin{array}{l} (Z \rightarrow b.AB ; \{ \# \}) \\ (A \rightarrow .cA ; \{ d \}) \end{array} \right\} \end{aligned}$$

$$V(e) = \left\{ \begin{array}{l} (A \rightarrow .c ; \{d\}) \\ (C \rightarrow .e. ; \{k\}) \end{array} \right\}$$

No lookahead de um item que não foi gerado pela ação de complementação colocamos o lookahead do item que o originou. Assim, em $V(C)$, o item $(Z \rightarrow C.k; \{\#\})$ tem o mesmo lookahead que o item $(Z \rightarrow .Ck ; \{\#\})$ do estado $V(\lambda)$, isto porque o lado direito procurado é o mesmo e provém do mesmo contexto da análise.

No estado $V(b)$ temos um novo caso de cálculo de lookahead: Como o item $(Z \rightarrow bAB ; \{\#\})$ indica que o não terminal A deve ser seguido pelo não terminal B , o lookahead dos itens gerados pela ação de complementação neste item terão como lookahead o conjunto $\text{First}(B)$. Caso $B \xrightarrow{*} \lambda$ então deveríamos acrescentar ao lookahead os símbolo ' $\#$ ' que é o unico que poderia suceder B nas circunstâncias.

Repetimos agora o princípio aplicado em $V(\lambda)$ para, a partir dos estados válidos referentes a configurações possíveis de tamanho 1 da pilha, construir os estados válidos referentes a possíveis configurações de tamanho 2 da pilha sintática. Destaca-se o fato de que o estado $V(e)$ não dá origem a nenhum outro pois tem um único item que indica o reconhecimento completo do lado direito de uma produção.

$$\begin{aligned} V(Z\#) &= \{ (S \rightarrow Z\#.; \{\}) \} \\ V(aA) &= \{ (Z \rightarrow aA.d; \{\#\}) \} \\ V(ac) &= \left\{ \begin{array}{l} (A \rightarrow .c.A ; \{d\}) \\ (A \rightarrow .c. ; \{d\}) \\ (A \rightarrow .cA ; \{d\}) \\ (A \rightarrow .c ; \{d\}) \end{array} \right\} \\ V(bA) &= \left\{ \begin{array}{l} (Z \rightarrow bA.B ; \{\#\}) \\ (B \rightarrow .d ; \{\#\}) \end{array} \right\} \\ V(bc) &= \left\{ \begin{array}{l} (A \rightarrow .c.A ; \{d\}) \\ (A \rightarrow .c. ; \{d\}) \\ (A \rightarrow .cA ; \{d\}) \\ (A \rightarrow .c ; \{d\}) \end{array} \right\} \\ V(Ck) &= \{ (Z \rightarrow Ck. ; \{\#\}) \} \end{aligned}$$

Podemos notar que os estados $V(ac)$ e $V(bc)$ são idênticos, podendo ser reduzidos a um estado somente.

Vejam os estados relativos a configurações de tamanho 3 da pilha sintática:

$$V(aAd) = \{ (Z \rightarrow aAd. ; \{\#\}) \}$$

$$V(acc) = V(ac)$$

$$V(acA) = \{ (A \rightarrow cA. ; \{d\}) \}$$

$$V(bAd) = \{ (B \rightarrow d. ; \{\#\}) \}$$

$$V(bAB) = \{ (Z \rightarrow bAB. ; \{\#\}) \}$$

$$V(bcA) = V(acA)$$

$$V(bcc) = V(ac)$$

Finalmente a partir dos estados relativos a configurações de tamanho 3 da pilha sintática, calculamos os estados válidos relativos às configurações de tamanho 4:

$$V(accc) = V(ac)$$

$$V(accA) = V(acA)$$

$$V(bccc) = V(bc)$$

$$V(bccA) = V(bcA)$$

Como podemos notar nenhum estado novo foi introduzido. Desta maneira para todas as outras possíveis configurações da pilha durante a análise sintática de uma sentença da gramática existe um estado válido entre os já gerados que descreve quais produções podem vir a ter seu lado direito reconhecido.

O conjunto de estados LR(1) válidos para uma determinada gramática nos fornecem uma representação das possíveis análises parciais de lados direitos de produções da gramática que podem ocorrer na pilha sintática. Vejamos agora como utilizar o conjunto de estados válidos para a análise sintática de uma cadeia de símbolos.

No nosso analisador sintático LR(1), os elementos da pilha do analisador serão estados válidos da gramática. Desta maneira, as funções Atitude e Destino serão definidas a partir de um estado válido e de um símbolo terminal. Vejamos:

Atitude($V(\alpha), a$) = avanço se $I = (A \rightarrow \gamma.a\delta; u) \in V(\alpha)$ e $a \neq \#$
 reduz $B \rightarrow \mu$ se $I = (B \rightarrow \mu.; u) \in V(\alpha)$ e $a \in u$
 aceita se $a = \#$ e $I = (Z \rightarrow Z.\#; \{\}) \in V(\alpha)$
 erro caso contrário.

Destino($V(\alpha), a$) = $V(\alpha a)$ se $V(\alpha a)$ é um estado LR(1) válido
 '?' caso contrário.

Podemos notar pelo processo de construção acima que cada estado de um analisador LR(1) pode ser atingido somente através da atitude avanço sobre um único símbolo. Desta maneira o estado $V(\alpha)$ será sempre atingido quando o analisador encontrar o símbolo $Ult(\alpha)$ à frente da cadeia de entrada quando estiver no estado $V(Início(\alpha))$. Partindo deste fato podemos encurtar a descrição da configuração de um analisador LR(1), fornecendo apenas a pilha do analisador e a porção restante da cadeia de entrada, já que a pilha sintática pode ser deduzida a partir da pilha do analisador, uma vez que a cada estado corresponde apenas um símbolo de acesso. Para a descrição dos algoritmos no restante deste trabalho utilizaremos esta configuração reduzida. Denotaremos o símbolo de acesso a cada estado q por $Sim(q)$. Nos exemplos utilizados no texto utilizaremos estratégia inversa: descreveremos as configurações através da pilha sintática e do restante da cadeia de entrada, por entendermos que esta estratégia os torna mais compreensíveis.

Para a representação dos estados válidos de um analisador designaremos um número inteiro a cada um deles, por ordem de criação de estados. Desta maneira o estado $V(\lambda)$ terá o número 0 e os outros os números 1, 2, ... etc. Denotaremos o estado de número i por e_i . As funções Atitude e Destino serão representadas cada uma por uma tabela, contendo uma linha para cada estado e uma coluna para cada símbolo. É possível que, para uma dada gramática, o método de construção acima indique mais de um valor para a função Atitude para um mesmo par estado-símbolo. Neste caso dizemos que ocorre um conflito, que indica que a gramática em questão não é LR(1) e portanto que o analisador não pode funcionar de maneira determinística. Existem maneiras de se conviver com estes conflitos estabelecendo princípios de escolha entre as várias atitudes, existindo inclusive implementações práticas, como é o caso do famoso YACC [16]; outra solução pode ser encontrado em [1]. Porém, não abordaremos a questão neste trabalho.

Para a descrição dos algoritmos dos capítulos seguintes utilizaremos algumas funções sobre estados e itens que definiremos abaixo:

def 2.2.7 Seja q um estado de um analisador LR(1):

$Base(q)$ = conjunto dos itens base do estado q

$Item(q, a) = \{(A \rightarrow \gamma \cdot \delta; u) \in q \mid (a = Prim(\delta)) \text{ ou } (\delta = \lambda \text{ e } a \in u)\}$

def 2.2.8 Seja $I = (A \rightarrow \gamma \cdot \delta; u)$ um item:

$$LE(I) = A \quad LD(I) = \gamma\delta \quad \text{Não\_rec}(I) = \delta \quad \text{Lookahead}(I) = u$$

ex

A tabela de Atitudes do analisador da gramática do exemplo anterior pode ser encontrada no apêndice B, localizado no final do texto.

2.3 Algoritmos de construção do analisador e das tabelas

Nesta seção descreveremos os algoritmos de construção dos estados de um analisador sintático LR(1) para uma dada GLC reduzida aumentada e o algoritmo de análise sintática que delas se utiliza.

Para a descrição do algoritmo de construção dos estados LR(1) válidos e das tabelas das funções utilizaremos dois algoritmos auxiliares. Adotando a terminologia inglesa vamos chamá-los de "scanner" e "completer".

O primeiro deles recebe como dados um estado $V(\alpha)$ e um símbolo 'a' da gramática, construindo os itens base de $V(\alpha a)$ se este for um estado LR(1) válido. Sua saída será composta de tres parâmetros: o primeiro retornará $V(\alpha a)$ se este for um estado LR(1) válido, e $\emptyset$ caso contrário; o segundo retornará o número de itens onde o ponto está no final do lado direito e para os quais 'a' pertence ao lookahead; o último parâmetro retornará a produção relativa ao último item daqueles contados no segundo parâmetro. Podemos notar que se o segundo parâmetro for maior que 1 ou se o seu valor for 1 simultaneamente com um valor não vazio do primeiro parâmetro então temos uma ambiguidade. O algoritmo "completer" executa a ação de complementação sobre os itens de um estado fornecido. Sua função é complementar a ação do "scanner" na geração do conjunto de estados LR(1) válidos.

Algoritmo Scanner

entrada: $V(\alpha)$ - um estado LR(1) válido

a - um símbolo de $V_n \cup V_t$

saída: Estado - variável contendo $V(\alpha a)$ se este for um estado válido

e $\emptyset$ caso contrário

Red - número de itens de $V(\alpha)$ com o ponto no fim do lado direito
e com o símbolo a em seu lookahead

Produção - produção do último item encontrado dos citados acima

com1

Estado := $\emptyset$;

para cada item $I = (A \rightarrow \gamma.\delta; u) \in V(\alpha)$

faça se $\text{Prim}(\delta) = a$

então Estado := Estado $\cup \{(A \rightarrow \gamma a.\text{Final}(\delta); u)\}$

senão se $\delta = \lambda$ e $a \in u$

então com2

Red := Red + 1;

Produção := $(A \rightarrow \gamma\delta)$

fim2

fim1.

Algoritmo Completer

entrada: C-conjunto de itens

$G = (V_n, V_t, P, S)$ - GLC

saída: C-aumentado pela ação de complementação

com1

para cada $I = (A \rightarrow \gamma.\delta; u) \in C$ faça

se $B = \text{Prim}(\delta) \in V_n$

então para cada $(B \rightarrow \mu) \in P$ faça

com2

$\Gamma := (B \rightarrow \cdot\mu; \text{First}(\delta u))$

se existe $\Gamma' = (B \rightarrow \cdot\mu; w) \in C$

então $\Gamma' := (B \rightarrow \cdot\mu; w \cup \text{First}(\delta u))$

senão $C := C \cup \{(B \rightarrow \cdot\mu; \text{First}(\delta u))\}$

fim2

fim1.

Algoritmo Gera Tabelas e Estados

entrada: $G = (V_n, V_t, P, S)$ - GLS reduzida extendida

saída: Q- conjunto dos estados LR(1) válidos da gramática G
 Aceita - tabela com os valores da função Aceita
 Destino- tabela com os valores da função Destino

com1

$e_0 := \{ (Z \rightarrow .S\#; \{ \}) \} ; (* e_0 \text{ é } V(\lambda) *)$

Completer(e_0);

$Q := \{e_0\};$

ult := 0; (* ult indica índice do último estado *)

para i de 0 até ult faça

para cada $a \in V_n \cup V_t$ faça

com2

Scanner($q_i, a, q_{ult+1}, \text{Red}, \text{Prod}$);

se $q_{ult+1} = \emptyset$

(*se o scanner não construiu um novo estado*)

então se Red = 1

então se Prod = $S \rightarrow Z$

então Atitude(q_i, a) := aceita

senão Atitude(q_i, a) := reduz Prod

senão Atitude(q_i, a) := erro

senão se Red = 0

então **com3**

Atitude(q_i, a) := avanço;

se $\exists j$ t.q. $q_j \in Q$ e $q_j = q_{ult+1}$

então Destino(q_i, a) := q_j

senão **com4**

$Q := Q \cup \{q_{ult+1}\} ;$

Destino(q_i, a) := q_{ult+1} ;

ult := ult + 1

fim4

fim3

senão Atitude(q_i, a) := erro;

fim2

fim1.

Algoritmo Analisador LR

entrada: Atitude - tabela com os valores da função Atitude

Destino - tabela com os valores da função Destino
 β - cadeia de entrada

saída: Listagem - lista contendo as configurações antes de
 cada redução efetuada na pilha

Sucesso - variável booleana indicando se a cadeia analisada
 pertence à linguagem gerada pela gramática em questão

com1

$\alpha := e_0; (*\alpha$ é a pilha do analisador*)
 $\gamma := \lambda; (*\gamma$ é a pilha sintática*)
 Erro:=falso;(*erro indica entrada no estado de erro*)
 Sucesso:=falso;(*ainda não acabou a análise*)
 enquanto não Sucesso e não Erro faça
   caso Atitude(Ult(α),Prim(β)) seja:
     avanço:com2
        $\alpha := \alpha \text{ cat Destino(Ult}(\alpha), \text{Prim}(\beta))$;
        $\beta := \text{Final}(\beta)$
     fim2
   reduz($A \rightarrow \gamma$):
     com2
       Listagem:=Listagem cat (α, γ, β);
       (* agora vai retirar da pilha o lado dir da prod*)
       para I de 1 até $|\delta|$ faça
         com3
            $\gamma := \text{Início}(\gamma)$;
            $\alpha := \text{Início}(\alpha)$
         fim3;
        $\alpha := \alpha \text{ cat Destino(Ult}(\alpha), A)$;
        $\gamma := \gamma \text{ cat } A$;
     fim2
   aceita: Sucesso:=verdade
   erro: Erro:=verdade
   fimcaso;
 Sucesso:= não Erro;
 fim1.

2.4 Analisadores SLR e LALR

Os analisadores sintáticos LR(k), apesar do amplo espectro das linguagens que são capazes de analisar, apresentam um problema na implementação: o número de estados é muito grande em relação ao número de produções, exigindo o algoritmo grandes quantidades de memória.

Neste sentido, em 1969 DeRemer [54] define duas novas classes de gramáticas, as gramáticas LALR e SLR. Ambas as classes surgem para definir as gramáticas analisadas por analisadores gerados por modificações dos algoritmos vistos na seção anterior. Em ambos os casos os analisadores resultantes terão o mesmo número de estados que um analisador LR(0) para a gramática dada. Na verdade, a metodologia de análise será a mesma que a dos analisadores LR, sendo que as modificações se restringem aos algoritmos de geração dos estados e das tabelas Atitude e Destino. Utilizaremos para a descrição destas modificações três definições auxiliares:

def 2.4.1 O núcleo de um item $(A \rightarrow \delta.\gamma; u)$ é a porção $A \rightarrow \delta.\gamma$ do item.

def 2.4.2 Dois estados de um analisador LR(K) são semelhantes se os conjuntos de núcleos de ambos são iguais

Um analisador LR(0) é um analisador gerado pelos algoritmos descritos no item anterior sem o cálculo de lookahead, isto é, um analisador onde os itens são compostos apenas de núcleo. Em analisadores LR(0), todo o estado que possui um item redução e um outro item qualquer é um estado com conflito, devido à ausência do lookahead.

Os analisadores SLR(k) são analisadores com os mesmos estados de um analisador LR(0) onde, nos estados com conflito, acrescenta-se, a cada item redução $(A \rightarrow \mu.)$, o conjunto $Follow_k(A)$ no lugar que caberia ao lookahead deste item, se estivessemos num analisador LR(k). As gramáticas para as quais este algoritmo gera conjuntos de estados sem conflito são chamadas gramáticas SLR(k).

def 2.4.3 Dados dois estados semelhantes:

$$e_a = \{(A_1 \rightarrow \gamma_1.\delta_1; u_1) \dots (A_n \rightarrow \gamma_n.\delta_n; u_n)\}$$

$$e_b = \{(A_1 \rightarrow \gamma_1.\delta_1; v_1) \dots (A_n \rightarrow \gamma_n.\delta_n; v_n)\}$$

então a junção de estados por núcleos de e_a e e_b é definida como :

$$e_j = \{(A_1 \rightarrow \gamma_1.\delta_1; v_1 \cup u_1) \dots (A_n \rightarrow \gamma_n.\delta_n; v_n \cup u_n)\}$$

O analisador LALR(K) de uma gramática G é um analisador onde os estados são resultantes da junção de estados por núcleos de estados semelhantes do analisador LR(k) da mesma gramática. Uma gramática LALR(k) é uma gramática para a qual o analisador LALR(k) não apresenta estados conflitantes. Esta nossa definição de analisadores LALR é oriunda do método de construção de analisadores LALR desenvolvido por Anderson, Eve e Hornig [9]. Uma implementação deste tipo de construção se encontra no apêndice deste trabalho. Outras definições de analisadores LALR, bem como outras metodologias de construção podem ser encontradas em [3] [36] e [37].

2.5 Otimização de analisadores LR

Como lembram Aho e Johnson [1] e Bigonha [52], existem uma série de otimizações que podem reduzir o tamanho e/ou aumentar a rapidez dos analisadores sintáticos do tipo LR. Entre elas podemos citar a representação das tabelas de atitude e destino por listas, a junção e a imersão de estados, a adoção de reduções "default" e a eliminação seletiva de estados de redução.

A representação de estados por listas é uma estratégia que parte da observação do fato de que as tabelas que representam as funções Atitude e Destino são matrizes esparsas, se considerarmos, respectivamente, erro e '?' como elementos nulos. Torna-se assim muito mais eficiente em termos de espaço a sua representação através de listas de atitudes e destinos para cada estado, relacionando apenas os elementos não nulos. No caso específico da função Atitude, a ação 'erro' seria determinada quando fosse encontrado um símbolo de entrada para o qual não existe nenhuma ação determinada pela lista do estado do topo da pilha. Numa compactação ainda maior podemos juntar as duas tabelas numa só: a cada estado estaria associada uma lista cujos elementos teriam tres campos: um deles conteria a representação de um símbolo da gramática, outro campo conteria a atitude determinada pelo símbolo do primeiro campo no estado ao qual a lista se refere, e o último campo conteria um número. Caso a atitude indicada no segundo campo seja avanço, o número do ultimo campo indicará qual o valor da função destino para o símbolo do primeiro campo e o estado ao qual a lista pertence, Caso atitude indicada seja redução o número indicará qual produção deve ser utilizada para a redução(as produções, por sinal, podem ter sua descrição comprimida, representando-se apenas o lado esquerdo e o comprimento do

lado direito). Todas as listas de todos os estados podem estar contidas em uma única estrutura em forma de vetor, existindo, para cada estado, dois apontadores indicando o início e o fim da respectiva lista no referido vetor. Em alguns casos podemos mesmo suprimir este segundo índice, utilizando o início do estado seguinte como delimitador.

As reduções "default" são aplicadas em estados com apenas um item redução. Nestes casos é possível, ao invés de acrescentar à lista de atitudes/destinos uma entrada para cada símbolo do lookahead do item em questão, indicando redução apropriada, acrescentar apenas uma única entrada ao final da lista indicando que a referida redução deve ser efetuada caso nenhuma outra atitude seja determinada. Isto torna a representação mais compacta, uma vez que as entradas para os símbolos do lookahead são substituídos por uma só. Não só isto como o processo de análise também é agilizado, já que a comparação do símbolo de entrada com os elementos do lookahead é eliminada. Este tipo de otimização abre a possibilidade de uma redução ser determinada mesmo quando o símbolo de entrada, concatenado à pilha sintática, não forma com ela um prefixo de uma forma sentencial da gramática, o que não ocorre normalmente em analisadores LR, devido à presença do lookahead. Isto poderá afetar a recuperação de um eventual erro sintático, como veremos adiante, mas não altera a linguagem reconhecida pelo analisador já que, como não houve modificação nas entradas relativas à atitude avanço, uma vez efetuadas a redução(ou cadeia de reduções) o analisador não será capaz de prosseguir a análise sobre o símbolo errado, já que, como já dissemos, ele não forma com a pilha sinática um prefixo de uma forma sentencial da gramática.

A eliminação seletiva de estados de redução consiste em eliminar os estados que possuem um único item, quando este item é um item redução de uma produção cujo lado direito é composto de um único símbolo. Maiores detalhes sobre esta estratégia podem ser encontrados em [1] [58] e [59].

Finalmente temos a junção e a imersão de estados. A junção consiste em representar os valores das funções Atitude e Destino de dois estados diferentes através de uma única lista, quando estes valores são idênticos. A imersão de estados é, na verdade, uma generalização da otimização anterior pois consiste em unir as representações das listas de dois estados quando a lista de um deles constitui um sufixo da lista do outro.

Para uma descrição mais detalhada destas e de outras otimizações o leitor pode consultar [1], [58], [59], [60], [61], [62], [63], [64], [65] e [66].

2.6 Recuperação de erros sintáticos

No processo de desenvolvimento de algoritmos invariavelmente chegamos ao estágio de programação. Neste estágio os algoritmos são codificados em alguma linguagem de programação e compilados para serem futuramente testados. Raríssimas são as vezes onde um programa compilado pela primeira vez não contém nenhum erro de sintaxe. Para que a análise sintática do programa não seja interrompida quando da ocorrência do primeiro erro, é necessário que o analisador sintático seja restaurado a um estado no qual a análise possa continuar. Desta maneira torna-se possível, se houver um diagnóstico apropriado, encontrar vários erros de sintaxe em apenas uma execução do algoritmo de análise sintática. Este processo de restauração do analisador sintático chama-se **recuperação de erros**.

A análise sintática do tipo LR, como vimos anteriormente, vai, a cada passo, verificando se o símbolo de entrada, concatenado com os símbolos já analisados, formam um prefixo da linguagem, isto é, formam um prefixo de uma sentença da linguagem que o analisador pretende reconhecer. Porém não se sabe em geral de que sentença estamos reconhecendo o prefixo, e sim quais as possíveis sentenças que tem o texto já analisado como prefixo. Este fato é expresso pelo fato de que, em geral, cada estado do analisador tem mais de um item, cada um correspondendo a um possível prefixo. Desta maneira, quando ocorre um erro são múltiplas as possibilidades de corrigi-lo pois são múltiplos os possíveis prefixos a serem reconhecidos. Mais que isso, uma escolha que de imediato pareça plausível pode, com o decorrer da análise, mostrar-se inviável com os símbolos subsequentes indicando que a correção levou em conta uma possibilidade errada de prefixo. Estas escolhas "erradas" causam o que se chama "erros espúrios", isto é, erros que não existem na verdade, sendo consequência de uma recuperação mal feita de um erro sintático anterior.

Como vimos então, o processo de recuperação de erros é importante para agilizar a fase de programação de algoritmos. Além disso o processo de recuperação em analisadores LR não é simples, sendo difícil obter uma recuperação ideal dadas as várias possibilidades. Provavelmente isto explica a multiplicidade de métodos de recuperação desenvolvidos e a profusão da bibliografia publicada a respeito nos últimos 15 anos.

Com o advento dos micro computadores, voltaram à cena os analisadores sintáticos onde a recuperação de erros inexiste, limitando-se o pro-

grama de análise a indicar o local do primeiro erro ocorrido encerrando a análise sintática. Como o processo é geralmente feito em tempo real e não em "batch", a alternativa de não recuperar os erros não é tão dolorosa ao programador. Este princípio, aliás, em programas pequenos é bem atraente, mas ao analisarmos programas maiores é enfadonho e acaba tomando um tempo muito maior do programador.

A medida que o tamanho da memória e a velocidade de processamento dos "micros" vai aumentando, torna-se possível e desejável a construção de analisadores com um bom esquema de recuperação de erros.

Vejam os então o que é uma boa recuperação de erros: A qualidade da recuperação de erros envolve dois aspectos básicos que são, de certa maneira, antagônicos: qualidade e eficiência.

A qualidade é o aspecto mais difícil de se avaliar pois obedece em geral a critérios subjetivos; a maioria dos autores (como [25], [24], [12]), na avaliação de seus métodos, utiliza como padrão de qualidade da recuperação a proximidade que ela tem em relação à correção que seria feita por um programador experimentado. Alguns autores ([24], [21]) na tentativa de tornar mais objetiva a avaliação de qualidade, parte do pressuposto que a melhor correção é aquela que envolve o menor número possível de operações de edição sobre o texto original, isto é, o menor número possível de inserções, eliminações e trocas de símbolos. Estes autores desenvolvem metodologias para encontrar a correção com menor "distância de Hanning" em relação ao texto original.

Já a eficiência tem critérios mais objetivos. Ela é medida em três pontos: a rapidez com que o algoritmo é executado, na memória necessária à execução do algoritmo e na maneira pela qual o algoritmo de recuperação implica a análise de programas sintaticamente corretos.

Geralmente uma maior qualidade na correção acarreta uma menor eficiência e vice-versa. Além disso, os autores não se preocupam com uma medida teórica precisa do tempo e memória gastos pelos seus algoritmos, limitando-se, com raras exceções, a fornecer resultados práticos obtidos por alguma implementação. Estes aspectos tornam muito difícil, senão impossível, uma real comparação entre todos os métodos pois esta se pautaria mais provavelmente em critérios subjetivos. Por este motivo nos absteremos de fazer comparações entre métodos, limitando-nos a alguns comentários relativos a problemas específicos e soluções encontradas quando dois métodos forem suficientemente parecidos para que isto seja

feito.

Para descrevermos os vários métodos de recuperação vamos dividi-los em três classes, seguindo a divisão proposta por Hamond e Rayward Smith [46], que julgamos adequada.

A primeira classe é a dos métodos locais, aqueles que na recuperação consideram apenas um símbolo de cada vez, sem nenhuma preocupação com o contexto no qual ele está inserido. A segunda classe é a dos algoritmos regionais, aqueles que consideram um contexto limitado ao redor da região onde foi detectado o erro. Finalmente, a terceira classe é a dos algoritmos globais, aqueles que durante a correção consideram o contexto total, isto é, o programa inteiro, para efetuar as correções.

Nos próximos capítulos abordaremos as diversas classes de métodos de recuperação de erros, descrevendo vários métodos em detalhe. Da mesma maneira que não descrevemos analisadores de linguagens não determinísticas não descreveremos algoritmos de recuperação para eles, mas o leitor interessado poderá consultar [21] [31] [34].

Capítulo 3

Recuperação de Erros Local

Os métodos locais de recuperação de erros levam em consideração apenas um símbolo de entrada e um estado da pilha de cada vez.

Nesta categoria está grande parte dos métodos utilizados na prática, pois geralmente são rápidos e de fácil programação, não envolvendo tampouco grandes recursos de memória.

Infelizmente, como a cada passo podem ser múltiplas as escolhas de recuperação, cada uma geralmente correspondendo ao reconhecimento parcial de uma produção relativa a um não terminal diferente, é difícil a escolha entre as várias opções sem levar em consideração o contexto. Além disto é possível que o local onde o erro foi detectado não corresponda à localização real do erro. Com isso aumentam os riscos de ocorrência de erros espúrios.

Ex No trecho de um programa

```

.
.
C:=D;
A ≠ B then A:=B; ...

```

A configuração de erro do analisador seria:
 (... 'iden', '≠' 'B' 'then' A:=B; ...)

Neste caso, numa correção local, a única opção seria trocarmos '≠' por ':='. Mas isto fará com que o analisador detecte outro erro quando 'then' for o símbolo de entrada, gerando um erro espúrio.

Ex : no trecho

```

prog A;
var A,B:integer;
begin
  A:=b; ;
end.

```

A configuração de erro será:

('prog' 'iden' ';' <vardecl> 'begin' <stat> ';' , ';' 'end' '.')

Se considerarmos apenas o símbolo de entrada várias são as opções como eliminar o ';' ou inserir um iniciador de comando, como 'if' ou 'for' ou então trocar ';' por algum iniciador de comando. Neste caso específico podemos ver que a melhor sugestão é a primeira, mas como o algoritmo de recuperação local pode tomar esta decisão sem levar em consideração o contexto ao redor do erro?

O problema da escolha de alternativas foi solucionado a nível local com a introdução de custos diferenciados para inserção e eliminação dos diversos símbolos, sendo a escolha orientada para a alternativa de menor custo. Os custos são determinados pelo implementador visando tornar certas opções de correção pouco razoáveis muito custosas, como a eliminação da palavra 'type' em um programa PASCAL. Existe porém, o problema da atribuição dos diversos custos, pois não existe nenhum critério realmente objetivo, apenas algumas heurísticas advindas da intuição dos autores que utilizam este tipo de esquema.

Cabe ainda uma última observação sobre a recuperação local: todos os métodos desta categoria , quando não mantém a pilha do analisador

inalterada, não executam nela modificações que não sejam a eliminação de estados de seu topo. Por trás desta atitude está a suposição de que o erro sintático é detectado imediatamente, ou seja, quando o local do erro for o símbolo de entrada. Esta suposição se baseia no fato de que analisadores do tipo LR possuem a **propriedade do prefixo correto**, definida como a propriedade da porção já analisada da cadeia de entrada constituir sempre um prefixo de uma sentença da linguagem em questão. Porém, a suposição acima faz com que não se considere corretamente erros como o do primeiro exemplo, onde o prefixo correto “esconde” o local real de ocorrência do erro. Este problema infelizmente não pode ser resolvido satisfatoriamente a nível local, pois sua solução implicaria considerarmos um certo contexto ao redor do **ponto de detecção do erro (PDE)** para a recuperação.

3.1 “Panic Mode”

Chamaremos de panic mode os métodos que trabalham apenas com eliminação de símbolos. Esta eliminação é feita inicialmente no primeiro símbolo da cadeia de entrada, prosseguindo até que seja encontrado um símbolo que pertença a um conjunto diferenciado. Este conjunto de símbolos, ao qual denominaremos **conjunto de recuperação**, pode ser totalmente definido pelo implementador ou ter parte de sua composição decidida durante a análise sintática.

Os panic mode são os métodos de recuperação de erros mais simples e menos acurados.

3.1.a) O Panic Mode Estático

Nesta versão, o conjunto de recuperação é fixo durante a análise (daí a alcunha “estático”), pois é definido previamente pelo implementador. A rotina de recuperação, ao ser acionada, vai consumindo os símbolos de entrada até encontrar um que pertença ao conjunto de recuperação. Em seguida a pilha é reconfigurada de maneira que o estado do topo aceite o novo símbolo de entrada.

Algoritmo Panic-Mode Simples

entrada: (α, β) - configuração de erro do analisador

C_Rec - conjunto de recuperação
 saída: (α', β') -nova configuração do analisador, onde
 α' é um prefixo de α e β' é um sufixo
 de β .

com1

enquanto $\text{Prim}(\beta) \notin C\_Rec$
 faça $\beta := \text{Final}(\beta)$;
 enquanto não $\text{Aceita}(\text{Ult}(\alpha), \text{Prim}(\beta))$
 faça $\alpha := \text{Inicio}(\alpha)$;

fim1

Apesar de sua simplicidade e rapidez, o método geralmente fornece recuperações fracas, “pulando” trechos longos da cadeia de entrada, efeito que pode ser desastroso se o programa analisado estiver na fase de declarações. Além disso, seu funcionamento depende da escolha do conjunto de recuperação. Neste conjunto, costumam ser colocados símbolos do tipo ‘;’ e ‘end’, que são delimitadores de trechos importantes do programa.

3.1b) O Panic Mode de Wirth

Este método foi concebido por N. Wirth [46] para analisadores sintáticos do tipo recursivo descendente, para os quais se ajusta perfeitamente. Como é possível, com um pequeno esforço, sua implementação em analisadores do tipo LR, passaremos a descrevê-lo.

A diferença entre o método de Wirth e o panic-mode estático está na construção do conjunto de recuperação. O cálculo deste conjunto é feito em duas etapas distintas. Na primeira delas calculamos, para cada símbolo não-terminal, o conjunto dos símbolos terminais que podem sucedê-lo numa forma sentencial da gramática, isto é o conjunto Follow (ver definição no cap.2). A segunda etapa do cálculo é feita durante a análise sintática: o conjunto de recuperação começa com o símbolo de fim de cadeia (‘#’) e toda vez que é iniciado o reconhecimento de um novo não-terminal, o seu conjunto de possíveis sucessores (Follow) é acrescentado ao conjunto de recuperação. Quando um não terminal é reconhecido, o seu conjunto de possíveis sucessores é retirado do conjunto de recuperação. Cuidado deve ser tomado para, no reconhecimento de um não terminal, não retirarmos do conjunto de recuperação símbolos que ali deveriam permanecer por serem

sucessores de outros não-terminais cujo reconhecimento foi iniciado mas não completado.

A rotina de erro ao ser acionada, vai eliminando os símbolos de entrada até que encontre um que pertença ao conjunto de recuperação. Neste momento, manipulando a configuração do analisador, simula o reconhecimento do não-terminal do qual o símbolo encontrado é sucessor, retornando o controle ao analisador sintático.

Vejamos como isto é feito em analisadores LR:

Para a manipulação do conjunto de recuperação utilizaremos uma pilha auxiliar, paralela à pilha do analisador, que denominaremos **pilha de recuperação** ou simplesmente **P_Rec**. Cada elemento desta pilha conterá dois campos :a descrição de um conjunto de símbolos terminais e uma tabela associando cada um destes terminais um símbolo não terminal.

Toda vez que o analisador realiza um avanço, é feito o cálculo do novo elemento do topo da pilha de recuperação. Se o avanço é determinado somente por itens base (gerado pelo "scanner"), apenas copia-se no novo topo de P_Rec o conteúdo do topo anterior. Se o avanço é determinado também por um item não base, ou seja, da forma $(A \rightarrow \cdot \beta, \{a_1 \dots a_n\})$, então é possível que se tenha iniciado o reconhecimento de um novo símbolo não terminal. Neste caso, conjunto de terminais do novo topo de P_Rec é dado pela união do conjunto do topo anterior com o conjunto $\{a_1 \dots a_n\}$ do item, já que este contém justamente os símbolos terminais que podem suceder o não-terminal do lado direito da produção em questão. . As entradas de índice $a_1, a_2, \dots, a_n$ da tabela do topo de P_rec conterão o valor 'A' e as entradas restantes o mesmo valor da tabela do topo anterior. Para que isto seja possível é necessário associar a cada símbolo que determina um avanço em um estado estas informações, quando este avanço for determinado também por um item não base. Descreveremos as modificações na parte relativa à ação avanço do algoritmo de análise para que P_Rec seja corretamente manipulada.

$\alpha := \alpha \text{ cat } \text{Prox}(\text{Ult}(\alpha), \text{Prim}(\beta));$ (*atualiza pilha*)

$\beta := \text{Final}(\beta);$

para cada $I \in \text{Item}(\text{Ult}(\alpha), \text{Prim}(\beta))$ **faça**

(* I recebe o item responsável
pelo avanço*)

se $I \notin \text{Base}(\text{Ult}(\alpha))$ (*se i é gerado pelo "completer"*)
então


```

com
  P_Rec:=P_Rec cat Ult(P_Rec);
  (*copia topo anterior*)
  Ult(P_Rec).Conj:=Ult(P_Rec).Conj ∪ Lookahead(I);
  (*atualiza conjunto de terminais*)
  para cada a∈Lookahead(I) faça
    Ult(P_Rec).Tab[a]:=LE(I)
    (*atualiza tabela*)
fim
senão P_Rec:=P_Rec cat Ult(P_Rec);

```

Vejamos agora o algoritmo do panic-mode de Wirth modificado para analisadores LR:

Algoritmo Panic Mode de Wirth

```

entrada : ( $\alpha, \beta$ )-configuração do analisador
         P_Rec - pilha de recuperação
saida:   ( $\alpha, \beta$ )-nova configuração do analisador
com1
  enquanto Prim( $\beta$ )  $\notin$  Ult(P_Rec).Conj
    faça  $\beta :=$  Final( $\beta$ ); (*procura símbolo do conj. de rec.*)
  Aux:=Ult(P_Rec).Tab[Prim( $\beta$ )]
  enquanto Atitude(Ult( $\alpha$ ), Aux) = erro
    faça com2
       $\alpha :=$  Inicio( $\alpha$ );
      P_Rec:=Inicio(P_Rec)
    fim2;
    (*procura estado que aceita o não terminal A*)
     $\beta :=$  Aux cat  $\beta$ ;
fim1

```

Esta versão do método de Wirth ainda possui uma vantagem sobre a versão original: em analisadores do tipo LR, como o lookahead é calculado na geração de cada estado, os conjuntos de recuperação são muito mais precisos. O método aqui descrito se aproxima do original se utilizarmos um analisador SLR, cujos lookaheads são calculados a partir da gramática

somente, sem levar em conta o fato de que o reconhecimento de um não terminal pode ser feito em contextos diferentes.

3.2 Os Métodos de Custo Mínimo

Na categoria de métodos de custo mínimo locais temos duas abordagens diferentes: uma delas utiliza os custos para impedir correções muito grosseiras, atribuindo custos altos a operações de edição que, em geral, não são razoáveis; a segunda abordagem visa, com a introdução de custos diminuir ao máximo o número de símbolos perdidos da cadeia de entrada, simultaneamente tentando encontrar uma continuação válida da cadeia já reconhecida. A primeira abordagem é a do método de custos de Fischer, Dyon e Mauney [8], e a segunda é a do método de Rörich [7].

3.2.1 O Método de Custos Designados

A idéia central do método é atribuir custos diferenciados às diversas operações de edição. Para fornecer estes custos ao algoritmo de recuperação, utiliza-se dois vetores auxiliares, definidos pelo implementador. Estes dois vetores devem ter uma entrada para cada símbolo terminal contendo, respectivamente, os custos de inserção e eliminação de cada um dos símbolos. O objetivo do método é encontrar uma série de operações de edição com custo mínimo que possibilite ao analisador retomar o seu trabalho. Como estamos buscando um mínimo, os custos de inserção devem ser todos positivos, para garantir a terminação do método. Se nosso interesse for apenas diminuir o número de operações de edição, basta fixarmos todos os custos iguais. Além disso, com uma pequena modificação no algoritmo podemos adaptá-lo para aceitar custos diferenciados de troca de símbolos, isto é, custos de troca diferente da soma dos custos de eliminação e inserção dos símbolos envolvidos.

O esquema de custos diferenciados introduz maior flexibilidade na correção de erros, possibilitando ao implementador uma certa "sintonização" do algoritmo de recuperação.

O funcionamento deste método é simples, baseado em dois algoritmos: O primeiro deles é um algoritmo de inserção de custo mínimo que, dada uma configuração de erro $(q_1 \dots q_n, a_1 \dots a_m)$, encontra $\mu \in V_t^*$ de menor

custo tal que $\text{sim}(q_1) \dots \text{sim}(q_n) \mu a_1$ constituem um prefixo de uma sentença da linguagem. Em outras palavras, o algoritmo encontra a cadeia de terminais de menor custo que faz a "ligação" da configuração atual da pilha ao símbolo de entrada. Pode ser que não exista μ nestas condições, neste caso o algoritmo indica seu insucesso.

O segundo algoritmo calcula a recuperação propriamente dita e para isto executa os seguintes passos: Inicialmente é calculada, utilizando-se o primeiro algoritmo, a cadeia de "ligação" entre configuração de erro da pilha do analisador e o símbolo de entrada. Em seguida, considerando como custo de uma recuperação a soma dos custos das operações de edição envolvidas, verifica-se a possibilidade de diminuir o custo da recuperação eliminando o símbolo de entrada e calculando a nova cadeia de ligação. Este processo é repetido até que não seja possível minimizar o custo de recuperação.

Para a descrição do algoritmo de inserção de custo mínimo serão necessárias algumas definições auxiliares:

def 3.2.1 Seja $\alpha \in V^*$, definimos o custo da cadeia α , denotado por $\text{Custo}(\alpha)$, pela igualdade:

$$\text{Custo}(\alpha) = \sum_{1 \leq i \leq n} C(a_i),$$

onde $\alpha = a_1, a_2, \dots, a_n$, $a_i \in V_t$ para $1 \leq i \leq n$, e $C(a_i)$ é o custo de inserção do símbolo terminal a_i designado pelo implementador.

def 3.2.2 $\forall a_1, a_2, \dots, a_n \in V_t^*$,

$\text{Min}\{\} = \text{'?'}'$

$\text{Min}\{a_1, \dots, a_n\} = z$ t.q. $z \in \{a_1, \dots, a_n\}$ e $\text{Custo}(z) \leq \text{Custo}(a_i)$, $i=1, 2, \dots, n$.

def 3.2.3 Definimos $S(x)$ como a cadeia de símbolos terminais de menor custo derivada a partir de x . Formalmente temos:

$$S(X) = \text{Min}\{\mu \in V_t^* / X \xrightarrow{*} \mu\}, \forall X \in V$$

$$S(X_1 \dots X_m) = S(X_1) \text{ cat } \dots \text{ cat } S(X_m), \forall X_1, \dots, X_m \in V.$$

def 3.2.4 Definiremos $E(\mu, a)$ como sendo o prefixo de menor custo de uma cadeia de símbolos terminais derivada de μ que tem a como símbolo sucessor. Formalmente:

$$E(Y, a) = \text{Min}\{\omega \in V_t^* / Y \xrightarrow{*} \omega a \alpha, \alpha \in V_t^*\},$$

$$\forall Y \in V, \forall a \in V_t.$$

$$E(Y_1 Y_2 \dots Y_m, a) = \text{Min} \{ S(Y_1 \dots Y_i) \text{ cat } E(Y_{i+1}, a) / 0 \leq i \leq m \}$$

Não é possível pré calcular as funções S e E para todos os possíveis argumentos, mas podem ser construídas tabelas com S(X) para todo o símbolo X da gramática e com E(Y,a) para todo $Y \in V$ e $a \in V_t$. Vejamos os algoritmos para a construção destas tabelas. Estes algoritmos foram desenvolvidos por Fischer, Milton e Quiring; maiores detalhes podem ser encontrados no seu artigo de 1980 [48].

Algoritmo Calcula S

entrada: C-vetor de custos de inserção dos símbolos, onde os custos dos símbolos terminais estão preenchidos com os custos designados pelo implementador
P-conjunto de produções da gramática

saída: S-vetor com os valores da função S(X)
C-vetor dos custos contendo agora também, para cada não terminal A, o custo da cadeia S(A)

com1

para cada $A \in V_n$
faça $C(A) := \infty$;
para cada $a \in V_t$
faça $S(a) := a$;
repita
Nãotroca := verdade;
para cada $p \in P$ faça
com2
 $A := LE(p)$;
se $Custo(LD(p)) < C(A)$
então **com3**
Nãotroca := falso
 $C(A) := Custo(LD(p))$;
(* novo custo de A-soma dos custos dos símbolos do lado direito *)
 $Aux := LD(p)$;
 $S(A) := \lambda$;

```

enquanto S(A) ≠ λ faça
    com4
        S(A) := S(A) cat S(Prim(Aux));
        Aux := Final(Aux);
    fim4 (*calculou S(A) a partir da
        da concatenação dos valores
        de S p/ lado dir*)
    fim3
    fim2
até que Não troca;
fim1.

```

Algoritmo Calcula E

entrada: S,C-vetores calculados pelo algoritmo anterior

saída: E-matriz da função $E(\hat{Y}, a)$, calculada para cada $Y \in V_n$ e para cada $a \in V_t$

```

com1
  (*inicializa a matriz E; '?' indica inexistência de cadeia*)
  para cada  $(A,a) \in (V \times V_t)$ 
    faça  $E(A,a) := '?'$ ;
  para cada  $a \in V_t$ 
    faça  $E(a,a) := \lambda$ ;
  repita
    Nãotroca := verdade;
    paracada  $(a,p) \in (V_t \times P)$  faça
      com2
        Sim := Prim(LD(p));
        Aux := Final(LD(p));
        Aux2 :=  $\lambda$ ;
        Minaux := 0;
        Cadmin :=  $\lambda$ ;
        repita
          se  $\text{Custo}(E(LE(p),a)) <$ 
              $\text{Custo}(Aux) + \text{Custo}(E(\text{Sim},a))$ 
          então
            com4

```



```

Cadmin:=Aux;
Simmin:=Sim;
Custmin:=Custo(Cadmin)+
           Custo(E(Simmin,a))

fim4;
Aux:=Aux cat Sim;
Sim:= Prim(Aux2);
Aux2:=Final(Aux2)
até que Aux = λ;(*eqto não utilizar
                todo o lado direito*)
se Cadmin≠λ(*achou solução?*)
então com3
    enquanto Cadmin≠λfaça
        com4
            E(A,a):=E(A,a)cat
                    S(Prim(Cadmin));
            Cadmin:=Final(Cadmin)

        fim4;
        E(A,a):=E(A,a)catE(Simmin,a);
        Nãotroca:=false
    fim3
fim2
até que Nãotroca;
fim1

```

A função Custo utilizada no algoritmo acima é uma função recursiva definida por:

$$\begin{aligned}
 \text{Custo}(X) &= C(X) \text{ se } X \in V \\
 &\infty \text{ se } X = '?' \\
 0 &\text{ se } X = \lambda \\
 C(\text{Prim}(X)) + \text{Custo}(\text{Final}(X)) &\text{ caso contrário}
 \end{aligned}$$

Com a matriz $E(X,a)$ e o vetor $S(X)$ definidos, o cálculo das funções $E(\mu,a)$ e $S(\mu)$, para todo $\mu \in V_t^*$ e todo $a \in V_t$ pode ser feito implementando-se as expressões das definições 3.2.3 e 3.2.4.

Para a descrição do algoritmo de inserção de custo mínimo é precisamos ainda de uma última função. Porém para compreendermos o seu

funcionamento é necessário compreender antes o funcionamento do algoritmo de inserção:

Suponhamos que a configuração do analisador no início do algoritmo de recuperação é (α, β) , onde $\alpha = q_1, \dots, q_n$ e $\text{Prim}(\beta) = x$.

No início o algoritmo verifica o estado do topo da pilha (q_n): para cada item básico deste estado, $(A \rightarrow \gamma.\delta; u)$, podemos calcular uma possível cadeia de inserção, $E(\delta, x)$. Mas isso não esgota todas as possibilidades, já que sempre pode ocorrer que $E(\delta, x) = ?$ em todos os itens do referido estado. Não só isso, mas também é possível que uma cadeia de menor custo seja encontrada inserindo a cadeia $S(\delta)$, que permite que a produção $A \rightarrow \gamma\delta$ seja reconhecida e em seguida examinando o estado onde o analisador se encontrará logo após a redução. Este estado é o estado q_j , onde $j = n - |\gamma|$. Um dos itens deste estado será o item $(A \rightarrow \gamma\delta; u)$, que chamaremos de item K.

Consideremos agora as maneiras de se completar os itens "predecessores" do item K, isto é, os itens que "geraram" o item K através da ação de complementação. Não podemos esquecer que um item pode ser gerado pela aplicação da complementação em mais de um item básico e que um item não básico também pode ser "predecessor" do item K. Para cobrirmos todas as possibilidades vamos construir, para cada estado q do analisador, um grafo orientado que denominaremos $\text{CG}(q)$ (do inglês "complection graph"). Os nós do grafo $\text{CG}(q)$ representarão itens de q . As arestas são definidas pela seguinte regra: se o item $L = (C \rightarrow \alpha; u)$ foi obtido pela aplicação da ação de complementação no item $J = (B \rightarrow \mu.C\tau; v)$, então existe uma aresta de L para J com rótulo τ . O rótulo representa a sequência de símbolos que podem ser reconhecidos quando a produção do item K for completada.

Resumindo, quando chegamos ao estado q_j , onde temos o item

$K = (A \rightarrow \gamma\delta; u)$; devemos considerar quais cadeias de símbolos podem suceder o reconhecimento do não terminal A . Estas cadeias são dadas pela concatenação dos rótulos dos passeios que saem do item K em $CG(q_j)$. Na verdade só precisamos considerar os passeios maximais, isto é, os caminhos que contém o maior número possível de arestas.

Definiremos então a função $LP(q, I, J)$ que, para todo o estado q , nos fornece todos os passeios que ligam o item I ao item J em $CG(q)$. Este conjunto nem sempre é finito (em caso de recursão na gramática, podem existir circuitos no grafo), o que dificulta a implementação da função. Podemos porém, sem perda de generalidade do resultado, fazer uma restrição adicional não mencionada no artigo original. Definimos a função LP restrita, ou $LPR(q, I, J)$, que calcula, em $CG(q)$ quais os caminhos (passeios sem circuitos) que ligam o item I ao item J . Como estamos interessados em encontrar cadeias de inserção de custo mínimo a restrição não compromete o resultado.

Vejamos então como fica o princípio de funcionamento do algoritmo de inserção de custo mínimo. No estado do topo da pilha (q_n) procuramos inicialmente os itens básicos, $(A \rightarrow \gamma.\delta; u)$, e calculamos todos os $E(\delta, x)$, com x sendo o símbolo de entrada. Em seguida, verificamos se existe um item $(A' \rightarrow \gamma'.\delta'; u')$ tal que:

$$\text{Custo}(S(\delta')) < \text{Min}\{\text{Custo}(E(\gamma, x)) / (A \rightarrow \gamma.\delta; u) \in q_n\}$$

neste caso consideraremos como candidatas à cadeia de inserção de custo mínimo as cadeias:

$$S(\gamma') \text{ cat } E(\Gamma, x), \text{ onde } \Gamma \in LPR(q_{n-|\gamma'|}, K, J), \\ \text{para } K \text{ e } J \text{ itens de } q_{n-|\gamma'|} \text{ e } K = (A' \rightarrow \gamma'.\delta'; u').$$

Novamente podemos repetir o mesmo raciocínio para o item $q_{n-|\gamma'|}$ e, após encontrarmos um item I e um caminho $\Gamma' \in LPR(q_{n-|\gamma'|}, K, I)$ tal que $\text{Custo}(S(\delta) \text{ cat } S(\Gamma'))$ é menor do que o custo da cadeia de inserção calculada até o momento, verificamos a possibilidade de melhorar a solução utilizando um estado anterior da pilha. Para que não seja necessário reconstruir o conteúdo da pilha a cada nova tentativa sobre os itens base, o algoritmo irá tratar todos os casos em "paralelo", desempilhando apenas um estado de cada vez. O algoritmo fica então com o seguinte aspecto:

Algoritmo Ins_Mín

entrada: (α, β) - configuração do analisador

S, E- tabelas de $S(X)$ e $E(X, a)$

CS, CE- funções que calculam, respectivamente, $S(\gamma)$ e $E(\gamma, a)$, $\forall \gamma \in V^*$, $\forall a \in V_t$

LPR- função que calcula os caminhos entre os rótulos. Os caminhos são representados pela concatenação dos seus rótulos

Predecessor - função que dado um item I e um estado q devolve o item de q que, pela ação do scanner, gerou o item I, se existir, e '?' caso contrário.

saída: Ins- cadeia de inserção de custo mínimo. Caso não exista cadeia de inserção o algoritmo devolve o valor '?'.

com1

$\alpha' := \alpha$;

$x := \text{Prim}(\beta)$;

Ins := '?';

$q := \text{Ult}(\alpha')$;

para cada $I \in \text{Base}(q)$ faça

com2

$\gamma := \text{Não\_Rec}(I)$; (* γ - parte não reconhecida do item I *)

$\text{Completador}(q, I) := \text{CS}(\gamma)$; (* $\text{CS}(\gamma) = S(\gamma)$ *)

se $\text{Custo}(\text{Ins}) > \text{Custo}(\text{CE}(\gamma, x))$

então $\text{Ins} := \text{CE}(\gamma, x)$;

fim2;

repita

$\text{Pode\_melhorar} := \text{falso}$;

$p := \text{Ult}(\text{Início}(\alpha'))$; (* p é o penúltimo estado da pilha *)

para cada $I \in \text{Base}(q)$ faça

se $\text{Custo}(\text{Completador}(q, I)) < \text{Custo}(\text{Ins})$

(* se posso melhorar solução com estado anterior *)

então

com2

$\text{Pode\_melhorar} := \text{verdade}$;

$K := \text{Predecessor}(q, I)$; (* K recebe o item de p que justifica a transição para o estado q *)

se $K \notin \text{Base}(p)$; (* se item K resulta da aplicação


```

da ação de complementação*)
então(*procura os caminhos do grafo *)
para cada  $J \in \text{Base}(p)$  faça
para cada  $\Gamma \in \text{LPR}(p, K, J)$  faça
com3
se  $\text{Custo}(\text{Completador}(p, J)) >$ 
 $\text{Custo}(\text{Completador}(q, I) \text{ cat } \text{CS}(\Gamma))$ 
então  $\text{Completador}(p, J) :=$ 
 $\text{Completador}(q, I)$ 
 $\text{cat } \text{CS}(\Gamma);$ 
se  $\text{Custo}(\text{Ins}) >$ 
 $\text{Custo}(\text{Completador}(q, I)$ 
 $\text{cat } \text{CE}(\Gamma, x))$ 
então  $\text{Ins} := \text{Completador}(q, I)$ 
 $\text{cat } \text{CE}(\Gamma, x);$ 
fim3
fim2
 $\alpha' := \text{Início}(\alpha');$ 
 $q := p; (*\text{vai para o estado anterior}*)$ 
até que não Pode_melhorar;
fim1

```

Como o algoritmo Inserção_Mínima em mãos, descreveremos finalmente o algoritmo de recuperação propriamente dito que, como já dissemos, vai procurar a solução mais "barata", eliminando símbolos da cadeia de entrada enquanto o custo da eliminação somado ao custo da nova cadeia de inserção de custo mínimo for menor que o custo da solução atual.

Algoritmo Recup_Custo_Mínimo

entrada: (α, β) : configuração de erro do analisador

C - vetor de custos de inserção

D - vetor de custos de eliminação

LPR- função que calcula os caminhos entre os rótulos. Os caminhos são representados pela concatenação dos seus rótulos

Predecessor - função que dado um item I e um estado q

devolve o item de q que, pela ação do scanner, gerou o item I , se existir, e '?' caso contrario.

saída: (α, β) - configuração do analisador após a recuperação

com1

El_teste:= λ ;
 Inserção:='?';
 Eliminação:= λ ;
 $\beta' := \beta$;
 Acabou:=falso;
enquanto não Acabou e $\beta' \neq \lambda$ **faça**

com2

se $\text{Custo\_El}(\text{El\_teste}) > \text{Custo}(\text{Inserção}) + \text{Custo\_El}(\text{Eliminação})$
 (* se eliminar mais símbolos torna a recuperação mais cara *)
então Acabou:=verdade (*impossível melhorar solução*)
senão

com3

Ins_min((α, β') , S, E..., Ins_teste);
 se $(\text{Custo}(\text{Ins\_teste}) + \text{Custo\_El}(\text{El\_teste})) < (\text{Custo}(\text{Inserção}) + \text{Custo\_EL}(\text{Cadeia\_Eliminada}))$
 (* se posso melhorar a recuperação eliminando símbolo *)
então com3

Inserção:=Ins_teste;
 Eliminação:= El_teste;
fim3; (*indica nova solução*)
 El_teste:=El_teste cat Prim(β');
 (*prepara o teste para eliminação do próximo símbolo*)
 $\beta' := \text{Final}(\beta')$

fim2

para I de 1 até /Eliminação/ **faça**
 $\beta := \text{Final}(\beta)$; (*executa a eliminação*)
 $\beta := \text{Inserção cat } \beta$;
 (* β recebe a nova configuração*)

fim1

A função Custo_El utilizada no algoritmo acima é uma função recursiva definida por:

$$\begin{aligned} \text{Custo\_El}(X) &= C\_E(X) \text{ se } X \in V \\ &\infty \text{ se } X = '?' \\ &0 \text{ se } X = \lambda \\ &C\_E(\text{Prim}(X)) + \text{Custo\_El}(\text{Final}(X)) \text{ caso contrário} \end{aligned}$$

onde $C\_E(X)$ é o custo de eliminação do símbolo X , fornecido pelo implementador.

3.2.b) O Método de Complementação

Rõrich propõe uma recuperação em duas fases. Na primeira fase encontra-se uma cadeia de terminais ω , tal que ω seja uma continuação válida do texto já analisado, formando com ele uma sentença da linguagem. Na segunda fase procuramos o primeiro símbolo da cadeia de entrada que, precedido de um prefixo de ω do texto já analisado, forma um prefixo da linguagem. Em outras palavras, se chamarmos de μ o texto já analisado e de β o restante da cadeia de entrada, procuramos:

- i) na primeira fase: $\omega t.q. \mu\omega \in L$
- ii) na segunda fase: $\beta' t.q. / \beta' /$ é mínima e
  - ii.1) $\beta = \beta' a \beta''$, $a \in V_t$
  - ii.2) $\omega = \omega' \omega''$
  - ii.3) $\mu \omega' a \in \text{Prefix}(L(G))$, G a gramática analisada

Quando for encontrado β' nas condições acima, o controle volta ao analisador, substituindo-se β por $\omega' a \beta''$. A cadeia β' é sempre encontrada pois ω e β têm o símbolo '#' como sufixo comum.

Para constuirmos a cadeia ω utilizaremos uma função $\text{Cont}(q)$ que, a cada estado q do analisador, associa um símbolo terminal aceito por este estado. Esta função deverá ser calculada de maneira que, para encontrarmos ω , bastará prosseguirmos a análise sintática utilizando em cada estado q o símbolo $\text{Cont}(q)$ como símbolo de entrada. Quando atingirmos o estado final, a concatenação dos símbolos utilizados para prosseguir a análise formará a cadeia desejada.

Algoritmo de Geração de ω (primeira fase)

entrada: (α, β) - configuração de erro

Atitude, Destino-tabelas do analisador
saída: ω - cadeia de continuação.
com1
  (*1*)
   $\omega := \lambda$;
  Sucesso := verdade;
   $\alpha' := \alpha$;
  enquanto não Sucesso
    faça com2
       $a := \text{Cont}(\text{Ult}(\alpha))$;
      (*2*)
      se Atitude($\text{Ult}(\alpha')$, a) = avanço
        então
          com3
             $\omega := \omega \text{ cat } a$;
             $\alpha' := \alpha' \text{ cat } \text{Prox}(\text{Ult}(\alpha'), a)$
          fim3
        senão
          se Atitude($\text{Ult}(\alpha')$, a) = aceita
            então Sucesso := verdade
            senão
              com3
                 $I := \text{Item\_Redução}(\text{Ult}(\alpha), a)$;
                para Aux **de** 1
                  até /LD(I)/
                  faça $\alpha' := \text{Início}(\alpha')$;
                   $\alpha' := \alpha' \text{ cat } \text{Prox}(\text{Ult}(\alpha'), \text{LE}(I))$
              fim3
          fim2
    fim1.

Após encontrarmos a cadeia ω , precisamos encontrar ainda qual prefixo da cadeia de entrada deve ser eliminado. Para isto construímos o conjunto:

$$A = \{ a \in V_t / \exists \omega' \text{ prefixo de } \omega \text{ tq } \mu\omega'a \in \text{Prefix}(L) \}$$

Denominamos este conjunto **conjunto de âncoras**. O cálculo do conjunto de âncoras é feito durante a geração de ω . Para isto basta

inserirmos no ponto do algoritmo indicado por (*1*) o comando ' $A:=\emptyset$ ' e no ponto indicado por (*2*) o trecho:

para todo $a \in V_t$ t.q. $\text{Atitude}(\text{Ult}(\alpha'), a) \neq ?$

faça se $a \notin A$

então com3

$A := A \cup \{ a \};$
 $\text{Cadeia}(a) := \omega$

fim3

Desta maneira a segunda fase da recuperação consiste em consumir símbolos da cadeia de entrada até encontrar um símbolo pertencente ao conjunto de âncoras. Neste momento, basta concatenar o prefixo de ω associado à âncora encontrada no início do restante da cadeia de entrada.

Algoritmo de Correção (segunda fase)

entrada: (α, β) - configuração de erro

A - conjunto de âncoras

Cadeia - vetor onde $\text{Cadeia}(a)$ contém o prefixo de ω associado a âncora ' a '.

saída: $(\alpha, \omega'a\beta)$ - nova configuração do analisador

com1

enquanto $\text{Prim}(\beta) \notin A$

faça $\beta := \text{Final}(\beta);$

$\beta := \text{Cadeia}(\text{Prim}(\beta)) \text{ cat } \beta;$

fim1

Resta-nos somente saber como construir $\text{Cont}(q)$. Para isto construiremos inicialmente uma sub-gramática, G_c , da gramática original G . Chamaremos G_c de **gramática de continuação**. O conjunto de símbolos terminais, não-terminais e o símbolo inicial de G_c serão os mesmos que os de G . A diferença entre as duas gramáticas residirá no conjunto de produções. Para cada símbolo não-terminal G_c conterá apenas uma produção onde este símbolo figura do lado esquerdo. Além disto não haverá recursão nas produções, seja ela direta ou indireta. Para a construção de G_c utilizaremos o seguinte algoritmo:

Algoritmo Custos

entrada: P - conjunto das produções de G

V_n, V_t - conjuntos de símbolos terminais e não terminais de G

C - vetor com uma entrada para cada símbolo de V_n e de V_t

saída: C - preenchido, onde $C(a)$ conterá o tamanho da menor cadeia de terminais derivável a partir de ' a ',
onde $a \in V_n \cup V_t \cup \{\lambda\}$

com1

$C(\lambda) := 0;$

para cada $a \in V_t$

faça $C(a) := 1;$ (*custo dos terminais é trivialmente 1*)

para cada $A \in V_n$

faça $C(A) := \infty;$

Muda := verdade;

enquanto Muda (*1*)

faça com2

Muda := falso;

para cada $p \in P$

faça se $C(LE(p)) > \sum_{q \in LD(p)} C(q)$

então com3

$C(LE(p)) := \sum_{q \in LD(p)} C(q);$
Muda := verdade

fim3

fim2

fim1.

TEOREMA 3.1

A cada passo do algoritmo acima, se $C(A) \neq \infty$, $A \in V$, então A deriva uma sequência de terminais com comprimento $C(A)$.

prova: Indução no número de passos.

LEMA 3.1

No fim da n -ésima execução do loop indicado por '(*1*)' do algoritmo, $\forall A \in V$, se $\exists \alpha \in V_t^* \text{ tq } A \xRightarrow{*} \alpha$ e $\text{prof}(T) \leq n$, onde T é a árvore de derivação de $A \xRightarrow{*} \alpha$, então $C(A) \neq \infty$.

prova: A demonstração é feita por indução em n .

COROLÁRIO

O algoritmo Custos termina em um número finito de passos.

prova: Se, dado $A \in V_n$, $\exists \alpha \in V_t^*$ tq $A \xRightarrow{*} \alpha$, isto é, se um dado não terminal é utilizável, então $\exists \beta \in V_t^*$ tal que $A \xRightarrow{*} \beta$ e a profundidade da árvore de derivação associada é menor do que $|V_n|$. Desta maneira, pelo lema 3.1, todos os não terminais utilizáveis receberam algum valor finito ao final de $|V_n|$ iterações. Pelo teorema 3.1, os símbolos não-terminais não utilizáveis não recebem nenhum valor finito, e os outros recebem um valor finito depois de, no máximo, $|V_n|$ iterações. Como o loop indicado por (*1*) só continua sendo executado se algum símbolo tem seu valor diminuído, o algoritmo pára. $\square$

TEOREMA 3.2

Ao final do algoritmo Custos, $\forall A \in V$,

$C(A)$ = comprimento da menor cadeia de terminais derivável de A .

prova: Se $A \in V_t$, trivial

Se $A \in V_n$, indução no número de passos da derivação mais curta de uma cadeia de terminais de comprimento mínimo.

Uma vez obtido o vetor C com os custos calculados de cada símbolo, escolhemos, para a gramática G_c , uma produção de G para cada símbolo não-terminal de maneira que o custo associado ao símbolo do lado esquerdo seja igual à soma dos custos dos símbolos do lado direito. Assim teremos, em G_c , para cada produção $(A \rightarrow c_1 \dots c_n)$, onde $c_i \in V$ que:

$$C(A) = \sum_{1 \leq i \leq n} C(c_i)$$

Este algoritmo é uma modificação do algoritmo criado por Rörich. No algoritmo original o custo dos símbolos era associado à profundidade da árvore de derivação menos profunda e não ao tamanho da cadeia de terminais mais curta derivável a partir do símbolo. Como o algoritmo Custos é utilizado para um cálculo feito previamente, o ganho na rapidez do algoritmo de Rörich não é relevante, enquanto que no nosso algoritmo

a cadeia w tem seu tamanho minimizado. Quanto ao tempo de geração da cadeia é difícil estabelecer uma comparação, já que dependerá das produções da gramática. Vejamos agora como G_c será utilizado. Falando de maneira genérica, quando a análise de um texto é interrompida, temos construída uma forma sentencial. O conceito pode ser melhor compreendido na análise descendente. Vejamos assim um exemplo:

ex:

$$G = S \rightarrow aBcB \mid bD$$

$$B \rightarrow abcd$$

$$B \rightarrow dB$$

$$B \rightarrow cD$$

$$D \rightarrow bB$$

$$D \rightarrow c$$

Após a aplicação do algoritmo Custos teremos :

$$C(S)=2, C(B)=2, C(D)=1.$$

Desta maneira :

$$G_c = S \rightarrow bD$$

$$B \rightarrow cD$$

$$D \rightarrow c$$

Seja o trecho 'addcabcdce'. A análise será interrompida quando a árvore de derivação estiver no seguinte estado:

Onde a cadeia já reconhecida é indicada pelos símbolos rodeados duas vezes. Neste ponto temos $\mu = \text{'addc'}$ e $\beta = \text{'abcde'}$.

A forma sentencial que temos neste ponto é :

addcCcB

Para encontrarmos ω basta derivarmos, enquanto existirem, os símbolos da forma sentencial utilizando produções de G_c . Neste exemplo obteremos a sentença 'addcccc', o que nos dá $\omega = \text{'cccc'}$.

Porém existe um problema. Em analisadores do tipo LR a noção de qual forma sentencial está sendo reconhecida não é precisa. A cada estado os vários itens nos dão uma multiplicidade de formas sentenciais que podem estar sendo reconhecidas. Para resolver este problema é necessário escolher uma delas para tentar completá-la utilizando G_c e no processo calcular ω , o conjunto de âncoras e seus respectivos prefixos. Temos então de novo o mesmo problema: como, na estrutura de um analisador LR, escolher uma produção determinada e seguir o seu reconhecimento, se os estados não fazem distinção entre as produções de seus vários itens?

Uma primeira idéia é marcar as produções relativas à gramática G_c e, a cada estado utilizar itens relativos a uma produção marcada. Porém podem ocorrer dois itens marcados em um mesmo estado e uma escolha mal feita do item a ser seguido, ou seja, do símbolo que será utilizado na transição, pode acarretar uma sucessão infinita de iterações no algoritmo de geração de ω . Vejamos um Exemplo

Ex Seja a seguinte gramática:

$$\begin{aligned} S &\rightarrow Z\# \\ Z &\rightarrow rRk \\ R &\rightarrow X \\ R &\rightarrow YR \\ Y &\rightarrow ey \\ X &\rightarrow ex \end{aligned}$$

As produções da gramática de continuação serão

$$\begin{aligned} S &\rightarrow Z\# \\ Z &\rightarrow fRk \\ R &\rightarrow X \\ Y &\rightarrow ey \\ X &\rightarrow ex \end{aligned}$$

Para este analisador teremos, entre outros, os seguintes estados:

$$\begin{aligned}
 V(\lambda) &= \{ \\
 &\quad (S \rightarrow .Z\# ; \{\}) \\
 &\quad (Z \rightarrow .rRk ; \{\#\}) \\
 &\} \\
 V(r) &= \{ \\
 &\quad (R \rightarrow .X ; \{k\}) \\
 &\quad (R \rightarrow .YR ; \{k\}) \\
 &\quad (Y \rightarrow .ey ; \{e\}) \\
 &\quad (X \rightarrow .ex ; \{k\}) \\
 &\} \\
 V(re) &= \{ \\
 &\quad (Y \rightarrow e.y ; \{e\}) \\
 &\quad (X \rightarrow e.x ; \{k\}) \\
 &\} \\
 V(rex) &= \{ (X \rightarrow ex. ; \{k\}) \} \\
 V(rey) &= \{ (Y \rightarrow ey. ; \{k\}) \} \\
 V(rY) &= \{ \\
 &\quad (R \rightarrow Y.R ; \{k\}) \\
 &\quad (R \rightarrow .X ; \{k\}) \\
 &\quad (R \rightarrow .YR ; \{k\}) \\
 &\quad (Y \rightarrow .ey ; \{e\}) \\
 &\quad (X \rightarrow .ex ; \{k\}) \\
 &\} \\
 V(rYe) &= V(re) \\
 V(rYey) &= V(rey)
 \end{aligned}$$

Seja a cadeia 'rk'. A configuração da pilha do analisador no momento da detecção do erro será: $V(\lambda)V(r)$.

No estado $V(r)$ dois itens da gramática de continuação determinam avanço utilizando um símbolo terminal: $(Y \rightarrow .ey; \{e\})$ e $(X \rightarrow .ex; \{k\})$.

Se escolhermos o primeiro para construir ω teremos a seguinte sequência:

$$\begin{aligned}
 \text{Pilha} &= V(\lambda)V(r)V(re) ; \omega = 'e' \\
 \text{Pilha} &= V(\lambda)V(r)V(re)V(rey) ; \omega = 'ey' \\
 \text{Pilha} &= V(\lambda)V(r)V(rY)V(rYe) ; \omega = 'eye' \\
 \text{Pilha} &= V(\lambda)V(r)V(rY)V(rYe)V(rYey) ; \omega = 'eyey' \\
 \text{Pilha} &= V(\lambda)V(r)V(rY)V(rYe) ; \omega = 'eyeye'
 \end{aligned}$$

...

Como podemos notar o processo de geração de w entraria em "loop" e não terminaria nunca. Isto porque deveríamos escolher o item $(X \rightarrow \cdot ex; \{k\})$ para fazer a transição já que a produção de R que deve ser utilizada e a da gramática de continuação e portanto o reconhecimento de Y não nos leva a lugar algum.

Como pudemos ver no exemplo, na estrutura tradicional de um analisador LR é impossível em alguns casos gerar w como desejado. Para que a geração de w sempre seja possível, faremos algumas alterações na gramática G e no algoritmo de construção dos estados do analisador. Na gramática, ordenaremos as produções de maneira que, para cada símbolo não-terminal A , a produção de A que pertence a G_c seja colocada antes das demais. No algoritmo de construção de estados do analisador, os estados passam a ser considerados como conjuntos **ordenados** de itens. Além disso, os itens gerados pelo 'completer' devem ficar logo após o item que lhes deu origem. Desta maneira o "completer" deve ser aplicado recursivamente de maneira que entre um item gerado pelo "completer" e seu item "gerador" existam apenas outros itens gerados pelo mesmo "gerador". Vejamos como fica o analisador do exemplo anterior:

Ex: Para este analisador teremos, que os estados descritos teriam o seguinte aspecto

$$\begin{aligned}
 V(\lambda) &= \{ \\
 &\quad (S \rightarrow \cdot Z\# ; \{\}) \\
 &\quad (Z \rightarrow \cdot rRk ; \{\#\}) \\
 &\quad \} \\
 V(r) &= \{ \\
 &\quad (R \rightarrow \cdot X ; \{k\}) \\
 &\quad (X \rightarrow \cdot ex ; \{k\}) \\
 &\quad (R \rightarrow \cdot YR ; \{k\}) \\
 &\quad (Y \rightarrow \cdot ey ; \{e\}) \\
 &\quad \} \\
 V(re) &= \{ \\
 &\quad (X \rightarrow e.x ; \{k\}) \\
 &\quad (Y \rightarrow e.y ; \{e\}) \\
 &\quad \}
 \end{aligned}$$

$$\begin{aligned}
 V(\text{rex}) &= \{ (X \rightarrow \text{ex.} ; \{k\}) \} \\
 V(\text{rey}) &= \{ (Y \rightarrow \text{ey.} ; \{k\}) \} \\
 V(\text{rY}) &= \{ \\
 &\quad (R \rightarrow Y.R ; \{k\}) \\
 &\quad (R \rightarrow .X ; \{k\}) \\
 &\quad (X \rightarrow .\text{ex} ; \{k\}) \\
 &\quad (R \rightarrow .YR ; \{k\}) \\
 &\quad (Y \rightarrow .\text{ey} ; \{e\}) \\
 &\quad \} \\
 V(\text{rYe}) &= V(\text{re}) \\
 V(\text{rYey}) &= V(\text{rey})
 \end{aligned}$$

Seja a cadeia 'rk'. A configuração da pilha do analisador no momento da detecção do erro será: $V(\lambda)V(r)$.

No estado $V(r)$ dois itens da gramática de continuação determinam avanço utilizando um símbolo terminal: $(Y \rightarrow .\text{ey}; \{e\})$ e $(X \rightarrow .\text{ex}; \{k\})$.

Neste caso escolheríamos o segundo, já que aparece antes no estado termos então:

$$\begin{aligned}
 \text{Pilha} &= V(\lambda)V(r)V(\text{re}) ; \omega = 'e' \\
 \text{Pilha} &= V(\lambda)V(r)V(\text{re})V(\text{rex}); \omega = 'ex' \\
 \text{Pilha} &= V(\lambda)V(r)V(\text{rX}); \omega = 'ex' \\
 \text{Pilha} &= V(\lambda)V(r)V(\text{rR})V(\text{rRk}); \omega = 'exk' \\
 \text{Pilha} &= V(\lambda)V(Z)V(Z\#) ; \omega = 'exk\#'
 \end{aligned}$$

Portanto a análise termina e obtemos $\omega = 'exk\#'$.

Rörich formaliza esta ordenação de estados introduzindo duas relações de ordem :

def3.2.5 Dados dois itens $i_1 i_2$, então $i_1 \ll i_2$ sse

$$\begin{aligned}
 I_1 &= (X \rightarrow \alpha.Y\beta ; u) \text{ e} \\
 I_2 &= (Y \rightarrow .\mu; \{c / \exists k \in u \text{ onde } c \in \text{First}(\beta k)\})
 \end{aligned}$$

isto é , se I_2 é gerado de I_1 através da aplicação do 'completer'
 $\ll^*$ é o fecho transitivo e reflexivo desta relação

def3.2.6 Dados dois itens $I_1 e I_2$, $I_1 *** I_2$ sse

$$I_1 = (X \rightarrow \alpha.Y\beta ; u) \text{ e}$$

$$I_2 = (X \rightarrow \alpha Y.\beta; u)$$

isto é se I_2 é gerado de I_1 através da aplicação do 'scanner'.

def3.2.7 O ancestral inicial de um item I , denotado por $AI(I)$, em um estado q , é um item I' t.q. $I' \ll^* I$ e $\forall I''$ t.q. $I'' \ll^* I$ temos que $I' \ll^* I''$.

Como já dissemos, cada estado q será um conjunto ordenado de itens $\{I_1, \dots, I_n\}$ t.q., dados $I_i, I_j \in q$ então $i < j$ sse:

$$a) AI(I_i) = AI(I_j) \text{ e } I_i \ll^* I_j$$

ou

b) $AI(I_i) = AI(I_j)$ e $\exists k < i, j$ t.q. $I_k \ll I_i$, $I_k \ll I_j$ e a produção de I_i vem antes da produção de I_j na gramática.

ou

c) $AI(I_i) = AI(I_j)$ e $\exists k < l < m$ t.q. $I_k \ll I_l$, $I_k \ll I_m$, $I_l \ll^* I_i$, $I_m \ll^* I_j$

ou

d) $AI(I_i) \neq AI(I_j)$ e existe um estado q do analisador, $q = J_1, \dots, J_m$, t.q. $J_n \ll^* I_i$, $J_p \ll^* I_j$, e $0 < n < p < m$.

Para obtermos esta numeração basta, como já dissemos, na geração de um estado, sempre aplicar o "completer" a um item logo após a inclusão deste num estado, incluindo em seguida os itens por ele gerados.

Esta ordenação garante que, sempre que um estado possuir itens relativos a produções de G_c , um destes itens estará em primeiro lugar. Mais que isso, a disposição dos itens garante que $\text{Cont}(q)$ pode ser corretamente calculado, bastando para isto utilizar um valor aceito pelo primeiro item do estado em questão.

Desta maneira, na fase de construção de ω , escolho sempre a produção do primeiro item do estado do topo da pilha determinando um avanço para completar inicialmente. Pela ordem de construção, se for determinado um avanço, o item do estado seguinte resultante da aplicação do "scanner" no item atual será também o primeiro item e assim por diante. Para a descrição do algoritmo utilizaremos duas funções auxiliares, Item_inicial , que, dado um estado, devolve o primeiro item deste estado; e Succ que, dado um item, devolve o próximo item do estado, se houver, e devolve "?" caso contrário. Vejamos o algoritmo:

Algoritmo Calcula Cont

entrada: Q - conjunto de estados do analisador, construídos conforme acima

saída: Cont- vetor com uma entrada para cada estado contendo o valor de $\text{Cont}(q)$

com1

para cada $q \in Q$ faça

com2

$I := \text{Item\_inicial}(q);$

(*procura primeiro item que não tenha avanço em símbolo não terminal*)

enquanto $\text{Prim}(\text{N\_Rec}(\text{Item}(I))) \in V_n$

faça $I := \text{Succ}(I);$ (*vai p/ o prox item*)

se $\text{N\_Rec}(\text{Item}(q,1)) \neq \lambda$

(*se o ponto do item não está no fim do lado direito*)

então $\text{Cont}(q) := \text{Prim}(\text{N\_Rec}(\text{Item}(q,1)))$

senão $\text{Cont}(q) := a$, onde $a \in \text{Look\_Ahead}(\text{Item}(q,i));$

fim2

fim1

Para encerrar, é preciso observar que, embora o conceito de custo de correção não apareça explícita neste algoritmo, ele foi classificado entre os algoritmos de custo mínimo por dois motivos: Em primeiro lugar a escolha de β' e ω' é orientada no sentido de tornar β' o menor possível, estando portanto embutida a idéia de custo mínimo de edição. Em segundo lugar a escolha de ω também envolve a idéia de mínimo. No algoritmo original queremos um número mínimo de passos de derivação; na modificação por nós proposta ω tem tamanho mínimo. S

Capítulo 4

Recuperação de Erros Regional

Os métodos locais de recuperação de erros levam em consideração apenas um símbolo da cadeia de entrada e um estado da pilha de cada vez. Além disso, partem todos do princípio de que um erro é sempre detectado no local da cadeia de entrada onde deveria ocorrer a correção. Desta maneira alguns erros importantes não são recuperados de maneira satisfatória, comprometendo a análise futura e gerando erros expúrios.

Como já foi dito, nos métodos regionais de recuperação de erros tenta-se resolver os problemas acima mencionados estipulando-se uma região ao redor do PDE (ponto de detecção de erro) no qual serão tentadas várias correções possíveis. Chamaremos esta região de contexto de correção. A decisão sobre qual das possíveis recuperações será levada a cabo varia de método para método, mas o objetivo é sempre modificar o menos possível o texto original, proporcionando uma recuperação sem erros expúrios.

Cabe observar que os métodos de recuperação regional não são necessariamente uma evolução dos métodos locais de correção de erro. Sua origem provém antes de uma outra escala de valores na avaliação da relação custo/benefício de uma correção, isto é, na decisão de que prioridade se deve dar à qualidade de recuperação e que prioridade dar à rapidez desta. Alguns conceitos desenvolvidos para métodos regionais foram inclusive utilizados posteriormente em algoritmos locais, como é o caso de custos diferenciados para operações de edição com símbolos distintos, utilizados por Fischer, Dyon e Mauney (ver cap. anterior) mas desenvolvidos inicialmente por Graham e Rhodes [10].

Podemos subdividir métodos de recuperação regionais em três categorias distintas, embora as duas primeiras contenham apenas um método cada uma. A existência de duas categorias com apenas um método se justifica, na nossa opinião, pelo grau de diferença entre os três grupos. A primeira destas categorias é a da recuperação por padrões. A recuperação aqui é feita através da escolha de um padrão de substituição de símbolos pertencente a uma lista pré definida e a partir dele gerando uma recuperação de custo mínimo.

A segunda categoria é a de recuperação por redução de cadeias, que consiste em escolher um trecho ao redor do PDE e substituí-lo por um não terminal adequado.

Finalmente temos a terceira e mais numerosa categoria. Nela estão incluídos os métodos que utilizando o método de análise sintática, realizam um processo de "backtracking" na pilha à procura do ponto real de ocorrência do erro sintático. Nesta categoria alguns métodos elaboram uma estrutura de custos ([10], [35]) para atribuir valores às várias correções possíveis, escolhendo no fim aquela mais "barata". Outros [26] procuram a correção que envolve o menor número possível de operações de edição e alguns simplesmente adotam a primeira solução encontrada [19] [22]. Todos os métodos porém estabelecem critérios mínimos de aceitabilidade de uma recuperação.

Nos algoritmos que seguem utilizaremos, eventualmente, dois procedimentos auxiliares: *Análise_limitada*((α, β), *Nsimb*, *Erro*, *Avanço*, (α', β')) e *Menor_custo*(*Lista*). O primeiro executa uma análise sintática limitada a *Nsimb* símbolos de entrada, partindo da configuração (α, β), devolvendo a configuração resultante em (α', β'), indicando em *Avanço* o número de símbolos analisados e em *Erro* a ocorrência de um eventual erro sintático.

No caso de ocorrência de erro, de fim de análise, ou na eminência de efetuar uma redução maior que a pilha, o algoritmo termina sua execução, indicando erro somente no segundo caso. Este algoritmo se encontra codificado abaixo. O segundo algoritmo, uma função, executa uma busca em Lista, uma lista onde cada elemento tem três campos: uma configuração de pilha de analisador, uma cadeia de terminais e um valor numérico. A função retorna o elemento da lista com menor valor do campo numérico.

Algoritmo Análise limitada

entrada: (α, β) - configuração do analisador

Nsymb - avanço máximo

saída: Erro - variável booleana indicando se a análise acabou em
ou não

Avanço - número de símbolos consumidos

(γ, β') - configuração do analisador após análise

com1

Avanço:=0;

$\gamma := \alpha$;

$\beta' := \beta$;

Erro:= falso;

Continua:=verdade;

enquanto Continua e não Erro faça

caso Atitude(Ult(γ),Prim(β')) seja

avanço:se Avanço= Nsymb

então Continua:=falso

senão)

co m2

Avanço:=Avanço+1;

$\gamma := \gamma$ cat Destino(Ult(γ),Prim(β'));

$\beta' := \text{Ult}(\beta')$

fim2(*consumiu um símbolo de entrada*)

reduz($A \rightarrow \delta$):

com2

se $|\delta| > |\gamma|$

(*se lado direito maior que pilha*)

então Continua:=falso(*para*)

```

senão
    com3
        para Aux de 1 até / $\delta$ /
            faça  $\gamma := \text{Início}(\gamma)$ ;
             $\gamma := \gamma$  cat Destino(Ult( $\gamma$ ), A)
        fim3(*completa a redução*)
    fim2
    aceita: Continua := falso
    erro: Erro := verdade
fimcaso
fim1.

```

4.1 A Recuperação por Padrões de Troca

def 4.1.1 Um padrão $P_{m,n}$ é uma sequência de números $p(1)p(2)\dots p(n)$ tal que:

- a) $0 \leq p(i) \leq m$ para $1 \leq i \leq n$
- b) $\exists j$ t.q. $p(j) = m$
- c) se $i \neq j$ e $p(i) \neq 0$ então $p(i) \neq p(j)$

def 4.1.2 Um padrão $P_{m,n}$ mapeia a cadeia $\alpha = a_1 \dots a_m$ na cadeia $\gamma = g_1 \dots g_n$ sse:

$p(j) \neq 0$ implica $g_j = a_{p(j)}$, $1 \leq j \leq n$

Ex

- 1) $P_{4,3} = 3\ 0\ 4$ mapeia $\alpha = \text{'dapc'}$ em $\gamma = \text{'pec'}$, pois:
 $p(1) = 3$ e $g_1 = a_3$ e
 $p(3) = 4$ e $g_3 = a_4$.
- 2) $P_{4,3} = 0\ 0\ 4$ também mapeia $\alpha = \text{'dapc'}$ em $\gamma = \text{'pec'}$ pois
 $p(3) = 4$ e $g_3 = a_4$
- 3) $P_{4,3} = 0\ 0\ 4$ mapeia $\alpha = \text{'dapc'}$ em $\gamma' = \text{'aac'}$ pois
 $p(3) = 4$ e $g'_3 = a_4$.

def 4.1.3 Para qualquer padrão P e qualquer cadeia x ,

$P(x) = \{y \mid P \text{ mapeia } x \text{ em } y\}$.

O método de correção por padrões foi proposto por Tai em 1978 [24], e funciona da seguinte maneira: inicialmente procura-se, numa lista pré definida de padrões, qual o primeiro deles que mapeia uma região ao redor do PDE num trecho sintaticamente correto. Este padrão será utilizado para encontrar uma recuperação para o erro em questão. Como um mesmo padrão pode mapear uma determinada cadeia em várias outras (ver exemplo anterior), uma escolha entre várias possíveis soluções pode se tornar necessária. Para resolver este problema são atribuídos custos diferenciados de inserção, eliminação para cada símbolo e custos diferenciados para troca simples e troca de símbolos em posições adjacentes para cada par de símbolos. A alternativa escolhida ao final é a que possui um custo associado menor.

def 4.1.4 Seja (α, β) uma configuração de erro de um analisador sintático de uma GLC $G = (V_n, V_t, P, S)$ e seja μ a porção da cadeia de entrada já analisada. Um padrão $P_{m,n}$ é dito com mapeamento bem sucedido se $\exists \gamma \in V_t^*, |\gamma| = n$, t.q.:

$$\gamma \in P_{m,n}(\text{First}_m(\beta)) \text{ e } \mu\gamma \in \text{Prefix}(L(G))$$

Dado um padrão $P_{m,n}$ e uma configuração de erro (α, β) de um analisador, para descobirmos se $P_{m,n}$ tem mapeamento bem sucedido aplicamos o seguinte algoritmo:

Algoritmo Bem_Sucedido

entrada: (α, β) - estado do analisador

P - padrão a ser testado

saída: Sucesso- variável booleana com o resultado do teste

com1

Sucesso:=falso;

$P' := P$;

$\alpha' := \alpha$;

se $P' = \lambda$ (*se temos o padrão vazio*)

então Sucesso:=verdade

senão

se $\text{Prim}(P') = 0$ (*se tenho o número 0 no padrão*)

então

```

com2
  para cada  $k \in V_t$  faça (*testo todos os símbolos*)
    se Aceita(Ult( $\alpha'$ ),  $k$ ) (*se símbolo é aceitável*)
      então
        com3
           $\alpha'' := \alpha$ ;
          Análise_limitada( $\alpha''$ ,  $k$  cat  $\beta$ , 1, Auxbool,
            Avanço, ( $\alpha''$ ,  $\beta$ ));
          (*simula reconhecimento do símbolo*)
          Bem_Sucedido(( $\alpha''$ ,  $\beta$ ), Final( $P'$ ), Sucesso);
          (*verifica o resto do padrão*)
          (*2*)
          se Sucesso (*se símbolo propicia recuperação*)
            então encerre algoritmo
          fim3
        fim2
      senão (*numero diferente de zero no padrão*)
        se Aceita(Ult( $\alpha'$ ,  $b_{Prim(P')}$ ))
          (*se símbolo é aceitável*)
          então
            com2
               $\alpha'' := \alpha$ ;
              Análise_limitada( $\alpha''$ ,  $k$  cat  $\beta$ , 1, Auxbool,
                Avanço, ( $\alpha''$ ,  $\beta$ ));
              (*simula reconhecimento do símbolo*)
              Bem_Sucedido(( $\alpha''$ ,  $\beta$ ), Final( $P'$ ),
                 $\gamma$  cat  $\beta_{Prim(P')}$ ,
                Sucesso);
              (*verifica o resto do padrão*)
            fim2
          fim1.

```

Se na lista de padrões não for encontrado nenhum com mapeamento bem sucedido é sugerido pelo autor um algoritmo alternativo de recuperação, que trataremos mais adiante.

A escolha dos padrões e sua ordenação é, como podemos notar, crucial no desempenho do método de recuperação. Infelizmente não é possível

utilizar todos os padrões possíveis, pois estes são em quantidade infinita. Desta maneira, o primeiro problema que se coloca é como limitar o conjunto de padrões.

def 4.1.5 $P_{m,n}$ é um padrão com k símbolos de lookahead corretos sse $p(n)=m, p(n-1)=m-1, \dots, p(n-k+1)=m-k+1 \neq 0$. Dizemos neste caso que $P_{m,n}$ é $CL(k)$.

É fácil ver que se $P_{m,n}$ mapeia γ em δ então γ e δ têm o mesmo sufixo de tamanho k .

A utilização de um padrão $CL(k)$ em uma recuperação vai garantir a análise bem sucedida de pelo menos mais k símbolos da cadeia de entrada. Com isto visa-se evitar, com a escolha de um k apropriado, a "propagação" de erros, isto é, a geração de erros expúrios. Quanto maior o k escolhido, mais garantias de uma boa recuperação teremos.

Para ordenarmos os padrões designaremos um custo a cada um deles. Como é desejável que a cadeia de entrada seja alterada o menos possível, é criado um custo que aumenta conforme aumenta o número de modificações indicadas pelo padrão. Os padrões serão ordenados por custo crescente.

def 4.1.6 Se $P_{m,n}$ é um padrão, definimos $\text{custo}(P_{m,n})$ como a distância de Hamming entre $P_{m,n}$ e a cadeia $M=1\ 2\ 3\ 4\ \dots\ m$.

Teorema 4.1

Sejam δ, γ cadeias com o mesmo sufixo de tamanho k . Então existe um padrão $P_{m,n} \in CL(k)$ tal que $\text{custo}(P_{m,n})$ é igual à distância de Hamming entre x e y .

O teorema garante que uma lista dos padrões $CL(k)$, se percorrida em ordem crescente de custo cobre todas as possibilidades de recuperação, possibilitando que se encontre uma com distância de Hamming mínima. Infelizmente o nosso problema ainda não está resolvido pois existe um número infinito de padrões $CL(k)$ para um k fixo, mesmo se fixarmos o custo dos padrões.

Ex O seguinte conjunto de padrões CL(1) tem só padrões de custo 1: $\{0\ 1, 0\ 1\ 2, 0\ 1\ 2\ 3, 0\ 1\ 2\ 3\ 4, \dots\}$

def 4.7 $P_{m,n}$ é um padrão RCL(k), isto é um padrão restrito com k símbolos de lookahead sse:

- a) $P_{m,n}$ é CL(k)
- b) $P_{m,n}(n-k) \neq m-k$
- c) não existe γ prefixo próprio de $P_{m,n}$ tal que
  - c.1) $\gamma = \dots r-k+1\ r-k+2 \dots r$
  - c.2) $\text{custo}(\gamma) < \text{custo}(P_{m,n})$

Teorema 4.2

Seja $P_{m,n}$ um padrão RCL(k) com custo C. Então o comprimento do $P_{m,n}$ é, no máximo: $k^*(1+c)+2^*c$

Com os padrões RCL(k) é possível construir uma lista finita de padrões cobrindo todas as possibilidades, uma vez fixados k e o custo dos padrões. Os padrões RCL(k) representam apenas as transformações com lookahead de k símbolos corretos que não possuem outras como prefixo. Em outras palavras, os padrões RCL(k) representam transformações de cadeias com k símbolos de lookahead que têm tamanho mínimo.

Teorema 4.3

Sejam δ e γ cadeias com o mesmo sufixo de comprimento k, mas que não tenham os sufixos de comprimento k+1 iguais. Suponhamos também que, para qualquer prefixo u de δ e v de γ , u e v não têm o mesmo sufixo de comprimento k se $d_h(u,v) < d_h(\delta,\gamma)$. Então existe um padrão RCL(k), $P_{m,n}$, que mapeia δ em γ t.q. $\text{custo}(P_{m,n}) = d_h(\delta,\gamma)$.

Este teorema garante que os padrões RCL(k) para um dado custo C são suficientes para encontrar um padrão com mapeamento bem sucedido em qualquer recuperação que envolva no máximo C operações de edição, se ela existir.

Teorema 4.4

Seja R uma lista de todos os padrões $RCL(k)$ com custo menor ou igual a C , ordenados por valores crescentes de custo. Então uma recuperação obtida utilizando o primeiro padrão de R com mapeamento bem sucedido é uma recuperação com k símbolos de lookahead corretos de distância de Hamming mínima em relação à configuração original da cadeia de entrada, onde esta distância é menor ou igual a C .

Este teorema garante finalmente que, se estabelecermos um limite no número de operações de edição, e se fixarmos a quantidade de símbolos corretos que queremos após o trecho modificado, podemos construir uma lista finita de padrões que nos fornecerão uma recuperação, se ela existir.

Temos agora o problema de como, dado um custo C e um número inteiro k , construir a lista de padrões $RCL(k)$ com custo menor ou igual a C . Sabemos que, para um dado padrão $P_{m,n}$ desta lista, n é limitado por k e C (ver teorema 4.2). Veremos agora que o mesmo acontece com m :

a) se $m < n$ então m é trivialmente limitado por k e C , já que n também é.

b) se $m > n$ então pelo menos $m-n$ eliminações serão necessárias para transformar $M = 1\ 2\ 3\ \dots\ m$ em $P_{m,n}$ e portanto $C \geq (m-n)$ de onde tiramos que $m \leq C+n$.

Para gerarmos a lista de padrões $RCL(k)$ de custo C basta gerarmos todos os padrões de tamanho até $(k*(1+C)+2*C)$ envolvendo números de 1 até $(k*(1+C)+3*C)$ e em seguida calcular seus custos, ordenando-os e descartando aqueles com custo maior que C . Um algoritmo para o cálculo do custo de um padrão pode ser encontrado em [28]

Uma vez resolvido o problema da escolha de um padrão, é necessário construir a cadeia que substituirá um trecho da cadeia de entrada, efetivando a recuperação. Se o padrão escolhido não tiver nenhum 0 então a escolha é única. Porém um padrão com o número zero em seu bojo possibilita várias escolhas. Nestes casos pode ocorrer que existam várias recuperações possíveis. Para realizarmos esta escolha quando ela se fizer necessária designamos custos às operações de edição a exemplo do que foi feito por Fischer, Dyon e Mauney (ver métodos de recuperação locais). Utilizando estes custos associamos a cada alternativa um valor, escolhendo aquela mais "barata".

Não podemos esquecer igualmente que mesmo depois de encontrar

um padrão com mapeamento bem sucedido, pode eventualmente ser possível encontrar outro padrão com mapeamento bem sucedido de mesmo custo que propicie uma recuperação mais "barata".

Para implementar a solução aos problemas acima levantados, é necessário modificar o algoritmo Bem_Sucedido:

Algoritmo Bem_Sucedido_II

entrada: (α, β) - configuração do analisador

P - padrão a ser testado

γ - cadeia de recuperação parcial ou totalmente construída

saída: Lista - lista de possíveis cadeias de recuperação

Sucesso-variável booleana indicando se existe um padrão com mapeamento bem sucedido.

com1

$\alpha' := \alpha;$

$P' := P;$

se $P' = \lambda$ (*se temos o padrão vazio*)

então

com2

Lista := Lista $\cup$ $\{\gamma\}$

Sucesso := verdade

fim2

senão

se $\text{Prim}(P') = 0$ (*se tenho o número 0 no padrão*)

então

para cada $k \in V_t$ faça (*testo todos os símbolos*)

se Aceita($\text{Ult}(\alpha'), k$) (*se símbolo é aceitável*)

então

com2

$\alpha'' := \alpha;$

Análise_limitada(α'', k cat $\beta, 1, \text{Auxboc}^1,$

Avanço, (α'', β));

(*simula reconhecimento do símbolo*)

Bem_Sucedido_II($(\alpha'', \beta), \text{Final}(P'),$

γ cat $\beta_{\text{Prim}(P')},$

Lista, Sucesso);


```

                                (*verifica o resto do padrão*)
                                fim2
senão (*numero diferente de zero no padrão*)
  se Aceita(Ult( $\alpha'$ ,  $b_{Prim(P')}$ ))
    (*se símbolo é aceitável*)
    então
      com2
         $\alpha'' := \alpha$ ;
        Análise Limitada( $\alpha''$ , k cat  $\beta$ , 1, Auxbool,
                          Avanço, ( $\alpha''$ ,  $\beta$ ));
        (*simula reconhecimento do símbolo*)
        Bem_Sucedido(( $\alpha''$ ,  $\beta$ ), Final( $P'$ ),
                      $\gamma$  cat  $\beta_{Prim(P')}$ ,
                     Lista, Sucesso);
        (*verifica o resto do padrão*)
      fim2
    fim1.

```

Desta maneira se, após a execução do algoritmo Bem_Sucedido_II existir algum padrão da com mapeamento bem sucedido, 'Lista' conterá todas as cadeias de recuperação para os padrões bem sucedidos. Deve ser acionado então um algoritmo que calcula o custo de cada uma das possíveis cadeias de recuperação, encontrando aquela de custo mínimo. Este algoritmo pode ser encontrado em [56].

As demonstrações de todos os teoremas utilizados para este método podem ser encontradas em [49]. Os teoremas foram todos enunciados originalmente supondo custos diferenciados às operações genéricas de inserção troca eliminação e intercâmbio de símbolos adjacentes em padrões. Em nosso texto consideramos todos com o mesmo valor para maior clareza.

Três extensões podem ser feitas para melhorar a performance do método de recuperação de contexto global, extensões estas sugeridas pelo próprio autor: A primeira delas consiste na inclusão dos símbolos não terminais entre aqueles que podem ser usados na recuperação, aumentando as possibilidades de correção. A segunda extensão sugerida por Tai consiste em considerar, para fins de recuperação, também a parte já analisada do texto. Se a segunda extensão não for feita juntamente com a primeira, torna-se necessário adicionar ao analisador uma estrutura de dados para armazenar um sufixo da porção da cadeia de entrada já analisada. Em

ambos os casos é necessário observar que será imperativo reanalisar todo o trecho recuperado. Uma última extensão sugerida pelo autor: podemos, concomitantemente com a primeira delas, implementar uma “fase de condensação” no analisador (ver item 4.3), artifício que nos permite minimizar a limitação na garantia de um certo número de símbolos corretos após a recuperação.

Finalmente, é importante lembrar que este algoritmo nem sempre é bem sucedido: se houver um número muito grande de erros em algum trecho da cadeia de entrada, isto é, com mais erros do que o custo do padrão mais “caro”, nenhum dos padrões da lista tem mapeamento bem sucedido. Neste caso deve ser acionado um algoritmo alternativo a fim de garantir a recuperação do erro.

def 4.1.7 $\forall b \in V_t,$

$$\text{Prec}(b) = \{A \in V_n / \exists \delta \in \text{FS}(G) \text{ tq } \delta = \delta' A b \delta'\}$$

($\text{Prec}(b)$ é o conjunto dos símbolos não terminais que podem preceder ao terminal ‘b’ na análise de uma sentença da linguagem).

O algoritmo alternativo proposto por Tai tem funcionamento semelhante ao do panic mode de Wirth: inicialmente são consumidos símbolos de entrada até que seja encontrado um símbolo ‘a’ pertencente a um conjunto pré determinado, que denominaremos conjunto de recuperação. Em seguida verifica-se se algum símbolo de $\text{Prec}(a)$ é aceito por algum estado da pilha. Em caso afirmativo a parte da pilha acima deste estado é eliminada e o símbolo de $\text{Prec}(b)$ aceito pelo estado que agora está no topo da pilha do analisador é colocado no início da cadeia de entrada. Caso nenhum símbolo de $\text{Prec}(a)$ seja aceito por nenhum estado da pilha todo o processo é repetido.

O conjunto de recuperação é formado pelos símbolos terminais cujo conjunto $\text{Prec}()$ tem um número de elementos menor do que um certo limite estabelecido a critério do implementador. A idéia é que o conjunto de recuperação contenha apenas símbolos que podem suceder a um número “pequeno” de não terminais e que por isto são bons “ganchos” de recuperação. Vejamos o algoritmo:

Algoritmo Alternativo de Tai

entrada: (α, β) - configuração do analisador
           Rec - conjunto de símbolos de recuperação
           Prec - a função Prec( ).
 saída: (α, β) - nova configuração do analisador
com1
     Sucesso := falso;
     enquanto não Sucesso faça
       com2
          $\alpha' := \alpha$;
         enquanto Prim(β) $\notin$ Rec faça
            $\beta := \text{Final}(\beta)$; (*procura símbolo de recuperação*)
            $b := \text{Prim}(\beta)$; (*b recebe símbolo de recuperação*)
           enquanto $\alpha' \neq \lambda$ e não Sucesso faça
             com3
               para cada $A \in \text{Prec}(b)$ faça
                 (*verifica se alguém de Prec é aceito *)
                 se Aceita(Ult(α), A)
                   então com4
                      $\beta := A \text{ cat } \beta$;
                     Sucesso := verdade;
                     abandone loop
                     (*sai do loop + interno*)
                   fim4
                 se não Sucesso
                   então $\alpha' := \text{Início}(\alpha')$;
                   (*tenta outro estado da pilha*)
               fim3
             fim2
              $\alpha := \alpha'$; (*reconfigura a pilha*)
       fim1.

Este algoritmo garante o sucesso da recuperação já que, na pior das hipóteses, terminaremos com $\alpha = e_0$ e $\beta = Z\#$, uma vez que o símbolo de fim de cadeia '#' sempre pertence a Rec ($|\text{Prec}(\#)| = |\{Z\}| = 1$), que e_0 sempre está no início da pilha e que S sempre é aceito por e_0 .

4.2 A Recuperação por Redução de Cadeias

def 4.2.1 Dada uma gramática $G = (V_n, V_t, P, S)$ e seu analisador sintático, seja $C = (q_0 \dots q_n, a_0 \dots a_m)$ uma configuração de erro. Uma cadeia erro é uma dupla $E = (q_{i+1}q_{i+2} \dots q_n, a_0a_1 \dots a_{j-1})$, onde $\exists A \in V_n$ e $\exists q_A, e \in Q$ t.q.:

$$(q_0 \dots q_i, Aa_j \dots a_m) << (q_0 \dots q_i q_A, a_j \dots a_m) << (q_0 \dots q_i q_A e, a_{j+1} \dots a_m) ;$$

chamamos q_i de estado de recuperação, A de objetivo de redução, ' e ' de estado objetivo e a_j de símbolo de recuperação.

O método de recuperação por redução de cadeias foi desenvolvido por Sippu e Soisalon-Soininen [25]. Seu princípio de funcionamento é encontrar, para uma dada configuração de erro, uma cadeia erro e removê-la inserindo o estado objetivo no topo da pilha do analisador e o objetivo de redução no topo da pilha sintática. Desta maneira a recuperação por redução de cadeias é equivalente a executar a redução pela produção:

$$A \rightarrow \text{sim}(q_{i+1}) \dots \text{sim}(q_n) a_0 \dots a_{j-1}$$

O analisador, após a execução do algoritmo terá a seguinte configuração:

$$(q_0 \dots q_i q_A, a_j \dots a_m)$$

Podemos notar que, a exemplo do método alternativo de Tai descrito no item anterior, o funcionamento deste método é semelhante na forma ao panic mode de Wirth. Porém particularidades na metodologia para encontrar a cadeia erro e o objetivo de redução colocam este método na categoria dos métodos regionais, particularidades estas que veremos a seguir.

Para encontrarmos a cadeia erro vamos testando todas as possíveis configurações que ela pode assumir para uma determinada configuração de erro até encontramos uma adequada. Inicialmente testamos a dupla (λ, λ) , se ela não for uma cadeia erro vamos sucessivamente calculando outras configurações até encontramos uma que seja uma cadeia erro. A estratégia de construção das configurações da dupla candidata a cadeia erro é pré determinada. A última configuração a ser testada é $(q_1 \dots q_n, a_0 \dots a_m)$ sendo que uma configuração não deve ser testada antes de configurações onde ambos os elementos da dupla são sub-cadeias de seus elementos. O objetivo é minimizar o número de símbolos e elementos da pilha presentes na cadeia erro. A proporção de inserção entre elementos da pilha e símbolos da cadeia de

entrada é denominado razão de eliminação e fica a critério do implementador. O autor sugere que dois estados da pilha sejam inseridos antes de tentar a inserção de cada símbolo da cadeia de entrada, estratégia que nos fornece a seguinte sequência de configurações de possíveis cadeias erro:

$$\begin{aligned} &(\lambda, \lambda) \\ &(q_m, \lambda) \\ &(q_{m-1}q_m, \lambda) \quad (\lambda, a_0) \\ &(q_{m-2}q_{m-1}q_m, \lambda) \quad (q_m, a_0) \\ &(q_{m-3}\dots q_m, \lambda) \quad (q_{m-1}q_m, a_0) \quad (\lambda, a_0a_1) \\ &(q_{m-4}\dots q_m, \lambda) \quad (q_{m-2}q_{m-1}q_m, a_0) \quad (q_m, a_0a_1) \\ &\dots \end{aligned}$$

Mas uma mesma cadeia erro pode ter mais de um objetivo de redução associado. Torna-se necessário assim definir critérios para a sua escolha, para que esta não seja arbitrária.

Ex Seja o seguinte trecho de programa

```
prog A;
var A,B,C,D : integer;
begin
...
end.
```

O tipo da declaração está com erro de digitação. A cadeia erro será:

(‘prog’id’;’var’<idlist>’,’ , ‘id’;’ ... ‘end’;’)

neste caso para a cadeia erro (‘:’ , ‘id’) temos dois objetivos de redução: <arraytype> e <sitype>.

Se partirmos do princípio de que o conteúdo da pilha sintática sempre forma um prefixo de uma sentença da gramática analisada, isto é, se considerarmos a propriedade do prefixo correto, é desejável que o objetivo de redução derive uma cadeia que tem a porção eliminada da pilha sintática como prefixo.

def 4.2.2 Um objetivo de redução A de uma cadeia erro $(q_{i+1}\dots q_m, a_0\dots a_j)$ é plausível se $\exists \gamma \in V_t^* t.q.$

$$A \xRightarrow{*} \text{sim}(q_{i+1}) \dots \text{sim}(q_n) \gamma$$

Com um objetivo de redução plausível garantimos que a porção já analisada da cadeia de entrada é considerada na escolha da solução.

Utilizando a definição acima podemos então procurar cadeias-erro que possuam objetivos de redução plausíveis e, para evitar escolhas arbitrárias, escolher para a recuperação uma cadeia erro com apenas um objetivo de redução plausível. Infelizmente a verificação da plausibilidade de um objetivo de redução pode se tornar onerosa em termos de processamento se a porção da pilha utilizada na cadeia erro for grande. Os autores afirmam que geralmente a plausibilidade pode ser decidida observando-se apenas o estado mais à esquerda da cadeia erro. Desta maneira é sugerido um relaxamento da plausibilidade:

def 4.2.3 Um objetivo de redução A de uma cadeia erro $(q_{i+1} \dots q_m, a_0 \dots a_j)$ é fracamente plausível se $\exists \delta \in V_t^+ \text{ t.q.}$

$$A \xRightarrow{*} \text{sim}(q_{i+1}) \delta$$

def 4.2.4 Um objetivo de redução importante A de uma cadeia erro (γ, δ) de uma configuração erro $(\alpha \gamma, \delta \beta)$ se (γ, δ) não possui nenhum objetivo de redução B t.q. $B \xRightarrow{+} A$. Considerando apenas os objetivos de redução importantes para a recuperação, eliminamos uma série de objetivos de redução que são equivalentes em termos da recuperação.

O método de recuperação por redução de cadeias procura a menor cadeia erro com um objetivo de redução importante fracamente plausível único. A recuperação é efetuada retirando-se a cadeia erro da configuração do autômato e inserindo o objetivo de redução citado à frente do restante da cadeia de entrada.

A escolha de objetivos de redução fracamente plausíveis em substituição aos plausíveis simplifica a recuperação já que para cada não terminal A que é objetivo de redução, para sabermos se ele é fracamente plausível basta verificar se o símbolo de entrada do estado mais à esquerda da cadeia erro pertence ao conjunto $First'(A)$ onde $First'(A)$ é uma modificação do conjunto $First(A)$ que inclui símbolos não terminais:

$$\text{def 4.2.5 } \forall A \in V_n \text{ First}'(A) = \{x \in V \mid A \xRightarrow{*} x\gamma, \gamma \in V^+\}$$

$$\forall A \in V_n \text{ First}'_k(A) = \{y \in V^* / |y| = k \text{ e } A \xRightarrow{*} y\gamma, \gamma \in V^*\}$$

o cálculo de $\text{First}'(A)$ pode ser feito previamente, podendo ser utilizado o algoritmo de cálculo de $\text{First}(A)$, presente na geração de analisadores LR e LALR. Caso adotássemos como critério de escolha de cadeias erro a existência de um único objetivo de redução plausível seria necessário, para cada comprimento t da parte da pilha envolvida na cadeia erro, calcular o conjunto $\text{First}'_t(A)$, o que só poderia ser feito no momento de execução do algoritmo de recuperação já que t não é fixo nem limitado. Isto tornaria inviável a utilização prática do método.

Algoritmo Recup_Red_Cadeias

entrada: (α, β) - configuração de erro

First' - vetor indexado por símbolos não terminais, contendo o conjunto $\text{First}'(A)$ para cada $A \in V_n$

Raz_el- razão de eliminação

saída: (α, β) - configuração do analisador após a recuperação

com1

$\alpha' := \alpha;$

$\beta' := \beta;$

Sucesso := falso;

$q := \text{Ult}(\alpha');$ (*q é o estado do topo da pilha *)

$\text{Quant}_p := 0;$ (*número de símbolos da pilha na cadeia erro*)

$\text{Quant}_e := 0;$ (*número de símbolos da cadeia de entrada na cadeia erro*)

enquanto não Sucesso faça

com2

$\text{Imp} := \emptyset;$ (*inicializa conjunto de símbolos importantes*)

para cada $I \in \text{Base}(q)$ faça (*para cada item base*)

com3

$\text{Simb} := \text{Prim}(\text{Não\_Rec}(I));$

se $\text{Simb} \in V_n$ e $\text{Atitude}(\text{Prox}(q, \text{Simb}), \text{Prim}(\beta')) \neq \text{'erro'}$

então $\text{Imp} := \text{Imp} \cup \{\text{Simb}\}$

fim3 (*calculou o conjunto de símbolos importantes*)

$\text{Fp} := \emptyset;$ (*inicializa o conjunto de símbolos fracamente plausíveis*)

se $\text{Imp} \neq \emptyset$ (*se existe algum símbolo importante*)

então se $\text{Quant}_p = 0$

```

    (*se cadeia erro não possui elementos da pilha*)
    então Fp:=Imp
    (*todos os sím. importantes são fracamente
    plausíveis*)
    senão
        para cada Simb∈Imp faça
            se sim(q)∈First'(Simb)
                então Fp:=Fp∪{Simb};
                (*calculou conj dos símbolos
                facamente plausíveis*)
se /Fp/=1 (*se temos só um símbolo fracamente plausível*)
    então
        com3(*efetuamos a recuperação*)
            α:=α';
            β:=β';
            Sucesso:=verdade;
        fim3
    senão
        com3 (*calcula novos tamanhos de cada parte
        da nova cadeia erro*)
            se Quant_p - Raz_el ≥ 0
                então
                    com4
                        Quant_p:=Quant_p - Raz_el;
                        Quant_e:=Quant_e + 1;
                    fim4
                senão
                    com4
                        Quant_p:=(Quant_e*Raz_el) + 1;
                        Quant_e:=0;
                    enquanto Quant_p>/α/ faça
                        (*enquanto Quant_p for maior que
                        o num de símbolos da pilha*)
                    com5
                        Quant_p:=Quant_p - Raz_el;
                        Quant_e:=Quant_e + 1
                    fim5

```



```

                                fim4;
                                (*calculamos agora nova configuração da pilha*)
                                 $\alpha' := \alpha$ ;
                                para l de 1 até (Quant_p - 1) faça
                                     $\alpha' := \text{Início}(\alpha')$ ;
                                    q := Ult( $\alpha'$ );
                                     $\alpha' := \text{Início}(\alpha')$ ;
                                (*finalmente calculamos nova configuração
                                da cadeia de entrada*)
                                 $\beta' := \beta$ ;
                                para l de 1 até (Quant_e) faça
                                     $\beta' := \text{Final}(\beta')$ ;
                                fim3
                                fim2
fim.

```

Existe porém um problema com este método. Ele não funciona satisfatoriamente para alguns casos simples como a ausência de um símbolo ou a existência de um símbolo a mais na cadeia de entrada.

ex

No trecho em pascal:

```

program A;
var B,C,D:integer;
begin
    B C*E
end.

```

a configuração de erro seria:

```

('prog'id' <vardecl> 'begin'id', 'id' '*' 'B' 'end' '.')

```

neste caso a cadeia erro utilizada seria:

```

('id','id' '*' 'b'),

```

com o objetivo de redução <stat>.

Para resolver este problema, os autores sugerem realizar a recuperação em duas fases. A nova fase será chamada de "fase local" e seria aplicada somente às cadeias erro (λ, λ) , (q_m, λ) e (λ, a_0) . Na fase local os símbolos terminais e a cadeia vazia são incluídos como candidatos a objetivos de redução. Desta maneira se um terminal for considerado como objetivo de redução para a cadeia (λ, λ) estaremos executando uma ação equivalente à inserção de um símbolo e se um terminal for objetivo de redução para as cadeias (q_m, λ) e (λ, a_0) teremos o equivalente a uma troca. Finalmente, adotar a cadeia vazia como objetivo de redução das cadeias (q_m, λ) e (λ, a_0) corresponderá finalmente à ação de eliminação de um símbolo. Quando a cadeia vazia for adotada como objetivo de recuperação não haverá estado objetivo, o símbolo de recuperação deve ser aceito pelo próprio estado de recuperação.

Na fase local, da mesma maneira que na outra fase, pode existir mais de um estado objetivo possível para a mesma cadeia erro. Porém neste caso a restrição de plausibilidade não tem sentido. Por outro lado, exigir que o objetivo de redução seja único limitará muito a eficiência da fase local. Para resolver este problema, é introduzido um sistema de custos para a recuperação local. Este sistema é semelhante ao descrito por Fischer, Dyon e Mauney, pois envolve custos individuais de inserção e eliminação para cada símbolo. Porém no sistema de custos da fase local do algoritmo em questão, um novo custo é acrescentado, denominado **custo de segurança** ("safety cost"), cujo objetivo é medir a confiabilidade da operação. Sippu e Soisalon-Soininen elaboram uma tabela de custos baseada no comprimento dos símbolos:

Palavras Reservadas

comprimento	1	2	3	4	5	> 5
custo eliminação	2	4	5	6	7	7
custo inserção	1	2	2	2	2	2
custo segurança	3	2	2	2	1	1

Palavras Não Reservadas

comprimento	1	2	3	4	5	> 5
custo eliminação	1	2	2	2	2	3
custo inserção	4	4	4	4	4	4
custo segurança	6	5	5	5	5	5

Desta maneira o custo de uma correção local é a soma de dois componentes: o primeiro é o custo de modificação que mede a extensão da mudança infligida pela recuperação e é calculado com base nos custos de inserção e eliminação, conforme o caso; o segundo componente é o custo de segurança, cuja finalidade é medir a confiabilidade da recuperação.

4.3 A Recuperação Regional com Backtracking

Em 1975 Lévy [26] publica um método de recuperação regional de erros utilizando um recuo do analisador na pilha a fim de determinar um ponto do texto já analisado a partir do qual o erro pode ter ocorrido. Para isto procura-se um símbolo que seja um ponto de convergência, isto é, um símbolo que, sejam quais forem as possíveis interpretações do texto original, considerando-se que este possui até N erros, é interpretado sempre da mesma maneira. Em outras palavras é um símbolo que, mesmo que se execute até N modificações na porção da cadeia de entrada já analisada, será sempre considerado pelo mesmo estado do analisador. A partir deste ponto, constroem-se todas as possíveis interpretações do texto que o segue, considerando-se até N possíveis correções (inserções, eliminações e trocas de símbolos). Esta construção termina quando todas as possíveis interpretações do texto que ainda podem ser continuadas levam a um mesmo estado do analisador. Neste momento uma das interpretações é escolhida como recuperação do trecho analisado. Para efeito da análise futura, a escolha da interpretação que será adotada como recuperação é indiferente, já que todas elas terminam, para um determinado símbolo da cadeia de entrada, no mesmo estado do analisador.

Para a geração das diversas interpretações são construídas N cópias do analisador original, numeradas de 1 a n . A cópia de número i corresponde à análise do texto com i erros.

Ex. Seja analisador M :

Com as cópias teremos M' :

em M temos uma transição de p para q pelo símbolo 'a'. Em M' as transições são:

- p^i a q^i pelo símbolo 'a' (sem correção)
- p^i a p^{i+1} pelo símbolo 'b' (eliminação de 'b')
- p^i a q^{i+1} pelo símbolo 'c' (troca de 'c' por 'a')
- p^i a q^{i+1} incondicional (inserção de 'a')

O índice dos estados se refere a qual cópia está sendo utilizada. Outros tipos de correção são possíveis no exemplo, naturalmente, mas para simplificar o exemplo elas não foram representadas.

Não foi por nós encontrada na literatura nenhuma tentativa de implementação direta deste método. Mesmo Lévy quando descreve os princípios do algoritmo só o faz para reconhecedores de linguagens regulares, isto é, autômatos finitos. Para os analisadores LR, mais gerais e complexos que os autômatos finitos, existem muitas complicações adicionais na implementação deste método, como a existência da pilha: como devem ser mantidas cópias dos analisadores, a necessidade de se guardar cópias da pilha agrava sobremaneira o aumento do espaço necessário para a implementação. Porém as dificuldades na implementação deste método não tiram a sua importância histórica, pois foi o primeiro a considerar a hipótese de um recuo do texto já analisado na tentativa de refazer a análise visando a melhor recuperação de um erro.

O primeiro relato de um método de recuperação de erros utilizando o "backtracking" que pudemos encontrar é devida a Susan Graham e Stephen Rhodes [10]. É importante ressaltar que apesar de sua publicação ser quase simultânea com a do método de Lévy, este foi desenvolvido anteriormente, sendo publicado em um relatório técnico de 1971 [57].

No método de Graham e Rhodes o princípio do recuo da pilha é mantido, mas com um mecanismo de funcionamento totalmente diverso. Esboçaremos a seguir o funcionamento do método devido a repercussão dos princípios por ele utilizados. Não será porém fornecida uma descrição detalhada devido ao fato de que ele foi desenvolvido para analisadores de precedência e não para analisadores LR.

A recuperação é feita em duas fases: a fase de condensação e a fase de correção. O objetivo da fase de condensação é resumir o contexto ao redor do PDE, e o objetivo da fase de correção é encontrar a recuperação propriamente dita. A fase de condensação é realizada em duas etapas: na

primeira delas, o “backward-move”, são feitas tentativas de reduções no topo da pilha, e na segunda delas, o “forward-move”, tenta-se prosseguir a análise sintática do texto após o PDE até que um novo erro seja encontrado ou até que o prosseguimento da análise exija uma redução que envolva o PDE. Em ambos os casos é esperada a ocorrência de reduções, daí o nome “condensação”.

Ex Seja o seguinte trecho de um programa :

```
program A;
var X,I,J:integer;
begin
  X:= I J;
end.
```

a configuração de erro será:

(‘prog’ ‘id’ <vardecl> ‘begin’ <vaiden> ‘:=’ ‘id’, ‘id’; ‘end’.)

no “backward-move” obteremos:

(...‘begin’ <stat>, ‘id’ ‘;’...)

no “forward-move” obteremos (o caracter ‘?’ indica o local do PDE):

(...‘begin’ <stat> ? <expression> , ; ...)

onde ‘id’ foi reduzido a <expression>.

A fase de correção é acionada após a fase de condensação. Nesta fase são ensaiadas modificações no trecho da pilha sintática ao redor do PDE de maneira a fazer com que algum sufixo da pilha sintática que inclua o PDE forme o prefixo de um lado direito de uma produção da gramática .

Ex Seja o programa :

```
program B;
var I,K,M:integer;
begin
  if I := 2*M ;a:=b
end.
```

a configuração de erro é:

(...‘begin’ ‘if’ <vaiden> , ‘:=’ 2^*M ; a := b end .)

após o “backward-move” :

(...‘begin’ ‘if’ <expression> , ‘:=’ 2^*M ...)

após o “forward-move” (o símbolo ‘?’ indica o PDE):

(...‘begin’ ‘if’ <expression> ? ‘:=’ <expression> , ‘;’ a := b end .)

Temos neste caso, mais de uma correção possível, uma delas seria eliminar ‘if’ e substituir <expression> por <vaiden>, e outra seria inserir ‘then’ e <vaiden> antes de ‘:=’. A segunda solução é a melhor, já que se aproxima mais do texto original.

Para a escolha da melhor solução é atribuído um custo a cada uma delas. Este custo é calculado pela “distância ponderada” em relação ao texto original. Para o cálculo da distância ponderada são designados custos de inserção e eliminação para cada símbolo. A “distância” é calculada somando-se o custo das operações envolvidas na recuperação. Chamaremos no restante deste capítulo de $CI(a)$ o custo de inserção do símbolo ‘a’ e $CE(a)$ o seu custo de eliminação.

Desta maneira no exemplo anterior teríamos que o custo da primeira correção seria $CE(\text{‘if’}) + CE(\text{<expression>}) + CI(\text{<vaiden>})$ e o da segunda seria $CI(\text{‘then’}) + CI(\text{<vaiden>})$

Os custos de eliminação e inserção de cada símbolo são definidos pelo implementador, que pode desta maneira “sintonizar” o mecanismo de correção conforme a situação. A utilização de um sistema de custos associado a operações de edição foi uma das grandes inovações deste método, sendo posteriormente utilizado por vários autores, inclusive em esquemas de recuperação local como é o caso do método de Fischer Dyon e Mauney, abordado no capítulo anterior.

Graham e Rhodes sugerem também que se estabeleça um limite para o custo de uma recuperação: se nenhuma das recuperações encontradas tiver um custo menor do que este limite, o método é abandonado, sendo acionado um método alternativo, como por exemplo o panic mode.

A implementação do “backward move” sugerido por Graham e Rhodes não é simples em analisadores LR, como observaram Mikunas e Modry [11]. Para isto seria necessária a manutenção de uma lista dos lados direitos de cada produção para identificação das possíveis reduções. Porém em linguagens LR não há a necessidade de que produções diferentes possuam

lados direitos diferentes. Mais que isso, o lado direito de uma produção pode ser uma subcadeia de outro lado direito. Isto sem contar o fato de que uma redução na pilha sintática pode ocultar possibilidades de correção como no primeiro exemplo onde a redução do trecho $\langle \text{vaiden} \rangle := \text{'id'}$ ao não terminal $\langle \text{stat} \rangle$ elimina a possibilidade de inserção de um operador aritmético como correção, alternativa no caso mais razoável que qualquer outra.

Estas dificuldades e desvantagens provavelmente explicam porque nenhuma implementação para analisadores LR dos princípios de recuperação introduzidos pelo método de Graham e Rhodes faz uso do "backward move" por eles sugerido. Por outro lado os princípios de condensação do contexto direito e de inclusão da pilha no contexto de recuperação ("backtracking") tiveram grande repercussão e motivaram vários métodos de correção regional. A seguir descreveremos alguns deles.

4.3.1 O método de Mikunas e Modry

Em 1978 Mikunas e Modry publicam um artigo [11] com um método de correção cujo objetivo explícito era implementar os princípios fundamentais do método de Graham e Rhodes em analisadores LR.

O método de Mikunas e Modry também tem duas fases, mas a fase de condensação foi simplificada, eliminando-se o "backward move". A condensação do contexto à esquerda do PDE é eventualmente feita na segunda fase da recuperação quando, se necessário, símbolos da pilha sintática são incorporados ao contexto condensado à direita do PDE.

Como o "backward move" foi eliminado, a fase de condensação se limita ao "forward move". Para executá-lo calculamos inicialmente o conjunto:

$$\mathcal{F} = \{p \in Q \mid \text{Atitude}(p, \text{Prim}(\beta)) = \text{avanço}\},$$

que é o conjunto dos estados que aceitam o primeiro símbolo da cadeia de entrada como símbolo de transição para outro estado. Após calcularmos o conjunto $\mathcal{F}$ tentamos prosseguir a análise a partir de cada um dos estados de $\mathcal{F}$. Esta análise prossegue até que:

- a) outro erro é encontrado ou
- b) o analisador tenta efetuar uma redução que envolva o PDE

Os estados de $\mathcal{F}$ a partir dos quais foram feitas análises que terminam por b) são chamados **candidatos a recuperação** e os outros **candidatos em espera**

A fase de correção é aplicada aos candidatos a recuperação. Se a fase de correção for bem sucedida para algum deles é escolhida a alternativa com menor custo e em caso de empate, aquela com menor número de operações de edição. Se, ao contrário, a fase de correção falhar para todos os candidatos a recuperação, o método é invocado recursivamente a partir do novo PDE de cada um dos candidatos em espera, na tentativa de corrigir o segundo erro detectado, independentemente.

Vejamos os algoritmos da primeira fase:

Algoritmo Condensação M.M

entrada: β - cadeia de entrada

saída: Recup - conjunto de estados candidatos a recuperação

Espera - conjunto de estados candidatos em espera

com1

$\mathcal{F} := \emptyset;$

Recup := $\emptyset$;

Espera := $\emptyset$;

para cada $q \in Q$ **faça**

se $\text{Atitude}(q, \text{Prim}(\beta)) = \text{avanço}$

então $\mathcal{F} := \mathcal{F} \cup \{q\}$;

(*calculou $\mathcal{F}^*$)

para cada $p \in \mathcal{F}$ **faça**

com2

$\text{Analise\_limitada}((p, \beta), \infty, \text{Erro}, \text{Avanço}, (\gamma, \beta'),);$

(*analisa até redução envolver PDE*)

se Erro

então $\text{Espera} := \text{Espera} \cup \{(p, \gamma, \beta')\}$

senão $\text{Recup} := \text{Recup} \cup \{p\}$

fim2

fim1.

Na fase de correção tentamos recuperar o erro utilizando um candidato a recuperação p fornecido. Para isto tentamos encontrar $x \in V_t$ t.q.:

$$(\alpha, x\beta) \rightarrow \rightarrow \rightarrow (\alpha p, \beta)$$

Se tal símbolo é encontrado então sua inserção fornece uma recuperação ao erro.

def 4.3.1 Se existe uma redução onde o analisador está no estado q antes da redução e no estado p após a redução, dizemos então que q é um antecessor redução de p e que p é um sucessor redução de q

$D(q)$ denota o conjunto de sucessores redução do estado q

$A(q)$ denota o conjunto de antecessores redução do estado q

A noção de antecessor e sucessor nos permite generalizar um pouco mais a idéia do algoritmo da fase de correção. Para um dado símbolo terminal x , é possível que sua presença no início da cadeia de entrada provocasse reduções na pilha. Por este motivo devemos verificar para cada símbolo terminal y se

$$\exists r \in D(\text{Ult}(\alpha)) \text{ t.q. } \text{Atitude}(r, y) = \text{avanço e Destino}(r, y) = p.$$

mas da mesma maneira a inserção de x poderia provocar reduções em um sufixo da pilha, isto é em um sufixo de $\text{sim}(a_1) \dots \text{sim}(a_n)x$, onde a pilha α é igual a $(a_1 \dots a_n)$. Desta maneira devemos verificar, para cada símbolo terminal x se:

$$\exists r \in D(\text{ult}\alpha) \text{ e } \exists s \in A(p) \text{ t.q. } \text{Atitude}(r, x) = \text{avanço e Destino}(r, x) = s.$$

Se nenhum símbolo terminal, quando inserido, efetuar a ligação descrita acima, então é feito um recuo na pilha, numa tentativa de verificar se o ponto real de erro não está na parte já analisada da cadeia. Vejamos o algoritmo:

Algoritmo Fase correção M_M

entrada: (α, β) - configuração de erro do analisador

p - estado candidato à correção

saída: Sucesso - variável booleana indicando o sucesso da operação.

com1

$\alpha' := \alpha;$


```

 $\beta' := \beta;$ 
Custo := 0;
Sucesso := falso;
P := p;
AP := Ap(p);
enquanto não Sucesso e  $\alpha' \neq \lambda$  faça
  com2
    q := Ult( $\alpha'$ );
    para cada  $(r, x) \in (D(q) \times V_t)$  faça
      se Destino(r, x)  $\in$  AP
        então
          com3
            Soluções := Soluções  $\cup$ 
               $\{(\alpha, x \text{ cat } \beta', \text{Custo\_Ins}(x))\};$ 
            Sucesso := verdade
          fim3;
        se não Sucesso (*se não encontramos o terminal desejado*)
          então com3
            P' :=  $\emptyset$ ;
            AP' :=  $\emptyset$ ;
            para cada p  $\in$  Q faça
              (*para cada estado do analisador*)
              se Destino(p, Sim(Ult( $\alpha'$ )))  $\in$  AP
                então
                  com4
                    P' := P'  $\cup$  {p};
                    AP' := AP'  $\cup$  Ap(p)
                  fim4;
              (*calculou o conjunto de estados que podem
                levar a algum estado de AP utilizando
                como símbolo de transição o último símbolo
                da pilha sintática*)
            se P'  $\neq \emptyset$ 
              (* se existe um estado cuja transição pelo último
                símbolo da pilha sintática leva a
                um estado de AP então o símbolo pode ser
                incorporado ao contexto direito, isto é à cadeia

```

```

    de entrada, sem gerar erro*)
então
    com4
         $\beta' := \text{sim}(\text{Ult}(\alpha')) \text{ cat } \beta'$ ;
         $P := P'$ ;
         $AP := AP'$ ;
    fim4
senão
    (*se último símbolo da pilha sinática
    não pode ser incorporado ao contexto
    da direita sem gerar erro, ele deve
    ser eliminado*)
    se  $\text{sim}(\text{Ult}(\alpha')) \in V_t$ 
        (*se o símbolo é terminal, elimino-o*)
        então
            com4
                 $\text{Custo} := \text{Custo} + \text{Elim}(\text{sim}(\text{Ult}(\alpha')))$ ;
                 $\alpha' := \text{Ult}(\alpha')$ ;
                se  $D(\text{Ult}(\alpha')) \cup P \neq \emptyset$ 
                    (*se eliminação resolve o problema*)
                    então
                        com5
                             $\text{Soluções} := \text{Soluções} \cup$ 
                                 $\{((\alpha', \beta', \text{Custo}))\}$ ;
                             $\text{Sucesso} := \text{verdade}$ 
                        fim5
                    fim4
            senão (*se símbolo é não terminal,
            devo "expandi-lo" para
            evitar uma perda muito
            expressiva do prog original*)
                com4
                    (*suporemos  $\alpha = a_1 \dots a_n$  e
                    que a parte de  $\beta$  já consumida
                    é  $\delta = d_1 \dots d_n d_{n+1} \dots d_{n+k}$ *)

                     $\beta' := d_{n+1} \text{ cat } \dots \text{ cat } d_{n+k} \text{ cat } \beta'$ ;

```



```

Analise_limitada(( $\alpha'$ , $\beta'$ ),k-1,Erro,
                  Avanço,( $\alpha'$ , $\beta'$ ));
 $\alpha'$ :=Destino(Ult( $\alpha'$ ),Prim( $\beta'$ ));
 $\beta'$ :=Final( $\beta'$ )
(*paro análise imediatamente
antes da redução do não terminal*)
fim4
    fim3
fim2
fim1.

```

Algoritmo Recuperação_M_M

entrada: (α,β) - configuração do analisador

saída: Sucesso - variável booleana indicando sucesso do algoritmo
       (α,β) - nova configuração do analisador caso seja encontrada
       recuperação

```

com1
  Condensação_M_M( $\beta$ ,Rec,Espera);(*executafase de condensação*)
  Sucesso:=falso;
  para cada  $q \in \text{Rec}$  faça(*para cada candidato à recuperação*)
    com2
      Fase_correção_M_M(( $\alpha,\beta$ ),q,Suc,Soluções);
      se Suc então Sucesso:=verdade;
    fim2
  (*executa fase de correção*)
  se não Sucesso (*se fase de correção não foi bem sucedida *)
    então
      para cada ( $p,\gamma,\delta$ ) $\in$ Espera faça
        com2
          Recuperação_M_M(( $\gamma,\delta$ ),Sucesso);
          (*tenta corrigir segundo erro*)
          se Sucesso (*conseguiu*)
            então se  $/\gamma/ < /\alpha/$ 
              (*se correção do segundo
              erro envolveu o primeiro*)

```

```

então
  com3
     $\alpha := \gamma$ ;
     $\beta := \delta$ ;
  fim3
senão (*preciso corrigir erro original*)
  com3
    enquanto  $\gamma \neq \alpha$  faça
      com4
         $\delta := \text{sim}(\text{Ult}(\gamma)) \text{ cat } \delta$ ;
         $\gamma := \text{Início}(\gamma)$ 
      fim4
      (*volto ao prim PDE*)
       $\beta := \delta$ ;
       $\alpha := \gamma$ ;
      Recuperação_MM( $(\alpha, \beta)$ , Sucesso)
    fim3
  se Sucesso então abandone para
  fim2
senão (*fase de correção foi bem sucedida*)
  com2
     $(\gamma, \delta, C) := \text{Menor\_custo}(\text{Soluções})$ ;
     $\alpha := \gamma$ ;
     $\beta := \delta$ 
  fim2(*implementa recuperação de menor custo*)
fim1.

```

O método acima apresentado envolve o cálculo preliminar dos conjuntos $A(p)$ e $D(p)$, para cada estado p do analisador. Os autores não fornecem nenhum algoritmo para seu cálculo mas este pode ser facilmente executado juntamente com a geração dos estados do analisador. Para isto acrescentamos a cada item mais um elemento, um apontador. No item inicial do primeiro estado da gramática este apontador é nulo. Toda vez que um item é gerado pela ação de complementação associa-se a ele um apontador para próprio estado onde ele se encontra inserido. Quando um item é gerado pela ação do Scanner, o seu apontador tem o mesmo valor do apontador do item que o originou. Quando um estado q tem um item da forma $(A \rightarrow \delta; u; p)$, onde u é o conjunto de lookahead e p o estado indicado

pelo apontador, incluímos q no conjunto $A(p)$ e p no conjunto $D(q)$.

Os conjuntos $A(p)$ e $D(p)$ causam um considerável aumento no espaço necessário ao analisador pois ambos ocupam da ordem de $O(Q^2)$ posições da memória, constituindo uma desvantagem deste método.

4.3.2 O Método de Pennello e DeRemer

Pennello e DeRemer, a exemplo de Mikunas e Modry, publicam um método de recuperação [35] seguindo princípios do método de Graham e Rhodes mas executando condensação apenas no contexto à direita do PDE. Porém, enquanto que Mikunas e Modry executam as várias condensações possíveis individualmente, no método que descreveremos agora o princípio de condensação adotado garante que a condensação seja única, de certa maneira agrupando todas as condensações em uma só. Neste método, o contexto à direita do PDE permanece sempre condensado após a recuperação, impedindo, na medida do possível, repetição de inúmeros passos da análise para cada tentativa de recuperação. A condensação executada pode ser única porque ela é desenvolvida de maneira a independar completamente do contexto à sua esquerda, continuando válida quaisquer que sejam as modificações executadas em porções anteriores do contexto (como alterações da pilha do analisador). Vejamos um exemplo:

Ex Seja o seguinte programa :

```
program A;
var A,B,C,D:integer;
begin
  while A>B begin if A=C then C:=D ;A:=A+1 end ; A:=C
end.
```

o programa está errado pois falta um 'do'. Vejamos a configuração de erro:

('prog'id'<vardecl>'begin'while'<expr>'>' 'id', 'begin' if ....)

Após a fase de condensação teríamos:

(...while'<expr>'>' 'id', <stlist> end ; <stat> end .)

Como podemos ver no exemplo, o processo de condensação se dá em "blocos". A condensação será executada pelo algoritmo que denominaremos

Cond_P_DR . Para garantirmos a unicidade da condensação, é necessário criar um novo conjunto de estados e um novo conjunto de tabelas de Atitude e Destino para serem utilizados pelo algoritmo. O objetivo desta medida é possibilitar o acompanhamento simultâneo das várias possíveis análises individuais do texto enquanto elas concordem, para cada símbolo de entrada quanto à atitude a ser tomada. Cada bloco da condensação é encerrado quando a atitude a ser tomada não está definida, ou quando a sua continuação da condensação implicaria numa redução envolvendo símbolos à esquerda do início do bloco, ou quando é detectado um novo erro. A nova função Atitude não está definida quando as várias condensações possíveis não concordam quanto à atitude a ser tomada (isto é, quando existe um conflito). Desta maneira, só são tomadas, ações que independem completamente do contexto exterior ao bloco, e portanto que continuam válidas qualquer que seja a recuperação que só envolva símbolos daquele contexto. Após o encerramento de um bloco de condensação, novo bloco é iniciado, repetindo-se este processo até atingir-se o fim da cadeia de entrada. O primeiro bloco de condensação é iniciado com o símbolo sucessor do símbolo de entrada da configuração de erro.

Os estados utilizados pelo algoritmo Cond_P_DR são na verdade conjunto de estados do analisador original. Para calcularmos estes estados e a função de transição utilizaremos o algoritmo abaixo:

Algoritmo Constrói_transição

entrada: Q - o conjunto dos estados do analisador original

Atitude - tabela de atitudes do analisador

Destino - tabela de destinos do analisador

saída: K - conjunto de conjuntos de estados a ser utilizado pelo algoritmo de condensação

Atitude_cond - tabela de atitudes do alg. de condensação

Destino_cond - tabela de destinos do alg. de condensação

com1

• $K := \{Q\};$ (*estado inicial é o conjunto de todos os estados*)

para cada $(k,s) \in (K \times V)$ faça

    (*para cada par estado de K/símbolo da gramática*)

    com2

        Sucesso:=verdade;


```

Aux_atitude:='?';(*atitude de k frente a s ainda indefinida*)
Prim:=verdade;(*ainda vamos tratar o prim estado de k*)
para cada  $q \in k$  faça (*para cada estado do conj.  $k$ *)
    se  $Atitude(q,s) \neq Aux\_atitude$  (*se existem atitude
        diferentes quanto ao simbolo  $s$ *)
    então se Prim (*se é o prim estado de  $k$  tudo bem*)
        então
            com3
                Prim:=falso;
                Aux_atitude:=Atitude( $q,s$ );
                se Aux_atitude = avanço
                    então  $k\_aux:=\{Destino(q,s)\}$ ;
            fim3
        senão (*se não é o prim estado de
            então tenho um conflito*)
            se Aux_atitude = erro
                então Aux_atitude:=Atitude( $q,s$ )
                senão
                    se  $Atitude(q,s) \neq erro$ 
                        então vá para PÕEAÇÃO
                    (*se o conflito envolve erro, ignoro,
                    senão nem trata outros estados*)
            senão (*sem conflito com outros estados*)
                se Aux_atitude = avanço
                    então  $k\_aux:=k\_aux \cup Destino(q,s)$ ;
                    (*continua calculo do prox estado*)

```

PÕEAÇÃO:

```

(*preenche as tabelas p/ condensação*)
Atitude_cond:=Aux_atitude;
se Aux_atitude = avanço
    então
        com3
             $K:=k \cup \{k'\}$ ;
            (*acrescento estado destino ao
            conj. de estados de condensação*)
            Destino_cond( $k,s$ ):= $k'$ 
        fim3

```

fim3

fim2

fm1.

O cálculo dos estados e tabelas utilizados pelo algoritmo Cond_P_DR pode ser feito previamente, e portanto não afetam a eficiência de tempo da recuperação. O espaço adicional requisitado pode ser diminuído se considerarmos que, assim como nos analisadores LR, uma vez construídas as tabelas não é mais necessário guardar uma descrição detalhada dos estados.

Vejamos agora o algoritmo Cond_P_DR:

Algoritmo Condensa_um_bloco

entrada: β - configuração da cadeia de entrada do analisador
 Atitude_cond - tabela de atitudes obtida no algoritmo anterior
 Destino_cond - tabela de destino obtida no algoritmo anterior
 saída: Erro - variável booleana indicando se algum bloco foi interrompido pelo aparecimento de novo erro;
 Bloco - configuração do bloco condensado
 β' - porção não condensada de β

com1

```

  Erro:=falso;
   $\beta'$ := $\beta$ ;
  Pilh:= Q;(*pilha auxiliar recebe prim estado da condensação*)
  Bloco:= $\lambda$ (*bloco de condensação começa vazio*)
  Continua:=verdade;
  enquanto Continua faça
    caso Atitude_cond(Ult(Pilh),Prim( $\beta'$ )) seja
      avanço:com3
        Bloco:=Bloco cat Prim( $\beta'$ );
        ,
        Pilh:=Pilh cat Destino_cond(Ult(pilh),Prim( $\beta$ ))
         $\beta'$ :=Final( $\beta'$ )
    fim3
  reduz ( $A \rightarrow \delta$ ):
    se  $/Pilh/ < /\delta/$ 
      (*se envolve símbolos anteriores*)
    então Continua:=falso
      (*encerra o bloco*)

```



```

senão
    para Aux de 1 até / $\delta$ /
        faça com5
            Pilh:=Início(Pilh);
            Bloco:=Início(Bloco)
        fim5
        Bloco:=Bloco cat A;
        Pilh:=Pilh cat Destino_cond(Ult(Pilh),A)
    fim4
erro:com2
    Erro:=verdade;
    Continua:=falso
    fim2
outros:Continua:=falso(*ação indefinida-encerra bloco*)
fimcaso
fim1.

```

Algoritmo Cond_P_DR

entrada: β - configuração da cadeia de entrada do analisador
 Atitude_cond - tabela de atitudes obtida no algoritmo anterior
 Destino_cond - tabela de destino obtida no algoritmo anterior
saída: Erro - variável booleana indicando se algum bloco foi interrompido
 pelo aparecimento de novo erro;
 β' - porção condensada de β
 β - porção não condensada de β se houver erro

```

com1
    Erro:=falso;
     $\beta'$ := $\lambda$ ;
    enquanto  $\beta \neq \lambda$  e não Erro faça
        (*enquanto não acabei condensação e não encontrei erro*)
        com2
            Condensa_um_bloco( $\beta$ ,Atitude_cond, Destino_cond, Erro, Bloco,  $\beta$ );
             $\beta'$ := $\beta'$  cat Bloco;(*coloca outro bloco no contexto direito*)
        fim2
    fim1.

```

Após a condensação do contexto direito tem início a recuperação propriamente dita. A exemplo de outros métodos, são tentadas a eliminação e troca do símbolo de entrada, seguidas de uma tentativa de inserção. Cada recuperação é testada continuando-se a análise sintática do através do contexto direito condensado até o final deste. Esta verificação é feita pelo algoritmo Tenta. Se nenhuma das tentativas obtiver sucesso, tenta-se incorporar o símbolo de erro ao contexto direito e repetir o processo, utilizando agora como símbolo de entrada o símbolo do topo da pilha sintática, numa tentativa de buscar o erro em algum lugar da pilha sintática. Quando não é possível esta incorporação assume-se a hipótese de que o erro está certamente no trecho compreendido entre o primeiro e o último símbolo incorporados ao contexto direito. Neste caso duas ações diferentes são sugeridas pelos autores. Uma é a eliminação pura e simples de todo o trecho, em seguida reinvocando recursivamente o método de recuperação. A segunda solução é uma espécie de panic mode denominado "stack forcing", que chamaremos *manipulação forçada da pilha*. A exemplo dos autores do método, descreveremos a segunda opção.

Algoritmo Recuperação Penn DR

entrada: (α, β) - configuração de erro do analisador
           Atitude_cond - tabela de atitudes da condensação
           Destino_cond - tabela de destinos da condensação
            e_a - estado de aceitação do analisador
saída: (α, β) - configuração do analisador após a recuperação

com1

```

 $\alpha' := \alpha;$ 
Acabou := falso;
 $s := \text{Prim}(\beta);$ 
 $\beta' := \text{Final}(\beta)$ 
Cond_P_DR( $\beta'$ , Atitude_cond, Destino_cond, Erro,  $\beta'$ );
(*faz a condensação*)
  enquanto  $\alpha' \neq \lambda$  e não Acabou faça
    com2
      Tenta( $\alpha', \lambda, \beta', \text{Acabou}$ );
      (*tenta a eliminação de s*)
      se não Acabou (*se eliminação não deu certo*)

```



```

então
  para cada  $v \in V_t$  faça
    com2
      Tenta( $\alpha', v, \beta', \text{Acabou}$ );
      (*tenta troca de s por v*)
      se Acabou então abandone para
        fim2;
    se não Acabou (*se eliminação e troca fracassaram*)
      então Incorpora( $s, \beta', \text{Atitude\_cond}, \text{Destino\_cond}, \text{Erro}$ );
    se Erro (*incorporação não deu certo*)
      então Acabou:=verdade ;
    se não Acabou
      (*se nenhum anterior deu certo e incorporação bem sucedida*)
      então para cada  $v \in V_t$  faça
        Tenta( $\alpha', v, \beta', \text{Acabou}$ );
        (*tenta inserção de v*)
      se não Acabou (*nada deu certo mas incorporei s a  $\beta'$ *)
        então com3
          s:=sim(Ult( $\alpha'$ ));
           $\alpha'$ :=Início( $\alpha'$ );
          fim3
          (*novo s é o topo da pilha sintática*)
        fim2;
    se Erro (*houve problema de incorporação*)
      então Manipulação_forçada_pilha( $\alpha', \beta', \text{Nts}, \beta_{\text{depred}}, \text{Achou}$ );
      (*tenta a manipulação forçada da pilha*)
    se não Achou (* se nenhuma estratégia funcionou*)
      então
        com2
           $\alpha := e_0 e_a$ ;
           $\beta := \#$ ;
          fim2(*coloca analisador em estado de aceitação*)
        fim1.

```

Para realizarmos a incorporação do símbolo do topo da pilha sintática ao contexto direito utilizaremos o algoritmo incorpora.

Algoritmo Incorpora

entrada: h - símbolo a ser incorporado

β - contexto direito condensado

Atitude_cond - tabela de atitudes da condensação

Destino_cond - tabela de destinos da condensação

saída: Erro - variável booleana indicando se operação foi possível

β - modificado, se for o caso

com1

$\beta' := h \text{ cat } \beta;$

Condensa_um_bloco(β' , Atitude_cond, Destino_cond, Erro, Bloco, β');

verifica se é possível condensar nova cadeia*)

se não Erro

então $\beta := \text{Bloco cat } \beta';$

fim1.

Finalmente, temos os algoritmos auxiliares *Manipulação_forçada_pilha* e *Tenta*. O primeiro, de maneira muito semelhante ao *panic mode* de Wirth, tenta substituir um trecho da pilha e da cadeia de entrada condensada por um não terminal, simulando uma redução. A diferença está que, neste caso, o algoritmo utiliza uma lista de não terminais, eliminando estados da pilha até encontrar algum que aceite um não terminal da lista, realizando o avanço determinado. Em seguida são consumidos elementos da cadeia de entrada (condensada) até que seja encontrado um elemento que seja aceito pelo novo topo da pilha do analisador. Esta lista de não terminais é construída durante a fase de condensação. Se um bloco é encerrado por conflito de atitudes e se neste conflito está presente a redução pela produção $A \rightarrow \alpha$, então o não terminal A é acrescentado à lista. Esta estratégia envolve portanto modificações no algoritmo de condensação e no algoritmo de construção das tabelas *Atitude_cond* e *Destino_cond*. A cada entrada da tabela *Atitude_cond* com valor '?' corresponderá um conjunto de símbolos não terminais, indicando os lados esquerdos das produções indicadas pelas eventuais reduções envolvidas no conflito.

O algoritmo *Tenta* verifica se uma determinada recuperação é bem sucedida, tentando prosseguir a análise sintática a partir da configuração de recuperação. O algoritmo *Tenta* não passa portanto de um analisador sintático que prossegue a análise a partir da configuração fornecida até analisar toda a cadeia de entrada ou até encontrar um erro. Caso a análise

seja bem sucedida a configuração do analisador fornecida é alterada e a variável booleana recebe o valor 'verdade', caso contrário a configuração fornecida não é alterada e a variável booleana recebe o valor 'falso'.

O algoritmo de recuperação de Pennello e DeRemer só funciona na hipótese de ocorrência de apenas um erro sintático em toda a cadeia de entrada. Para resolver este problema os autores sugerem algumas soluções. Uma delas é, no caso do algoritmo de condensação detectar outros erros, aceitar, para cada um, uma recuperação que possibilite analisar com sucesso todos os símbolos contidos no intervalo entre ele e o próximo erro. Mas existem ainda casos que não são cobertos por esta solução: o algoritmo de condensação, devido a generalidade de sua análise pode não detectar alguns erros. Um exemplo disto é a sequência

...;X < Y + Z then then ....else..;

se a gramática contiver a produção $\langle \text{if clause} \rangle \rightarrow \text{if} \langle \text{exp} \rangle \text{then}$, o algoritmo de condensação encerra um bloco após o primeiro 'then', pois uma redução envolveria um símbolo antes do início do bloco. Em seguida é iniciado o segundo bloco com o próximo 'then', ficando o erro, desta maneira, despercebido. A solução sugerida neste caso é aceitar a recuperação que propiciar a análise correta de um número maior de símbolos.

4.3.3 a Recuperação não corretiva

A partir do algoritmo de condensação de Pennello e DeRemer pode-se implementar outro tipo de recuperação: a recuperação não corretiva, que consiste em, a partir da ocorrência do primeiro erro, executar apenas o algoritmo de condensação, indicando os locais onde este encontra erro, isto é, os pontos onde a atitude determinada pela tabela *Atitude.cond* é erro.

A noção de recuperação não corretiva foi idealizada por Richter [**], num artigo que analisa em detalhe as propriedades formais deste tipo de recuperação.

A grande vantagem deste tipo de recuperação, como lembra Richer, é a ausência total de erros espúrios, já que o algoritmo de condensação detecta um erro apenas quando há consenso sobre sua existência, seja qual for a análise do bloco. Outra grande vantagem é a rapidez, já que o tempo de análise permanece praticamente inalterado, esteja o programa correto ou não.

As desvantagens deste esquema são: o espaço necessário, já que um novo conjunto de tabelas de transição deve ser utilizado; o fato de que alguns erros permanecem não detectados; e, finalmente, a impossibilidade de execução da maioria das "rotinas semânticas" após a detecção do erro.

4.4.3 Métodos Sem Condensação

Existem também implementações de métodos de correção com "backtracking" que não utilizam a fase de condensação. É o caso dos métodos desenvolvidos por Feycock e Lazarus [22] e por Burke e Fischer [19]. Descreveremos inicialmente o método de Burke e Fischer, mais complexo, para em seguida delineararmos apenas as diferenças em relação ao método de Feycock e Lazarus. Não descreveremos ambos em detalhe devido à sua semelhança.

4.4.3.a) O método de Burke e Fischer

Este método não utiliza vetores de custos como os descritos anteriormente. No caso de várias soluções serem encontradas, a escolha é feita utilizando-se heurísticas fornecidas pelo implementador. Recuperações absurdas são evitadas através de uma consulta a uma "lista de proibições", relacionando os pares de símbolos que não devem ser trocados um pelo outro e as inserções que não devem ser feitas.

A recuperação se faz em duas fases: fase primária e fase secundária. Na fase primária são tentadas correções singulares, envolvendo apenas uma operação de edição. Na fase secundária é executado um algoritmo semelhante ao panic mode, onde a correção é feita através da eliminação de vários símbolos. Vejamos com mais detalhe:

A fase primária começa por tratar a configuração de erro do analisador. A partir do PDE, são tentadas, sucessivamente, a junção de símbolos, inserção, troca, recuperação de contexto e eliminação. A junção de símbolos é utilizada somente neste ponto, e consiste em concatenar o símbolo de entrada com o último símbolo reconhecido (o símbolo do topo da pilha sintática), e com o símbolo que o segue na cadeia de entrada. Tal estratégia visa corrigir erros do tipo escrever 'go to' em um programa PASCAL, quando o correto seria escrever 'goto'. A recuperação de contexto será tratada mais adiante e as operações de inserção, troca e eliminação de símbolos dispensam maiores comentários, pois já foram descritos

em estratégias anteriores. Os símbolos candidatos a inserção e troca pelo símbolo do início da cadeia de entrada são os símbolos aceitos pelo estado do topo da pilha do analisador.

Cada tentativa de correção é testada através de uma continuação da análise sintática por um número pré definido de símbolos (os autores utilizaram 25 em sua implementação). Todas as recuperações cujas análises lograrem este avanço são consideradas recuperações válidas, sendo a escolha determinada por heurísticas fornecidas pelo implementador, como já dissemos acima. Caso nenhuma correção atinja o limite pré estabelecido, são consideradas aquelas cuja análise lograr um avanço superior a um limite mínimo, também pré estabelecido. Destas é escolhida aquela cujo avanço foi maior. Finalmente, se nenhuma das tentativas logrou nem mesmo o avanço mínimo, o último símbolo da pilha sintática é retirado e inserido no início da cadeia de entrada, sendo o estado do topo da pilha do analisador eliminado, após o que reinicia-se todo o processo.

O recuo na pilha pela fase primária é executado até que se encontre uma recuperação que satisfaça os critérios acima ou até o aparecimento de um "iniciador de contexto" no topo da pilha sintática. O conjunto de iniciadores de contexto é determinado pelo implementador com o intuito de conter os símbolos que iniciam extruturas sintaticamente importantes da gramática, como por exemplo o 'begin' do PASCAL.

Quando a fase primária termina sem que tenha sido encontrada uma recuperação, o analisador é restaurado a sua configuração de erro original e é acionada a fase secundária, que executa o que denominaremos de recuperação de contexto.

A recuperação de contexto tem por objetivo completar certas extruturas sintáticas que se encontram incompletas na pilha sintática. As extruturas para as quais será tentada a recuperação de contexto são aquelas referentes aos iniciadores de contexto descritos acima.

Como o leitor já deve ter notado a recuperação de contexto é tentada tanto na fase primária como na secundária. Na fase primária a recuperação de contexto tem por objetivo completar as extruturas sintáticas que exigem mais de um símbolo de finalização, e que portanto não são tratadas pelas correções singulares, pois estas lidam com no máximo um símbolo novo. Um exemplo é o finalizador 'end record' da linguagem ADA: a omissão deste finalizador apesar de consistir em um erro relativamente simples, não poderia ser corrigida na fase primária não fosse a recuperação

de contexto. Porém, em linguagens onde os finalisadores possuem apenas um símbolo, a recuperação de contexto torna-se redundante na fase primária, pois passa a ser um caso particular da estratégia de inserção.

Na fase secundária os símbolos do topo da pilha vão sendo eliminados até que se encontre um para o qual a recuperação de contexto seja bem sucedida. A recuperação de contexto é considerada bem sucedida na fase secundária se ela propiciar a análise sem erro de Limite_mínimo + 2 símbolos da cadeia de entrada, onde Limite_mínimo é o limite pré estabelecido pelo implementador como o mínimo necessário para que uma recuperação seja adotada na fase primária. (os autores não fornecem explicações sobre a razão da constante 2 na expressão). Porém, se na recuperação de contexto a inserção de uma sequência finalizadora a análise sintática apenas ultrapassar os símbolos inseridos, o algoritmo é invocado recursivamente para a configuração alcançada. Provavelmente o objetivo é tratar casos como o da sequência abaixo:

```
begin
  ...
  begin
    ...
    begin
      ...
      begin
        end
      end
    end
  end
end.
```

onde é necessária a inserção de mais de uma sequência de finalizadores (no caso o símbolo 'end') para recuperar o erro.

Para se otimizar um pouco o algoritmo aconselha-se a manutenção de uma lista dos iniciadores de contexto presentes na pilha sintática com a finalidade de utilizar na recuperação de contexto apenas os finalizadores que podem vir a propiciar uma recuperação.

Algoritmo Verifica_solução

entrada: (α, β) - configuração do analisador

Soluções - lista das recuperações já encontradas

Sucesso - variável booleana indicando se foi encontrada

solução com avanço máximo na análise
 Maxsimb - avanço máximo da análise
 Minsimb - avanço mínimo aceitável da análise
 saída: Sucesso, Soluções - modificados, se for o caso
 com1
   Análise_limitada $((\alpha, \beta), \text{Maxsimb}, \text{Avanço}, \text{Erro}, (\alpha', \beta'))$;
   (*tenta a análise*)
   se não Erro (*se análise foi até o máximo*)
   então se Sucesso (*se já tivemos outra solução do tipo*)
   então Soluções := Soluções $\cup \{(\alpha', \beta', \text{Maxsimb})\}$
   (*acrescenta à lista*)
   senão (*se é a primeira*)
     com2
       Sucesso := verdade;
       Soluções := $\{(\alpha, \beta, \text{Maxsimb})\}$
       fim2 (*inicializa lista*)
   senão (*análise não foi até o máximo*)
   se Avanço > Minsimb e não Sucesso
   (*se avanço maior que o mínimo e não houve
   outro melhor*)
   então Soluções := Soluções $\cup \{(\alpha', \beta', (- \text{Avanço}))\}$
 fim1.

Algoritmo Recuperação_B_F

entrada: (α, β) - configuração do analisador
 Maxsimb - avanço máximo da análise nos testes de recuperações
 Minsimb - avanço mínimo para recuperação ser aceita
 Iniciadores - conjunto dos iniciadores de contexto
 Finalizador - tabela com uma entrada para cada iniciador
                   de contexto, contendo o seu finalizador associado
 Junta_símbolos - função que, dados dois símbolos devolve a sua
                   concatenação
 saída: (α, β) - nova configuração do analisador
 com1
   Sucesso := falso;
   Soluções := $\emptyset$; (*inicializa variáveis*)

```

Simb:=Junta_símbolos(Prim( $\beta$ ),Prim(Final( $\beta$ )) );
(*junta dois primeiros símbolos de entrada*)
Verifica_solucões( ( $\alpha$ ,Simb cat Final(Final( $\beta$ )) ), ....);
se Sim(Ult( $\alpha$ )) $\in V_t$ (*se símbolo do topo da pilha sint. for terminal*)
então com2
    Simb:=Junta_símbolos( sim(Ult( $\alpha$ )),Prim( $\beta$ ) );
    (*junta último símb. da p.sint. com o simb de entrada*)
    Verifica_solucões( Início( $\alpha$ ),Simb cat Final( $\beta$ ) ), ....);
fim2;
(*acabamos junção de símbolos, que só é feita ao redor do PDE*)
Acabou:=falso;(*variável de controle*)
 $\alpha'$ := $\alpha$ ;(*copia a pilha*)
 $\beta'$ := $\beta$ ;(*copia a cadeia de entrada*)
repita
    para cada  $s \in V_t$ .q.Atitude(Ult( $\alpha'$ ), $s$ ) $\neq$ erro faça
        Verifica_solucões( ( $\alpha'$ , $s$  cat  $\beta'$ ) ,...);
        (*testa as inserções*)
    para cada  $s \in V_t$ .q.Atitude(Ult( $\alpha'$ ), $s$ ) $\neq$ erro faça
        Verifica_solucões( ( $\alpha$ , $s$  cat Final( $\beta'$ ) ),...);
        (*testa as trocas*)
    Recuperação_contexto( ( $\alpha'$ , $\beta'$ ),Maxsimb,Iniciadores,
        Finalizador,Achou,Avanço, ( $\alpha''$ , $\beta''$ ) );
    se Achou (*se rec. de contexto funcionou*)
        então se Sucesso (*se já tivemos outra solução do tipo*)
            então Solucões:= Solucões $\cup$ {( $\alpha''$ , $\beta''$ ,Maxsimb)}
            (*acrescenta à lista*)
            senão (*se é a primeira*)
                com3
                    Sucesso:=verdade;
                    Solucões:={( $\alpha$ , $\beta$ ,Maxsimb)}
                fim3 (*inicializa lista*)
            senão se Avanço>Minsimb e não Sucesso
                (*atingiu o mínimo? e não houveram melhores?*)
                então Solucões:=Solucões $\cup$ {( $\alpha''$ , $\beta''$ ,(- Avanço) )} ;
    Verifica_Solucões( ( $\alpha'$ ,Final( $\beta'$ ) ),....);
    (*testa a eliminação*)
    se Solucões $\neq \emptyset$ 

```



```

    então Achou:=verdade(*indica sucesso*)
    senão com3
         $\beta' := \text{sim}(\text{Ult}(\alpha')) \text{ cat } \beta'$ ;
         $\alpha' := \text{Início}(\alpha')$ ;
        fim3(*incorpora ult. simb. da p.sint.à  $\beta'$ *)
    até que Acabou ou  $\text{sim}(\text{Ult}(\alpha')) \in \text{Iniciadores}$ 
        (*repete enquanto não achar solução e não encontrar iniciador
        no topo da pilha sintática*)
    se Soluções  $\neq \emptyset$  (* se achou alguma solução*)
        então se Sucesso (*é solução com avanço máximo?*)
            então <aplique heurística determinada
                pelo implementador>
            senão com3
                 $(\delta, \gamma, C) := \text{Menor\_custo}(\text{Soluções})$ ;
                 $\alpha := \delta$ ;
                 $\beta := \gamma$ ;
            fim3
        senão (*fase primária não teve sucesso*)
            enquanto não Sucesso
                com2
                    Avanço:=0;
                     $\alpha := \text{Início}(\alpha)$ ; (*elimina topo da p.sintática*)
                    Recuperação_contexto(( $\alpha, \beta$ ), ..., Sucesso, Avanço, ( $\alpha'', \beta''$ ));
                    se Sucesso ou Avanço > Minsimb+2
                        (*se chegou ao fim ou completou avanço min.*)
                        então
                            com3
                                 $\alpha := \alpha'$ ;
                                 $\beta := \beta'$ ;
                            fim3
                        senão  $\alpha := \text{Início}(\alpha)$ ;
                    fim2
                fim2
            fim1.

```

Algoritmo Recuperação_contexto

entrada: (α, β) - configuração do analisador

Maxsimb - avanço máximo de teste de análise

Iniciadores - conj. dos iniciadores de contexto

Finalizador - tabela com uma entrada para cada iniciador de contexto, contendo o seu finalizador associado

saída: Achou - variável booleana indicando sucesso

Avanço - quantidade de símbolos da cadeia de entrada analisados com sucesso. Este parametro deve ser inicializado antes do algoritmo

(α', β') - nova configuração do analisador

com1

Achou:=falso;

para cada $s \in \text{Iniciadores}$ faça

com2

$\mu := \text{Finalizador}(s);$ (* μ recebe cadeia finalisadora*)

Análise limitada $((\alpha, \mu \text{ cat } \beta), \text{Maxsimb}, \text{Aux}, \text{Erro}, (\alpha', \beta'))$;

se $\text{Erro} \neq \mu$ (*se apenas ultrapassou finalizador*)

então Recuperação_contexto((α', β') , Maxsimb, Achou, Avanço, Iniciadores, Finalizador, (α', β') ;

(*chama o algoritmo recursivamente*)

senão Achou:=verdade; (*achou solução*)

se Achou

então abandone para;

fim2

fim1.

Ex Seja o seguinte programa :

prog P;

label 3;

var A,B: integer;

begin

A:=B then goto 3 else B:=0;

3: A:=1

end.

A configuração de erro será:

('prog'id'; '<vardecl>'begin'<vaiden>':='id', 'then' goto 3 else...)

Suponhamos 'begin' como iniciador de contexto e Lim_min=3.

Fase primária:

a) Configuração atual:

- junção de símbolos: não funciona ('thenA' e 'Bthen' não tem sentido)
- inseção: não funciona ('then' não pode seguir nenhum dos símbolos)
- troca: não funciona (análise não ultrapassa limite:para em 'else')
- recup. contexto: não tem sentido (finalizadores só tem 1 símbolo)
- eliminação: não funciona ('goto' não pode seguir 'id')

b) Configuração (...<vaiden> ':=' , 'id' 'then' 'goto'...)

devido à presença de 'then' nada ultrapassa o limite mínimo de análise

c) Configuração (...<vaiden> , ':=' 'id' 'then' 'goto'...)

novamente nenhuma estratégia tem sucesso, já que o estado do topo da pilha será composto apenas do item:

(<stat> → <vaiden> . ':=' <expr>; ...)

d) Configuração (... 'begin' , <vaiden> ':=' 'id' 'then' 'goto'...)

neste caso os símbolos aceitos pelo analisador são: 'goto', 'if', 'while', 'for', um numero, um identificador.

- inserção: nenhuma ultrapassa o limite mínimo ('if' devido à presença de ':=' e os outros devido a <vaiden>)
- troca: idem (devido à presença de ':=')

Fase Secundária:

como o analisador atingiu um iniciador de contexto a fase secundária é iniciada restaurando o analisador à configuração original.

A inserção de 'end' provoca o avanço de apenas um símbolo provocando a chamada recursiva do algoritmo que provoca a inseção de '#' que encerra a análise.

Ex Seja o programa

```

prog P2;
var A,B,C:integer;
begin
  A:=B C ; if A > D then A:=D;
end.

```

A configuração de erro:

('prog' 'id' <vardecl> 'begin' <vaiden> ':=' 'id' , 'id'; 'if'...)

Neste caso a junção dos dois identificadores resolve o problema sintaticamente, já que 'AB' é também um identificador. Porém neste caso

geramos um erro semântico já que foi criado um identificador não declarado. Na hipótese de erros como este serem considerados, o algoritmo tenta a inserção, neste caso com sucesso pois no nosso caso a inclusão de um operador aritmético resolve o problema.

Como pudemos ver no primeiro exemplo, o método de Burke e Fischer pode, quando ocorrerem dois erros com distância menor que o limite mínimo de avanço estabelecido, inserir uma sequência de símbolos que encerre prematuramente a análise sintática, impedindo o diagnóstico de parte significativa da cadeia de entrada. Os autores não comentam esta dificuldade, não propondo portanto nenhuma solução. Uma maneira de contornar esta dificuldade seria impedir a inserção do símbolo designativo de fim de cadeia ('#') na recuperação. Com esta medida porém, o método não será sempre bem sucedido em encontrar uma recuperação, criando a necessidade de se implementar um método alternativo que seria acionado quando o principal falhasse.

4.4.3b) O método de Feycock e Lazarus

Em 1976 Feycock e Lazarus apresentam um método de recuperação de erros [22] cuja estrutura básica foi, aparentemente, utilizada por Burke e Fischer. Devido à semelhança entre ambos os métodos, não descreveremos o de Feycock e Lazarus em Detalhe, limitando-nos a enumerar as diferenças:

i) Não são realizadas junção de símbolos e nem recuperação de contexto

ii) Apesar do critério de aceitação de uma recuperação estar condicionado a uma análise do texto ainda não reconhecida em busca de erros, esta análise não é limitada pelo número de símbolos analisados com sucesso e sim pela detecção do fim de uma unidade sintática considerada importante (isto é, de um finalizador de contexto)

iii) Caso não seja encontrada nenhuma solução satisfatória, todo o trecho que vai desde o primeiro iniciador de contexto encontrado na pilha até o primeiro encerrador equivalente encontrado na cadeia de entrada é eliminado (este dispositivo aliás poderia ser implementado para resolver o problema por nós levantado do método de Burke e Fischer).

Capítulo 5

Recuperação de erros Global

Os métodos globais de recuperação de erros sintáticos são aqueles que levam em conta a gramática da linguagem a ser reconhecida como um todo, implicando a consideração de um contexto de certa maneira ilimitado. Dois grupos de métodos de recuperação global são encontrados na literatura. O primeiro deles denominaremos métodos de recuperação implícita e o segundo de método de recuperação de contexto global, utilizando a nomenclatura dos próprios autores (Pai e Kieburz [13]).

A recuperação implícita não se processa através de um algoritmo separado, como é o caso dos outros métodos de recuperação, mas é feita implicitamente pelo próprio analisador sintático durante o reconhecimento da cadeia de símbolos. Sua estratégia se resume em acrescentar à gramática original novas produções, que denominaremos produções erro, nas quais estão previstos erros de sintaxe. Em outras palavras, acrescentando novas produções à gramática, permitimos ao analisador sintático aceitar, sem

entrar em estado de erro, cadeias de terminais que não são sentenças da gramática original.

O método de recuperação de contexto global de Pai e Kieburz tem funcionamento semelhante ao do panic mode (ver cap.3). Quando ocorre um erro o algoritmo de recuperação consome símbolos da cadeia de entrada até encontrar um símbolo pertencente a um classe especial, a dos símbolos confiáveis. Em seguida a pilha é reorganizada de maneira a aceitar o símbolo encontrado. Apesar da semelhança com o "panic-mode", existem duas diferenças importantes: em primeiro lugar a classe dos símbolos confiáveis é muito bem definida, e em segundo lugar a reorganização da pilha não é feita somente pela eliminação de estados, mas também pela expansão de estados da pilha do analisador cujo símbolo de entrada é um não terminal.

3.1 A Recuperação implícita

Como já dissemos, denominaremos recuperação implícita aquela que é feita diretamente pelo analisador, sem a aplicação de um algoritmo independente de recuperação. Isto é possível acrescentando-se à gramática original produções erro, que são modificações da gramática original com o objetivo de prever determinados tipos de erro.

Ex Na gramática:

$$\begin{aligned} S &\rightarrow Z\# \\ S &\rightarrow a \text{ ':='} a \text{ ';' } S \\ S &\rightarrow \lambda \end{aligned}$$

podemos fazer com que a troca de ':= ' pr '=' seja recuperada implicitamente adicionando à gramática a produção: $(S \rightarrow a \text{ '=' } a \text{ ';' } S)$.

A idéia de acrescentar produções erro à gramática, estendendo a linguagem reconhecida pelo analisador não é nova. Já em 1972 Aho e Peterson [21] propõe um esquema de recuperação implícita. Nele as produções erro são acrescentadas de maneira que a nova gramática G' tenha $L(G') = V_t^*$, isto é de maneira que todos os possíveis erros estejam previstos. Além disto tudo é feito de maneira que o analisador utilize o menor número possível de produções erro, simulando uma correção de distância de Hamming mínima.

Infelizmente a gramática gerada por este processo não é necessariamente determinística, o faz com que o método seja incompatível com a análise LR, fugindo ao escopo deste trabalho.

Desta maneira, para nos utilizarmos da correção de erros implícita em analisadores do tipo LR, é necessário restringir as produções erro de maneira que as gramáticas permaneçam determinísticas, de preferência LR(1). Neste sentido, podem ser encontradas duas abordagens na literatura.

A primeira delas, feita por Fischer e Mauney [15], considera a correção implícita como uma maneira de aumentar, eficientemente, a capacidade de recuperação de algoritmos já existentes. Nesta filosofia, as produções erro seriam introduzidas como maneira de resolver certos casos patológicos, onde um determinado algoritmo de recuperação não funciona eficientemente, como é o caso, por exemplo, do famoso erro “; else” do PASCAL, que raramente é recuperado de maneira satisfatória. Não existe nenhuma sistematização, por parte de Fischer e Mauney, no processo de geração de produções erro, limitando-se os autores a descrever algumas soluções particulares, notadamente para erros na linguagem PASCAL.

A segunda abordagem de recuperação implícita, feita pelo polonês Krazwyck [32], visa corrigir, através de produções erro, erros de eliminação e troca de determinadas classes de símbolos. As classes de símbolos utilizadas por Krazwyck, bem como o processo de geração das produções erro, são bem definidos, possibilitando a automatização do processo.

A utilização de produções erro porém não se restringe à recuperação implícita. O famoso YACC [10] utiliza este conceito de maneira diversa: são acrescentados, a critério do implementador, produções com um símbolo terminal especial, não pertencente à gramática original, denominado “erro”. Nestas produções o símbolo “erro” indica pontos onde a ocorrência de erros é mais provável. Ao ser detectado um erro, o terminal ‘erro’ é colocado no início da cadeia de entrada. Em seguida, vão sendo eliminados elementos da pilha até que o estado do topo da pilha do analisador aceite o referido símbolo. Neste momento é efetuado o avanço sobre o símbolo ‘erro’, passando o algoritmo a descartar símbolos da cadeia de entrada até encontrar um que seja aceito pelo estado do topo da pilha. Esta operação é repetida se o analisador não for capaz de avançar com sucesso os próximos dois símbolos. Opcionalmente pode ser associada à produção erro uma rotina semântica especial.

Ex Nesta metodologia de recuperação, se acrescentarmos a produção

$\langle \text{stat} \rangle \rightarrow \text{erro} ;$

a

nossa gramática: Ao ser encontrado um erro em um comando, são eliminados estados da pilha até atingir-se o estado onde se iniciou o reconhecimento deste comando e em seguida são descartados símbolos da pilha até um ';' ser encontrado. Desta maneira, para o programa:

```

prog A;
var A,B,C: integer;
begin
  A:= 2 * 3 + D;
  if A:= B then A:=0;
  while ...
  ...
end.

```

a configuração de erro será:

$(\text{'prog' id} \langle \text{vardecl} \rangle \text{'begin'} \langle \text{stat} \rangle \text{'if' id} , \text{' := ' B' 'then' ...})$

após o recuo da pilha teremos:

$(\text{'prog' id} \langle \text{vardecl} \rangle \text{'begin'} \langle \text{stat} \rangle , \text{'erro' := ' B' 'then' ...})$

e

então efetuado o avanço sobre o terminal 'erro':

$(\text{'prog' id} \langle \text{vardecl} \rangle \text{'begin'} \langle \text{stat} \rangle \text{'erro' , ' := ' B' 'then' ...})$

em seguida a cadeia de entrada é consumida até ser encontrado um símbolo aceito pelo estado do topo da pilha (no nosso caso procuramos um ';'):

$(\text{'prog' id} \langle \text{vardecl} \rangle \text{'begin'} \langle \text{stat} \rangle \text{'erro' , ';' while ...})$

temos então a configuração final após a recuperação.

Cabe finalmente destacar que a recuperação implícita em analisadores determinísticos é sempre vista como complementação de algum outro método de recuperação, já que só ela não garante que todos os erros sejam tratados de maneira satisfatória.

A recuperação por gramáticas mutacionais

Gramáticas mutacionais são extensões de gramáticas obtidas adicionando "mutações", que são modificações das produções originais, também chamadas de produções erro. A principal diferença entre este método, desenvolvido por Krazwyck [32] e os outros métodos de recuperação por produções erro reside na sistematização da geração destas. Desta maneira, são fornecidos critérios objetivos de extensão das gramáticas, critérios estes que possibilitam cobrir determinadas classes de erro e obter, com a extensão, gramáticas LR.

Os erros recuperados pelas gramáticas mutacionais são erros de omissão e troca de determinadas classes de símbolos, que definiremos a seguir.

Seja a $G=(V_n, V_t, P, S)$ uma GLC

def 5.1 Um símbolo $s \in V_t$ é um **separador** sse $\exists \alpha A s B \beta \in FS(G)$
t.q.:

- a) $S \xRightarrow{*} (rm) \alpha A s B \beta$,
- b) $\alpha, \beta \in V^*$
- c) $A, B \in V_n$.

def 5.2 Dado $\alpha \in FS(G)$, $x, y \in V_t$,

- a) $Oc(x, \alpha)$ = número de ocorrências de x em α
- b) $R(x, y, \alpha) = Oc(x, \alpha) - Oc(y, \alpha)$

def 5.2 Um par (s_1, s_2) de símbolos terminais constitui uma **estrutura de parentização** de alcance 2 em $L(G)$ sse

- a) $s_1 \neq s_2$
- b) $\forall \alpha \in L(G)$, $R(s_1, s_2, \alpha) = 0$
- c) $\forall \beta \in Prefixo(L(G))$, $R(s_1, s_2, \beta) \geq 0$

Uma estrutura de parentização de alcance dois é uma dupla de terminais que sempre ocorrem em dupla em uma sentença de uma gramática. como exemplo podemos citar os parênteses $('(' \text{ e } ')'$).

def 5.3 $\forall a \in V_t$, chamamos $A \in V_n$ de **não terminal mais próximo** de a sse

- a) $A \xRightarrow{*} \alpha_1 a \alpha_2$, $\alpha_1, \alpha_2 \in V^*$

- b) $\forall B \in V_n - \{A\}$ t.q. $B \xRightarrow{*} \beta_1 a \beta_2$,
 se $A \xRightarrow{*} \gamma_1 B \gamma_2$ então $B \xRightarrow{*} \delta_1 A \delta_2$
 $(\beta_1, \beta_2, \gamma_1, \gamma_2, \delta_1, \delta_2, \in V^*)$

Um não terminal mais próximo de um símbolo terminal é um não terminal que gera uma cadeia com o referido terminal sem a derivação de nenhum outro símbolo não terminal.

def 5.4 $\forall a \in V_t$,

$M_a = \{A / A \text{ é não terminal mais próximo de } a\}$

def 5.6 Dois terminais $a, b \in V_t$ são sintaticamente indistiguíveis sse:

- a) $\exists \alpha_1, \alpha_2, \beta_1, \beta_2 \in V^*, \exists A, B \in V_n$ t.q.

$$\alpha_1 A a \beta_2, \beta_1 A a \beta_2 \in FS(G)$$

- b) $M_a = M_b$

def 5.7 Delimitador é um separador para o qual não existe outro separador sintaticamente indistinguível.

Segundo Krawzyck, os erros mais comuns em programação são aqueles que ocorrem em separadores e em símbolos de extruturas de parentização. Tendo isto em vista, as mutações, ou produções erro, serão construídas tendo em vista erros nestes tipos de símbolos. Porém nem todos os tipos de erro em separadores e extruturas de parentização serão cobertos pelas gramáticas mutacionais.

Para evitar um aumento excessivo do número de produções, a criação de mutações que prevejam os erros nas classes de símbolos acima definidos será restita aos casos considerados "práticos" (não são dadas maiores explicações), que são omissão e troca de delimitadores, omissão de símbolos de extruturas de parentização e omissão de separadores que não são delimitadores.

Algoritmo Constrói Mutações

entrada: P - conjunto de produções da gramática original

Sep - conjunto dos símbolos separadores

Par - conjunto dos símbolos de extruturas de parentização
 Del - conjunto dos símbolos delimitadores
 saída: P' - conjunto de produções da gramática mutacional.
 com1

$P' := P;$

$P_d := P;$

enquanto $P_d \neq \emptyset$ faça

com2

$PAUX := \emptyset;$

para todo $p = (A \rightarrow a_1 \dots a_n) \in P_d$ faça

para i de 1 até n faça

se $a_i \in \text{Sep} \cup \text{Par}$

então

$PAUX := PAUX \cup \{(A \rightarrow a_1 \dots a_{i-1} a_{i+1} \dots a_n)\};$

$P' := P' \cup PAUX;$

$P_d := PAUX$

fim2;

$P_r := \emptyset;$

para cada $p = (A \rightarrow a_1 \dots a_n) \in P'$ faça

para i de 1 até n faça

se $a_i \in \text{Del}$

então

para cada $s \in \text{Del} - \{a_i\}$ faça

(*2*) $P_r := P_r \cup \{(A \rightarrow a_1 \dots a_{i-1} s a_{i+1} \dots a_n 0)\};$

$P' := P' \cup P_r$

fim1.

Para garantir que não sejam introduzidas ambiguidades na gramática mutacional torna-se necessário retirar algumas das mutações. São dados três critérios de suficiência que garantem que uma mutação não introduz ambiguidades em uma dada gramática:

1) Seja $p = A \rightarrow \alpha_1 s_1 \alpha_2 \in P$, a mutação $A \rightarrow \alpha_1 s_2 \alpha_2$ pode ser introduzida na gramática G sem gerar ambiguidades se não existe um estado q não analisador $LR(1)$ da gramática com $\text{Atitude}(q, s_1) \neq '?'$ e $\text{Atitude}(q, s_2) \neq '?'$.

2) Sejam $p = A \rightarrow \alpha_1 s_1 \alpha_2 \in P$ e $r = B \rightarrow \beta_1 s_2 \beta_2 \in P$, a mutação $p = A \rightarrow \alpha_1 s_2 \alpha_2$ pode ser introduzida na gramática G sem gerar ambiguidades se $\text{Follow}(s_1) \neq \text{Follow}(s_2)$.

3) Seja $p = A \rightarrow \alpha_1 s_1 \alpha_2 \in P$ então a mutação $A \rightarrow \alpha_1 \alpha_2$ pode ser introduzida na gramática G sem gerar ambiguidades se :

$$\forall a \text{ t.q. } s_1 \in \text{Follow}(a), \text{Follow}(a) \neq \text{Follow}(s_1)$$

Estes critérios, se aplicados como condição de inclusão de mutações garantem que a gramática mutacional permaneça LR(1). Resta porém um problema. A maioria das mutações relativas a símbolos de extruturas de parentização introduz ambiguidades na gramática. Para resolver este problema Krawzyck define um tipo de ambiguidade "aceitável" que ele denomina "ambiguidade local", que se caracteriza por gerar conflitos que, quando resolvidos escolhendo-se a atitude determinada pela produção da gramática original, não geram erros espúrios .

A Recuperação de Contexto Global

Este método, segundo seus autores Pai e Kieburtz [13], tem como objetivo servir de alternativa a uma recuperação local simples, onde seriam tentadas a inserção a eliminação e a troca de 1 símbolo no PDE, sendo o método acionado apenas caso estas três tentativas falhassem.

A Recuperação de Contexto Global se processa em duas fases distintas: Na primeira desmonta-se a cadeia de entrada vai sendo consumida até que se encontre um símbolo pertencente a um conjunto pré determinado, denominado **conjunto de símbolos confiáveis** ("fiducial symbols"). Na segunda fase, a pilha é reconfigurada através de eliminação de elementos e inserção de outros resultantes da "expansão" de elementos relativos a estados cujo símbolo de entrada é um não terminal. Esta reconfiguração visa vazer com que o símbolo encontrado seja aceito pelo analisador, consumindo o menor número possível de elementos da pilha.

A escolha do conjunto de símbolos confiáveis, como podemos ver, é crucial. Para isso, são sugeridos dois critérios de escolha. Um deles, mais difícil de ser satisfeito, garante recuperação com ausência de erros expúrios; o outro, mais fácil de ser satisfeito, não.

def 5.8 Uma cadeia $tw \in V_t^*$, $t \in V_t$, é chamada **t-sufixo terminal** de uma linguagem L se $\exists \alpha$ t.q. $\alpha tw \in L$

def 5.9 Um símbolo $t \in V_t$ é um **símbolo fortemente confiável** da gramática G se $\exists \beta \in V^*$, β sufixo de uma torma sentencial de G , tal que para qualquer t-sufixo terminal ω de $L(G)$ temos que: $\beta \xrightarrow{*} tw$.

Em outras palavras, um símbolo é dito fortemente confiável se todos os sufixos de sentenças da linguagem iniciados por ele podem ser derivados a partir de um único sufixo de uma forma sentencial da gramática.

Na recuperação de erros, ao ajustamos as pilhas para que o analisador aceite um símbolo fortemente confiável, estamos garantindo que a recuperação adotada não gerará erros expúrios se o analisador for "direcionado" para o reconhecimento da cadeia β relativa ao símbolo fortemente confiável encontrado. Torna-se então necessário um método para determinar os símbolos fortemente confiáveis de uma gramática.

def 5.10 Dado $G = (V_n, V_t, P, S)$, um símbolo $s \in V_t$ tem unicidade forte em frases de G (denota-se UFF) se $L(G) = \emptyset$ ou:

- 1) Existe no máximo uma produção $p \in P$ t.q. $p: A \rightarrow \alpha_1 s \alpha_2, \alpha_1, \alpha_2 \in V^+$
e
- 2) s ocorre apenas uma vez em $LD(p)$
e
- 3) Não existe derivação $A \xRightarrow{*} \gamma A \delta$ onde $\delta \neq \lambda$
e
- 4) A tem UFF em $G(A) = (V_n - \{A\}, V_t \cup \{A\}, P', S)$, onde P' são as produções de P com lado esquerdo diferente de A .

Teorema 5.1

Se S tem UFF em $G = (V_n, V_t, P, S)$, então s é um símbolo fortemente confiável.

Dem (informal): Como s tem UFF, então s pertence ao lado direito de uma única produção; chamemos esta produção de $p = A \rightarrow \alpha_1 s \alpha_2$. Como a recursão em A só pode ocorrer à esquerda de s e como o mesmo vale para A , então qualquer s -sufixo terminal precisa ser derivado a partir da cadeia $\beta = s \alpha_2$.

A restrição de unicidade que UFF impõe à produção $A \rightarrow \alpha_1 s \alpha_2$, garante que, se tomarmos como conjunto de símbolos confiáveis aqueles que possuem UFF, não serão detectados erros expúrios após a recuperação pelo método de recuperação de contexto global. Isto porque todo o s -sufixo terminal de $L(G)$ deve ser derivado a partir de $s \alpha_2$, e a unicidade imposta a s garante que, após seu reconhecimento, o único estado possível para o

analisador é $\{(A \rightarrow \alpha_1 s \alpha_2; u)\}$, garantindo que a análise do restante da cadeia de entrada será bem sucedida, se não existirem mais erros.

Infelizmente a restrição UFF é muito forte. Os autores afirmam que para linguagens de programação reais existem muito poucos símbolos com UFF. Quando o conjunto de símbolos confiáveis é pequeno, aumenta a chance de que, na recuperação do erro, trechos grandes da cadeia de entrada sejam eliminados antes que seja encontrado um símbolo adequado.

Para minimizar este problema é sugerido um relaxamento nas condições de seleção dos símbolos confiáveis.

def 5.11

Dada uma gramática $G = (V_n, V_t, P, S)$, um símbolo $s \in V_t$ tem **unicidade fraca em frases de $L(g)$ (Uff)** se $L(G) = \emptyset$ ou:

1) existe no máximo uma produção p de P t.q.:

$$p = (A \rightarrow \alpha_1 s \alpha_2), \alpha_1, \alpha_2 \in V^*$$

2) s ocorre apenas uma vez em $LD(p)$

3) A tem Uff em $G(A)$.

A unicidade fraca elimina a exigência de ausência de recursão de A à direita de s . Desta maneira, se definirmos o conjunto de símbolos confiáveis como aqueles que tem Uff, aumentamos o seu espectro. Porém pagamos um preço: não mais podemos garantir a ausência de erros expúrios após a recuperação, pois a recursão de A à esquerda de s pode determinar a inexistência de uma cadeia β que gera todos os s -sufixos terminais da linguagem.

Ex. Seja G a gramática definida pelas produções:

$$S \rightarrow Z\#$$

$$Z \rightarrow aA$$

$$A \rightarrow AsP$$

$$A \rightarrow r$$

$$P \rightarrow Pq$$

$$P \rightarrow kc$$

s tem Uff em G . Temos que a única cadeia β t.q. $\beta \xrightarrow{*} s\alpha, \alpha \in V_t^*$, é a cadeia $\beta = sP$. Além disso α será da forma $kcqqq\dots q$. Seja porém a seguinte cadeia de derivação :

$$S \Rightarrow aA \Rightarrow aAsP \Rightarrow aAsPsP \Rightarrow aAsPskc \Rightarrow aAskcskc \Rightarrow arskcskc$$

como podemos ver 'skcskc' é um s-sufixo terminal em $L(G)$, porém não existe uma cadeia de símbolos β' t.q. $\beta' \xrightarrow{*} skcskc$.

A partir das definições 5.10 e 5.11 podemos construir algoritmos para calcular se um símbolo possui UFF ou Uff em $L(G)$, dada uma gramática $G = (V_n, V_t, P, S)$.

Com o conjunto de símbolos confiáveis definido, o método de recuperação de contexto global pode ser utilizado. Como já dissemos, numa primeira fase são consumidos (i.é. eliminados) símbolos da cadeia de entrada até que seja encontrado um símbolo pertencente ao conjunto dos símbolos confiáveis. Em seguida a pilha é reorganizada de maneira a que o símbolo encontrado seja aceito pelo analisador. Para isto é consultado o estado do topo da pilha do analisador, que chamaremos de q : se $Atitude(q, First(\beta)) \neq \text{'erro'}$ então o algoritmo termina.; caso contrário o estado q é eliminado e, se $Sim(q) \in V_n$ e se

$$Sim(q) \Rightarrow (rm)\gamma \xrightarrow{*} (rm)\mu s \delta, \gamma \in V^*, \mu, \delta \in V_t^*,$$

então são inseridos na pilha os estados relativos à análise da cadeia γ , repetindo-se em seguida todo o processo para a nova configuração da pilha. Vejamos o algoritmo:

Algoritmo Rec_Contexto_Global

entrada: (α, β) - configuração do analisador

Sc- conjunto dos símbolos confiáveis

Prod_Sc-tabela que indica, para cada dupla não terminal/símbolo confiável

qual a produção do não terminal

que deriva o símbolo confiável, se existir (esta tabela pode ser construída juntamente com o conjunto dos símbolos confiáveis).

saída: (α, β) - nova configuração do analisador

com1

enquanto $Prim(\beta) \notin Sc$ faça

```

     $\beta := \text{Final}(\beta); (*\text{procura primeiro símb. confiável}*)$ 
     $\text{Simbc} := \text{Prim}(\beta); (*\text{Simbc recebe o símbolo confiável encontrado}*)$ 
    enquanto  $\text{Atitude}(\text{Ult}(\alpha), \text{Simbc}) = \text{erro}$  faça
      com2
         $\text{Sim} := \text{Simb}(\text{Ult}(\alpha)); (*\text{Sim recebe símbolo do topo da p.sint.}*)$ 
         $\alpha := \text{Início}(\alpha); (*\text{desempilha último estado}*)$ 
        se  $\text{Sim} \in V_n$  e  $\text{Prod\_Sc}(\text{Sim}, \text{Simbc}) \neq ''?'$ 
           $(* \text{se é o não terminal que deriva Simbc} *)$ 
          então
            com3
               $(* \text{acrescenta lado direito da}$ 
                 $\text{produção à cadeia de entrada} *)$ 
               $p := \text{Prod\_sc}(\text{Sim}, \text{Simc});$ 
               $\beta := \text{LD}(p) \text{ cat } \beta;$ 
               $(* \text{acrescenta estados da análise de LD}(p) \text{ à pilha} *)$ 
               $\text{Análise\_limitada}((\alpha, \beta), / \text{LD}(p) /, \text{Sucesso});$ 
            fim3;
          se  $\alpha = \lambda$   $(* \text{se a pilha ficou vazia} *)$ 
             $(* \text{coloca o lado direito da prod inicial da gramática} *)$ 
            então  $\text{Análise\_limitada}((\alpha, Z \text{ cat } \beta), 1, \text{Sucesso})$ 
          fim2;
        fim1.

```


Capítulo 6

Final

Um algoritmo de recuperação de erros sintáticos, como já dissemos, tem como objetivo reconfigurar um analisador sintático após a ocorrência de um erro de maneira que a análise do restante da cadeia de entrada possa ser executada.

Procuramos, ao longo deste trabalho, descrever, na medida do possível, metodologias e princípios de recuperação que se aplicassem a qualquer analisador do tipo LR, independente da gramática que estivesse sendo analisada. Porém resta a pergunta: como deve ser um esquema de recuperação de erros na prática?

6.1 A recuperação em analisadores de produção

O leitor pode, após a leitura, ter a impressão errônea de que qualquer mecanismo de recuperação que não seja completamente objetivo, inde-

pendendo portanto de especificações do implementador, seja inferior. Isso não é de maneira nenhuma verdade. É desejável que os compiladores de produção (aqueles que são utilizados intensamente) possuam um esquema de recuperação de erros que possa ser "sintonizado" às particularidades da gramática utilizada e do usuário do sistema, com vistas a incrementar a sua capacidade de executar boas recuperações.

Porém, não se pode igualmente adotar a abordagem oposta, ou seja, executar a recuperação somente através de rotinas desenvolvidas para a correção de erros específicos de determinada linguagem. É justamente este tipo de política que torna a recuperação de erros em alguns compiladores comerciais desastrosa. Isto acontece porque a qualidade da recuperação fica completamente dependente da capacidade de previsão do implementador. Ao nosso ver, um bom esquema de recuperação de erro em analisadores sintáticos de produção deve aliar métodos genéricos de recuperação a dispositivos que garantam a possibilidade de "sintonizar" o esquema de recuperação às peculiaridades da gramática utilizada e do usuário deste analisador. Como exemplo de esquemas "sintonizáveis" podemos citar os sistemas com custos associados às operações de edição ([10],[24],[25],[28]) e as listas de proibições([19]). Não podemos deixar de citar também o esquema de recuperação desenvolvido por Susan Graham e Stephen Rhodes, publicado em 1979 [13] do qual faremos agora um breve resumo

O método Prático de Graham e Rhodes

O objetivo do desenvolvimento deste método era a implementação de estratégias de erro que podiam ser tratadas eficientemente. O resultado foi um algoritmo com duas fases:

Na primeira fase são tentadas operações simples de edição (inserção, eliminação e troca de símbolos) no símbolo de entrada o no símbolo do topo da pilha sintática, se este for terminal. A cada solução é atribuído um custo, baseado no custo da operação de edição e no comportamento de uma análise sintática limitada pelo número de símbolos, ensaiada pelo algoritmo de recuperação. O custo das operações de edição é fornecido pelo implementador, como em [28],[24],[10]. Este custo é multiplicado por um valor que decresce conforme aumenta a quantidade de símbolos analisados com sucesso. O número de símbolos analisados na atribuição de custo à recuperação também é determinado pelo implementador, bem como os valores de multiplicação.

Uma vez calculadas as recuperações da primeira fase escolhe-se aquela com menor custo. Se porém todas estas recuperações tiverem um custo associado que ultrapasse um determinado limite máximo (também determinado pelo implementador), descarta-se a recuperação escolhida e aciona-se a segunda fase.

A segunda fase executa o mesmo algoritmo adotado pelo sistema YACC [16]: inicialmente é inserido no início da cadeia de entrada um símbolo terminal especial, 'erro'; após esta inserção os elementos da pilha são retirados até que o estado do topo da pilha do analisador aceite o terminal 'erro' como símbolo de entrada. Neste momento é executado o avanço sobre o símbolo em questão. Finalmente o analisador passa a consumir símbolos da cadeia de entrada até encontrar um que seja aceito pelo novo estado do topo da pilha do analisador.

Este símbolo terminal 'erro' não pertence à gramática original. O implementador deve fornecer juntamente com as produções da gramática original, novas produções com este terminal indicando os locais mais prováveis de ocorrência de erro.

Além deste esquema básico o método de Graham e Rhodes conta também com alguns dispositivos adicionais que visam melhorar a qualidade da recuperação:

1) Informações semânticas são consideradas para calcular o custo das recuperações da primeira fase visando eliminar alternativas sintaticamente corretas mas sem sentido. Um exemplo disso é a recuperação do trecho:

....A := B C + D * E * 150};.....

aqui, mesmo se a análise limitada se restringir a menos de 9 símbolos, o reconhecimento de B como um identificador de vetor elimina possibilidades de correção como a inserção de operadores aritméticos.

2) Erros semânticos também são tratados pelo algoritmo em sua primeira fase (como por exemplo um identificador de função seguido de '[' ao invés de '(').

3) Restrições sintáticas da gramática são eventualmente substituídas por restrições semânticas, quando isto simplificar a recuperação de erros, como no caso da linguagem PASCAL; onde declarações de rótulos, constantes, tipos e variáveis devem ser feitas em determinada ordem. Neste

caso poder-se-ia utilizar uma gramática que aceitasse as declarações em qualquer ordem, impondo esta ordem nas restrições semânticas.

4) Caso o analisador sintático ao qual o esquema de recuperação vai ser adicionado tenha suas tabelas comprimidas utilizando a técnica da redução incondicional (vide cap2), os autores sugerem que não sejam comprimidos os estados onde existe algum item que determine avanço quando o símbolo de entrada for o terminal 'erro'. Desta maneira, nos locais onde a ocorrência de erros é mais frequente, não são efetuadas reduções sem que se tenha certeza de que o símbolo de entrada é válido. com isto evitamos que uma condensação excessiva do contexto ao redor do PDE "mascare" possibilidades de correção, como já vimos na seção 4.3.

6.2 A recuperação em sistemas de geração de analisadores

Uma das grandes vantagens de analisadores do tipo LR é impor menos restrições nas gramáticas durante o processo de geração automática de analisadores sintáticos. Desta maneira a utilização de sistemas de geração de analisadores é facilitada. O exemplo mais conhecido de um sistema destes é o do YACC da Bell Laboratories.

Nestes sistemas de geração de analisadores, também é importante a presença, nos analisadores gerados, de um bom esquema de recuperação de erro. Porém neste caso nem sempre é interessante que este sistema de recuperação dependa de uma "sintonização" por parte do implementador. Ao contrário: a não ser que o analisador gerado pelo sistema seja um analisador sintático de produção (a ser utilizado intensamente), é preferível poupar o usuáriedeste trabalho tedioso, simplificando a utilização do sistema. Desta maneira, torna-se mais recomendável a implementação de um sistema genérico de recuperação, que independe da interferência do usuário-implementador.

Não existe infelizmente uma receita ideal nem um critério único e objetivo que nos permita uma real comparação entre os vários métodos, determinando qual é o melhor. Porém, é nossa opinião que existem dois bons caminhos:

O primeiro deles é a implementação de um esquema de recuperação em duas fases: na primeira tenta-se a aplicação de algum método regional, que verifique o efeito de pequenos conjuntos de operações de edição em

um contexto ao redor do PDE, contexto este delimitado de preferência sintaticamente (ex: até o fim de uma produção). Caso nenhum destes conjuntos propicie uma correção sem erros no contexto considerado, passa-se a segunda fase. Na segunda fase adotar-se-ia um algoritmo nos moldes do panic-mode, visando garantir que alguma recuperação sempre seja efetuada.

O segundo caminho seria a adoção de um esquema de recuperação não corretivo, como sugere Richter (vide 4.3.2). Este esquema possui duas grandes vantagens: em primeiro lugar é rápido, já que não aumenta em nada o tempo de análise, esteja o programa correto ou não; em segundo lugar porque a ocorrência de erros expúrios é nula. Pela sua simplicidade de funcionamento e devido aos fatores citados acima, torna-se a nosso ver, a opção mais atraente. Porém não deve ser adotada em alguns casos que vale a pena citar:

- em sistemas com problemas de espaço de memória, pois este tipo de recuperação envolve um "overhead" considerável de memória, já que um novo conjunto de tabelas do analisador é necessário.
- em sistemas onde os compiladores executem a geração de código mesmo para programas sintaticamente incorretos, mas com erros de pequena gravidade, já que neste esquema isto é impossível.

Anexo a este trabalho figura a listagem da implementação de um sistema de geração de compiladores que, dada uma gramática, gera para ela um analisador LALR(1), utilizando a técnica de compressão de tabelas descrita em 2.5. Neste analisador está implementado um algoritmo não corretivo de erros sintáticos, baseado no algoritmo de condensação de Pennello e DeRemmer (vide 4.3.2).

Bibliografia

- [1] J. Aho, LR-Parsing, *Computing Surveys* **6-2** (1974), 99-124.
- [2] R. Terada, "Linguagens Determinísticas", Dissertação de Mestrado, Univ. de São Paulo, SP (1975).
- [3] W.R. Lalonde, An "Efficient LALR Parser" Generator, *Tech. Report CSRG-2, Univ. of Toronto* (1971).
- [4] D.E. Knuth, On the Translation of Languages from Left to Right, *Information and Control* **8** (1965), 607-639.
- [5] F.H. De Remer, Simple LR(K) Grammars, *CACM* **14** (1971), 453-460.
- [6] J. Earley, An Efficient Context-free Parsing Algorithm, *CACM* **13-2** (1970), 94-102.
- [7] J. Röhrich, Methods for the Automatic Construction of Error Correcting Parsers, *ACTA Informatica* **13** (1980) 115-139.
- [8] C.N. Fischer, B.A. Dion, J. Mauney, A Locally Least-cost LR-Error Corrector, *Comp. Scien. Dept. Report - Univ. of Wisconsin - Madison* **363** (1979).

- [9] Anderson, Eve, Horning, Efficient LR(1) Parsers, *ACTA Informatica* 2 (1973) 10-32.
- [10] S.L. Graham, S.P. Rhodes, Practical Syntactic Error Recovery, *Comm. ACM* 18 (1975) 639-650.
- [11] M.D. Mikunas, J.A. Modry, Automatic Error Recovery for LR Parsers, *Comm. ACM* 21-6 (1978) 459-465.
- [12] S. Graham, C. Hayley, W.N. Joy, Practical LR Error Recovery, *Sigplan Notices* 14-8 (1979), 168-175.
- [13] A.B. Pai, R.B. Kieburtz, Global Context Recovery-A New Strategy for Parser Recovery from Syntax Errors, *Sigplan Notices* 14-8 (1979), 168-175.
- [14] K. Groening, Combined Actions to Reduce LR-Parsetables, *Sigplan Notices* 19-3 (1984), 42-45.
- [15] C.N. Fischer, J. Mauney, On the Role of Error Productions, *Computer Languages* 5 (1980), 131-139.
- [16] S.C. Johnson, YACC-Yet An Other Compiler Compiler, *Bell Laboratories, Murray Hill, Technical Report, NJ-07974*.
- [18] C. Wethenel, Why Automatic Error Correctors Fail, *Computer Languages* 2 (1977), 179-186.
- [19] M. Burke, G.A. Fisher, A Practical Method for Syntactic Error Diagnosis and Recovery, *Sigplan Notices* 17-6 (1982), 67-78.
- [20] G.D. Ripley, F.C. Druseikis, A Statistical Analysis of Syntax Errors, *Computer Languages* 3 (1978), 227-240.
- [21] A.V. Aho, T.G. Peterson, A Minimum Distance Error-Correcting Parser for Context-Free Languages, *SIAM Journal on Computing* 1-4 (1972), 305-312.
- [22] S. Feycock, P. Lazarus, Syntax-Directed Correction of Syntax Errors, *Software-Practice and Experiences* 6 (1976), 207-219.
- [23] E.B. James, D.P. Partridge, Adaptive Correction of Program Statements, *CACM* 16-1 (1973), 27-37.

- [24] K.C. Tai, Syntatic Error Correction in Programming Languages, *IEEE Trans. on Software Eng.* 4-5 (1978), 305-312.
- [25] S. Sippu, E. Soisalon-Soininen, A Syntax Error-Handling Technique and its Experimental Analysis, *ACM Toplas* 5-4 (1983), 656-679.
- [26] J.P. Lévy, Automatic Correction of Syntax-Errors in Programming Languages, *Acta Informatica* 4 (1975), 271-292.
- [27] C. Burgen, L. James, An Indexed Bibliography for LR Grammars and Parsers, *Sigplan Notices* 16-8 (1981), 14-26.
- [28] R.A. Wagner, M.J. Fischer, The String-to-String Correction Problem, *JACM* 21 (1974), 168-173.
- [29] R.A. Wagner, M.J. Fischer, An Extension of the String-to-String Correction Problem, *JACM* 22-2 (1975), 177-183.
- [30] H. Richter, Noncorrecting Syntax Error Recovery, *ACM Toplas* 7-3 (1985), 478-489.
- [31] S.O. Anderson, R.C. Backhouse, Locally Least Cost Error Recovery in Earley's Algorithm, *ACM Toplas* 3-3 (1981), 318-347.
- [32] T. Krawzyck, Error Correction by Mutational Grammars, *Info. Processing Letters* 11-1 (1980), 9-15.
- [33] E. Soisalon-Soininen, Inessential Error Entries and their use in LR Parser Optimization, *ACM Toplas* 4-2 (1982), 179-195.
- [34] C.N. Fischer, J. Mauney, A Forward Move Algorithm for LL and LR Parsers, *ACM Toplas* 4-4 (1982), 615-649.
- [35] T.J. Pennelo, F. De Remer, A Forward Move Algorithm for LR Error Recovery, *Conference Record of the 5th Ann. ACM Symposium on Principles of Prog. Languages* (1978), 241-254.
- [36] F. De Remer, T.J. Pennelo, Efficient Computation of LALR(1) Look-Ahead Sets, *ACM Toplas* 4-4 (1982), 615-649.
- [37] Park, Choe, Chang, A New Analysis of LALR Formalisms, *ACM Toplas* 7-1 (1985), 155-175.

- [38] O.L. Madsen, B.B. Kristensen, LR Parsing on Extended C.F. Grammars, *Acta Informatica* 7 (1976), 61-73.
- [39] W.R. Lalonde, Regular Right Part Grammars and their Parsers, *CACM* 20-10 (1977), 731-740.
- [40] W.R. Lalonde, Constructing LR Parsers for R.R.P. Grammars, *Acta Informatica* 11 (1979), 177-193.
- [41] S. Heilbrunner, On the Definition of ELR(K) and ELL(K) Grammars, *Acta Informatica* 11 (1979), 169-176.
- [42] A. Celentario, An LR Parsing Technique for E.C.F. Grammars, *Computer Languages* 6 (1981), 95-107.
- [43] P.W. Purdon, C.A. Brown, Parsing Extended LR(K) Grammar, *Acta Informatica* 15 (1981), 115-127.
- [44] N.P. Chapman, LALR(1,1) Parser Generation for Regular Right Past Grammars, *Acta Informatica* 21 (1984), 25-45.
- [45] J. Nakata, M. Sassa, Generation of Efficient LALR Parser for Regular Right Past Grammars, *Acta Informatica* 23 (1986), 149-162.
- [46] N.Wirth, PL-360 A Programming Language for the 360 Computer, *JACM* 15-1 (1968), 37-64.
- [47] V.W. Setzer, I.H. de Melo, A Construção de um Compilador, *Escola de Computação* (1981).
- [48] Fischer, Milton, Quering, Error Correction using only insertions, *Acta Informatica* 13 (1980), 141-154.
- [49] K.Tai Fault Tolerant String Processing by Pattern Mapping, *Proc.ACM Annu.Conf.* (1977), 336-342.
- [50] M.Hammond, V.J.Rayward-Smith, A Survey of Syntatic Error Recovery and Repair, *Computer Languages* 9-1 (1984), 51-67.
- [51] M.A. Harrison, "Introduction to Formal Language Theory", Addison Wesley (1978).
- [52] M.A.S. Bigonha, SIC - Sistema de Implantação de Compiladores, Dissertação de Mestrado-UFMG (1985).

- [53] I. Simon, *Linguagens Formais e Autômatos, Segunda Escola de Computação* (1981).
- [54] F. de Remer, *Practical Translations for LR(K) Languages*, Ph.D. Thesis, MIT, (1969).
- [55] A.V. Aho, J.D. Ullman, *The Theory of Parsing Translating and Compiling V.L.-Parsing*, Prentice Hall, (1972).
- [56] K.C. Tai, *Sintatic Error Correction in Programming Languages*, Ph.D. Thesis, Cornell University, (1977).
- [57] J.P. Levy, *Error Correction and Recovery of Syntax Errors in Programming Languages*, Ph.D. Thesis, Cornell University, (1971).
- [58] A.V. Aho, J.D. Ullman, A Technique for Speeding up LR(K) Parsers, *SIAM J. of Computing* 2-2 (1973) 166-127.
- [59] D. Pager, *On Eliminating Unit Productions from LR(K) Parsers*, Tech. Report Honolulu, Hawaii (1974).
- [60] A.V. Aho, J.D. Ullman, Optimizations of LR(K) Parsers, *J. of Computer and System Sci.* 6-6 (1972) 573-602.
- [61] T. Anderson, *Syntatic Analysis of LR(K) Languages*, Ph.D. Thesis, Univ. Newcastle-upon-Tyne, (1972).
- [62] M.L. Jolliat, *On the Reduced Matrix Representation of LR(K) Parser Tables*, Ph.D. Thesis, Univ. Toronto, (1973).
- [63] D. Pager, A Solution to an Open Problem by Knuth, *Information and Control* 17 (1970) 462-473.
- [64] T. Anderson, Efficient LR(1) Parsers, *Acta Informatica* 2 (1973) 12-39.
- [65] A. Demers, Elimination of Single Productions and Marging Non Terminal Symbols of LR(1) Grammars, Tech. Report Univ. of Princeton (1973).
- [66] J. Ciesinger, A Bibliography on Error Handling, *SIGPLAN Notices* 14-1 (1979) 16-26.
- [67] G. Lyon, Syntax Directed Least Cost Error Analysis for Context Free Languages- A Practical Approach, *CACM* 17-1 (1974) 3-14.

APÊNDICE A

$\langle S \rangle \rightarrow \langle \text{program} \rangle .$
 $\langle \text{program} \rangle \rightarrow \text{'prog' id} \langle \text{vardecl} \rangle \text{'begin'} \langle \text{stlist} \rangle \text{'end'}$
 $\langle \text{vardecl} \rangle \rightarrow \text{'var'} \langle \text{declist} \rangle$
 $\langle \text{declist} \rangle \rightarrow \langle \text{idlist} \rangle \langle \text{type} \rangle ; \langle \text{declist} \rangle$
 $\langle \text{declist} \rangle \rightarrow \langle \text{numlist} \rangle \langle \text{latype} \rangle ; \langle \text{declist} \rangle$
 $\langle \text{idlist} \rangle \rightarrow \text{'id'}$
 $\langle \text{idlist} \rangle \rightarrow \text{'id'}, \langle \text{idlist} \rangle$
 $\langle \text{numlist} \rangle \rightarrow \text{num}$
 $\langle \text{numlist} \rangle \rightarrow \text{num} , \langle \text{numlist} \rangle$
 $\langle \text{type} \rangle \rightarrow \langle \text{sitype} \rangle$
 $\langle \text{type} \rangle \rightarrow \langle \text{arraytype} \rangle$
 $\langle \text{arraytype} \rangle \rightarrow \text{array} [\text{num} .. \text{num}] \text{ of } \langle \text{sitype} \rangle$
 $\langle \text{latype} \rangle \rightarrow \text{: label}$
 $\langle \text{sitype} \rangle \rightarrow \text{integer}$
 $\langle \text{sitype} \rangle \rightarrow \text{real}$
 $\langle \text{stlist} \rangle \rightarrow \langle \text{stat} \rangle$
 $\langle \text{stlist} \rangle \rightarrow \langle \text{stat} \rangle ; \langle \text{stlist} \rangle$
 $\langle \text{stat} \rangle \rightarrow \langle \text{vaiden} \rangle \text{ := } \langle \text{expression} \rangle$
 $\langle \text{stat} \rangle \rightarrow \text{if } \langle \text{expression} \rangle \text{ then } \langle \text{stat} \rangle$
 $\langle \text{stat} \rangle \rightarrow \text{if } \langle \text{expression} \rangle \text{ then } \langle \text{stat} \rangle \text{ else } \langle \text{stat} \rangle$
 $\langle \text{stat} \rangle \rightarrow \text{while } \langle \text{expression} \rangle \text{ do } \langle \text{stat} \rangle$
 $\langle \text{stat} \rangle \rightarrow \text{for } \langle \text{vaiden} \rangle \text{ := } \langle \text{vaiden} \rangle \text{ to } \langle \text{expression} \rangle \text{ do } \langle \text{stat} \rangle$
 $\langle \text{stat} \rangle \rightarrow \text{goto } \langle \text{label} \rangle$
 $\langle \text{stat} \rangle \rightarrow \langle \text{label} \rangle : \langle \text{stat} \rangle$
 $\langle \text{stat} \rangle \rightarrow \text{begin } \langle \text{stlist} \rangle \text{ end}$
 $\langle \text{vaiden} \rangle \rightarrow \text{id}$
 $\langle \text{vaiden} \rangle \rightarrow \text{id} [\text{num} , \text{num}]$
 $\langle \text{expression} \rangle \rightarrow \langle \text{siexpr} \rangle$
 $\langle \text{expression} \rangle \rightarrow \langle \text{siexpr} \rangle > \langle \text{siexpr} \rangle$
 $\langle \text{expression} \rangle \rightarrow \langle \text{siexpr} \rangle < \langle \text{siexpr} \rangle$
 $\langle \text{expression} \rangle \rightarrow \langle \text{siexpr} \rangle \leq \langle \text{siexpr} \rangle$
 $\langle \text{expression} \rangle \rightarrow \langle \text{siexpr} \rangle = \langle \text{siexpr} \rangle$
 $\langle \text{expression} \rangle \rightarrow \langle \text{siexpr} \rangle \neq \langle \text{siexpr} \rangle$
 $\langle \text{expression} \rangle \rightarrow \langle \text{siexpr} \rangle \diamond \langle \text{siexpr} \rangle$
 $\langle \text{expression} \rangle \rightarrow \langle \text{siexpr} \rangle = \langle \text{siexpr} \rangle$
 $\langle \text{siexpr} \rangle \rightarrow \langle \text{term} \rangle$

$\langle \text{siexpr} \rangle \rightarrow \langle \text{siexpr} \rangle + \langle \text{term} \rangle$

$\langle \text{siexpr} \rangle \rightarrow \langle \text{siexpr} \rangle - \langle \text{term} \rangle$

$\langle \text{siexpr} \rangle \rightarrow + \langle \text{term} \rangle$

$\langle \text{siexpr} \rangle \rightarrow - \langle \text{term} \rangle$

$\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle$

$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle$

$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle / \langle \text{factor} \rangle$

$\langle \text{factor} \rangle \rightarrow \langle \text{vaiden} \rangle$

$\langle \text{factor} \rangle \rightarrow \text{num}$

APENDICE B

TABELA DE ATITUDES

Estado	#	a	b	c	d	e	k
V(λ)	erro	avanca	avanca	erro	erro	avanca	erro
V(Z)	avanca	erro	erro	erro	erro	erro	erro
V(C)	erro	erro	erro	erro	erro	erro	avanca
V(a)	erro	erro	erro	avanca	erro	erro	erro
V(b)	erro	erro	erro	avanca	erro	erro	erro
V(e)	erro	erro	erro	erro	erro	erro	reduz8
V(Z#)	aceita	aceita	aceita	aceita	aceita	aceita	aceita
V(aA)	erro	erro	erro	erro	avanca	erro	erro
V(ac)	erro	erro	erro	avanca	reduz6	erro	erro
V(bA)	reduz7	erro	erro	erro	erro	erro	erro
V(bc)	erro	erro	erro	avanca	reduz6	erro	erro
V(Ck)	reduz2	erro	erro	erro	erro	erro	erro
V(aAd)	reduz3	erro	erro	erro	erro	erro	erro
V(acA)	erro	erro	erro	erro	reduz5	erro	erro
V(bAd)	reduz7	erro	erro	erro	erro	erro	erro
V(bAB)	reduz4	erro	erro	erro	erro	erro	erro

Os números indicados nas ações de redução identificam as produções conforme a numeração abaixo:

- 1 - $S \rightarrow Z\#$
- 2 - $Z \rightarrow Ck$
- 3 - $Z \rightarrow aAd$
- 4 - $Z \rightarrow bAB$
- 5 - $A \rightarrow cA$
- 6 - $A \rightarrow c$
- 7 - $B \rightarrow d$
- 8 - $C \rightarrow e$