

LEONARDO TEIXEIRA PASSOS

**GERADOR LALR COM SUPORTE A  
RESOLUÇÃO DE CONFLITOS**

Belo Horizonte, Minas Gerais  
Setembro de 2007



UNIVERSIDADE FEDERAL DE MINAS GERAIS  
INSTITUTO DE CIÊNCIAS EXATAS  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

## GERADOR LALR COM SUPORTE A RESOLUÇÃO DE CONFLITOS

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

LEONARDO TEIXEIRA PASSOS

Belo Horizonte, Minas Gerais  
Setembro de 2007



# Sumário

<b>1</b>	<b>Introdução</b>	<b>1</b>
1.1	Definição do Problema . . . . .	2
1.2	Solução Proposta . . . . .	4
1.3	Contribuições . . . . .	4
1.4	Organização da Dissertação . . . . .	5
<b>2</b>	<b>Ferramentas Visuais para Auxílio na Construção de Analisadores Sintáticos</b>	<b>7</b>
2.1	Ferramentas Educativas . . . . .	7
2.1.1	GYacc . . . . .	7
2.1.2	Visual Yacc . . . . .	9
2.1.3	Parser . . . . .	11
2.1.4	LRParse . . . . .	13
2.1.5	CUPV . . . . .	14
2.2	Ferramentas Comerciais . . . . .	18
2.2.1	AnaGram . . . . .	18
2.2.2	Visual Parse++ . . . . .	21
2.3	Avaliação das Ferramentas . . . . .	23
2.4	Conclusão . . . . .	25
<b>3</b>	<b>Análise Sintática LALR(<math>k</math>)</b>	<b>27</b>
3.1	Conceitos e Terminologia Adotada . . . . .	27
3.2	Classificação de Conflitos . . . . .	30
3.3	Fronteira entre Conflitos em LR e LALR . . . . .	34
3.4	Cômputo da Função $LA_k$ . . . . .	35
3.4.1	Algoritmo de DeRemer e Pennello ( $k = 1$ ) . . . . .	35
3.4.2	Algoritmo de Charles ( $k \geq 1$ ) . . . . .	45
3.5	Conclusão . . . . .	57

<b>4</b>	<b>Compactação de Tabelas</b>	<b>59</b>
4.1	Codificação . . . . .	60
4.2	Compactação por Vetor . . . . .	62
4.2.1	Método de Compactação Proposto por Aho (ACS) . . . . .	62
4.2.2	Método de Compactação Proposto por Bigonha (BCS) . . . . .	64
4.2.3	Método de Compactação <i>Row Displacement</i> (RDS) . . . . .	66
4.2.4	Método de Compactação por Distância Significativa (SDS) . . . . .	68
4.2.5	Método de Compactação <i>Row Column</i> (RCS) . . . . .	69
4.2.6	Método de Compactação por Supressão de Zero (SZS) . . . . .	70
4.2.7	Análise . . . . .	72
4.3	Compactação por Matriz . . . . .	74
4.3.1	Método de Compactação por Coloração de Grafos (GCS) . . . . .	74
4.3.2	Método de Compactação por Eliminação de Linhas (LES) . . . . .	76
4.3.3	Análise . . . . .	78
4.4	Compactação de Tabelas LALR- $k$ variável . . . . .	81
4.5	Conclusão . . . . .	82
<b>5</b>	<b>Ferramenta SAIDE</b>	<b>83</b>
5.1	Metodologia Proposta . . . . .	83
5.2	Ambiente da Ferramenta . . . . .	87
5.2.1	Entendimento . . . . .	89
5.2.2	Classificação . . . . .	89
5.2.3	Alteração da especificação . . . . .	91
5.2.4	Testes . . . . .	95
5.3	Arquitetura . . . . .	95
5.4	Geração do Analisador Sintático . . . . .	96
5.5	Algoritmos . . . . .	97
5.6	Conclusão . . . . .	117
<b>6</b>	<b>Avaliação</b>	<b>119</b>
6.1	Testes Caixa Preta . . . . .	119
6.2	Remoção Automática de Conflitos . . . . .	128
6.3	Estudo de Caso: Machina . . . . .	129
6.4	Conclusão . . . . .	134
<b>7</b>	<b>Conclusão</b>	<b>137</b>
<b>A</b>	<b>Diagrama de Atividades</b>	<b>141</b>

<b>B Tabelas <i>Action</i> e <i>Goto</i> da Gramática 6.10 (<math>k_{max} = 3</math>)</b>	<b>145</b>
<b>C Gramática de Machina (<math>k_{max} = 2</math>)</b>	<b>187</b>
<b>Referências Bibliográficas</b>	<b>223</b>





# Capítulo 1

## Introdução

Em 1965 Knuth [Knuth, 1965] apresentou a análise sintática  $LR(k)$ , uma técnica que processa a entrada da esquerda para a direita (*Left to Right*), inspecionando  $k$  tokens à frente de modo a decidir o não terminal correspondente a um fragmento (*handle*) de uma forma sentencial obtida até o momento.

Ao longo dos anos, essa técnica obteve grande atenção por parte dos pesquisadores da área de linguagens pelos seguintes motivos:

- permite o reconhecimento de todas as linguagens determinísticas, isto é, aquelas para as quais existe um autômato de pilha determinístico;
- dada uma especificação de uma gramática, pode-se construir automaticamente o analisador sintático correspondente;
- um analisador  $LR(k)$  aceita ou rejeita um *string* de entrada em um único escaneamento da esquerda para a direita, sem realizar qualquer retrocesso (*backtracking*);
- durante a execução de um analisador sintático  $LR(k)$ , detecta-se um erro sintático tão logo o mesmo ocorra.

Uma gramática livre de contexto é dita ser  $LR(k)$  se existe um analisador sintático  $LR(k)$  que a reconhece. Por sua vez, uma linguagem é  $LR(k)$  se puder ser definida por uma gramática  $LR(k)$ . Para utilização prática, os analisadores sintáticos  $LR(k)$  na forma canônica necessitam de uma grande quantidade de memória [Aho et al., 1986]. Como consequência, duas variações foram propostas por DeRemer: (*Look Ahead LR*) LALR [DeRemer, 1969] e (*Simple LR*) SLR [DeRemer, 1971]. A vantagem dessas variações em relação à proposta original de Knuth é que ambas constroem seus autômatos característicos com base no autômato  $LR(0)$ , acrescentando a este os possíveis *strings* de *lookaheads*. Isto faz com que, dada uma gramática, o número de estados do autômato  $SLR(k)$  ou  $LALR(k)$  seja constante para qualquer valor de  $k$ .

As linguagens  $SLR(k)$  são um subconjunto próprio das linguagens  $LALR(k)$  que por sua vez são um subconjunto próprio das linguagens  $LR(k)$ . Na prática, as gramáticas  $LALR(k)$  são as mais utilizadas, pois seus analisadores utilizam menos memória em relação aos analisadores de gramáticas  $LR(k)$ , ao mesmo tempo em que permitem acomodar grande parte das construções de linguagens de programação. Esta dissertação enfoca nos analisadores sintáticos  $LALR(k)$ , embora grande parte das contribuições deste trabalho também sejam aplicáveis a analisadores  $LR(k)$  e  $SLR(k)$ .

## 1.1 Definição do Problema

A grande vantagem de se utilizar gramáticas  $LALR(k)$  é a possibilidade de se gerar automaticamente analisadores sintáticos correspondentes. Para esta tarefa, existem ferramentas especializadas denominadas *geradores de analisadores sintáticos LALR*, dentre as quais merecem destaque Yacc [Johnson, 1979], Bison [Bison, 2007], CUP [CUP, 2007], SableCC [SableCC, 2007], dentre outras.

O uso de geradores LALR é útil quando se deseja um desenvolvimento rápido e seguro [Kaplan e D., 2000]; no entanto, estas ferramentas possuem duas grandes deficiências: ausência de recursos para facilitar a remoção de conflitos e pouca flexibilidade de escolha do nível de compactação das tabelas sintáticas.

**Ausência de recursos para facilitar a remoção de conflitos:** grande parte dos geradores de analisadores LALR não dispõem de recursos eficazes à remoção de conflitos, embora estes sejam recorrentes e de difícil remoção. Usualmente, conflitos são removidos manualmente pelo projetista via análise de *logs* reportados pelo gerador. Ocorre que o volume de dados nesses arquivos é extremamente grande, o que dificulta esta prática. Para se ter uma idéia, o *log* criado pelo gerador Bison [Bison, 2007] para a especificação da linguagem Notus [Tirelo e Bigonha, 2006]<sup>1</sup>, onde são reportados 575 conflitos, tem tamanho igual a 54 Kb, com 2.257 linhas e 6.244 palavras. Aliado a isto está o fato de que esses dados não estão interrelacionados, já que hiperlinks não são possíveis em arquivos texto, o que acarreta em uma difícil navegação. O nível de abstração do conteúdo desses arquivos de *log* também é um problema, uma vez que consistem inteiramente de informações associadas ao funcionamento do autômato LALR. Nesta situação, usuários menos experientes se sentem intimidados e em alguns casos até migram para geradores menos poderosos, como os LL (*Left to Right, Left most derivation*). Estes, apesar de possuírem uma teoria mais clara e simplificada, muitas vezes não são adequados ao reconhecimento de certas construções sintáticas, já que as

---

<sup>1</sup>A gramática de Notus possui 77 terminais, 110 não terminais e 236 produções.

linguagens LL são um subconjunto próprio das linguagens LALR. Para usuários mais experientes, a remoção de conflitos em um ambiente tão ríspido acarreta em perda de produtividade.

**Pouca flexibilidade de escolha do nível de compactação das tabelas sintáticas:**

o tamanho das tabelas sintáticas de analisadores LALR( $k$ ) de linguagens como C++, Ada, Pascal, etc., consomem uma quantidade satisfatória de memória. Desta forma, muitos geradores oferecem mecanismos para compactação das tabelas produzidas. No entanto, não foi encontrado na literatura especializada um gerador que implementasse o conceito de níveis de compactação. Um nível de compactação serve para classificar o grau de compactação da tabela sintática e o conseqüente aumento no tempo de execução do analisador resultante. O aumento na complexidade temporal ocorre devido à substituição da matriz que representa as tabelas sintáticas por uma estrutura menos eficiente do ponto de vista de acesso e/ou pela utilização de ações padrão que atrasam a detecção de erro. Em analogia aos níveis de otimização fornecidos pelo compilador GCC [GCC, 2006], os níveis de compactação podem ser classificados em alto e médio. O primeiro corresponde a um alto grau de compactação, mas com uma deterioração no tempo de execução. O segundo oferece uma taxa de compactação mediana, mas preserva a ordem de complexidade original do analisador sintático gerado. A oferta de um mais de um nível de compactação é necessária, pois possibilita a geração de analisadores sintáticos conforme a disponibilidade de memória dos computadores que irão executá-los. Por exemplo: um analisador sintático projetado para execução em um rede de sensores dispõe de uma quantidade mínima de memória, sendo útil o nível de compactação alto. Outra possibilidade são os analisadores sintáticos de aplicações como editores de texto e navegadores Web escritos para execução em dispositivos móveis, como celulares e PDAs. Para essas situações, é adequado a utilização do nível de compactação mediano.

Dois outros problemas identificados estão relacionados às ferramentas que auxiliam o entendimento (ferramentas educativas) e a produção de analisadores sintáticos LR (ferramentas comerciais), incluindo as variações SLR e LALR. Estas ferramentas não permitem a interpretação de especificações escritas em linguagens senão as aceitas pelo gerador considerado originalmente no projeto das mesmas. Por exemplo, a ferramenta Visual Parse [Parse++, 2005], descrita no Capítulo 2, não permite a utilização de especificações escritas para o CUP [CUP, 2007]. Isto faz com que os usuários fiquem presos a um único gerador e possivelmente a uma única linguagem de especificação, ocasionando menor flexibilidade frente a mudanças de ambiente. Além disso, essas ferramentas dão pouco enfoque à remoção de conflitos, que definitivamente é o maior problema existente na construção de analisadores LALR.

O objetivo desta dissertação é solucionar a falta de recursos e métodos que facilitem a resolução de conflitos e flexibilizar a escolha do nível de compactação desejado pela utilização de uma ferramenta que forneça um ambiente que dê suporte a múltiplas linguagens de especificação.

## 1.2 Solução Proposta

Para eliminar os problemas enunciados, é apresentado neste trabalho o protótipo de uma ferramenta visual para desenvolvimento de analisadores LALR( $k$ ) denominado SAIDE<sup>2</sup> – *Syntax Analyser Integrated Development Environment*.

Para o problema de remoção de conflitos, SAIDE suporta uma metodologia criada com o objetivo de guiar o usuário nesta tarefa. A metodologia proposta divide os conflitos em dois grandes conjuntos: os que podem ser removidos automaticamente e os que devem ser removidos manualmente. O primeiro conjunto constitui a classe de conflitos decorrentes do valor inadequado de  $k$ . SAIDE tenta removê-los pelo aumento do  $k$ , respeitado um limite superior, definido pelo usuário. Na remoção manual, a metodologia proposta define quatro fases com a intenção de capturar a maneira natural que o projetista remove conflitos: (i) entender a causa do conflito; (ii) classificá-lo segundo um conjunto de categorias pré-definidas; (iii) alterar a especificação de modo a remover o conflito; (iv) resubmeter a especificação ao gerador para verificar a eliminação do conflito.

Na parte de compactação de tabelas, a ferramenta gera analisadores sintáticos LALR( $k$ ) cujas tabelas são compactadas de acordo com dois níveis, alto e médio, conforme o perfil desejado pelo projetista.

Por último, para dar suporte a múltiplas linguagens de especificação, a arquitetura da ferramenta estabelece uma camada de abstração que a torna independente de qualquer gerador LALR. Essa camada constitui um arcabouço de escrita de *plugins* para diferentes geradores, permitindo assim, a utilização de especificações codificadas em diferentes linguagens.

## 1.3 Contribuições

As seguintes contribuições são resultado desta dissertação:

- a metodologia proposta sistematiza a remoção de conflitos, que é um processo laborioso e consome grande parte do tempo gasto na construção de analisadores LALR( $k$ );

---

<sup>2</sup>Pronúncia correta: /said/

- criação da ferramenta SAIDE, de forma a dar suporte à metodologia proposta;
- garantia de término do algoritmo de cômputo de *lookaheads* de tamanho  $k$  independentemente de certas características da gramática. Isto permite a aplicação do processo de remoção de conflitos de maneira uniforme;
- cômputo eficiente de árvores de derivação para exibição de conflitos, de forma a consumir o mínimo de memória possível. Os algoritmos originais, embora corretos, em alguns casos consumiam toda a memória do computador;
- na remoção manual, os conflitos são ordenados segundo a prioridade em que devem ser removidos. Esta ordenação visa capturar situações em que conflitos são ocasionados em decorrência de outros; com isto a remoção destes implicam na remoção dos primeiros;
- avaliação dos métodos de compactação atualmente existentes;
- geração das tabelas dos analisadores sintáticos LALR( $k$ ) compactadas, com a flexibilidade de escolha do nível de compactação: alto e médio;
- todas as funcionalidades da ferramenta SAIDE estão disponibilizadas em um ambiente independente de linguagens de especificação, ao contrário do que ocorre nas ferramentas visuais atualmente existentes.

## 1.4 Organização da Dissertação

Esta dissertação está dividida em oito capítulos. No Capítulo 2 são apresentadas ferramentas visuais com propósitos educacionais e comerciais com o objetivo de facilitar o entendimento e/ou construção de analisadores LALR. No Capítulo 3 mostra-se todo o formalismo da análise sintática LALR, com a explanação dos algoritmos de cômputo de *lookaheads*. No Capítulo 4 analisam-se oito métodos de compactação de tabelas LALR, onde apenas duas estratégias de compactação são selecionadas para compor os níveis de compactação alto e médio, respectivamente. O quinto capítulo apresenta a ferramenta proposta SAIDE, desenvolvida de forma a permitir a aplicação de uma metodologia de remoção de conflitos em um ambiente em que várias linguagens de especificação podem ser utilizadas. O capítulo seguinte valida as idéias discutidas nesta dissertação pelo uso de SAIDE, onde são reportados os resultados obtidos desta experiência. Por fim, o Capítulo 7 conclui a dissertação e sumariza os resultados obtidos.



## Capítulo 2

# Ferramentas Visuais para Auxílio na Construção de Analisadores Sintáticos

Este capítulo apresenta algumas ferramentas visuais utilizadas na construção de analisadores sintáticos LR/SLR/LALR. Essas ferramentas são divididas segundo dois enfoques: as que priorizam o aspecto do funcionamento interno dos analisadores sintáticos, explicitando suas estruturas e como as mesmas colaboram para o reconhecimento ou rejeição de um *string* – *ferramentas educativas*, e as que objetivam prover um ambiente que propicie ao projetista uma maior rapidez e integração durante a produção de analisadores sintáticos – *ferramentas comerciais*.

### 2.1 Ferramentas Educativas

Esta seção descreve cinco ferramentas educativas: GYacc [Lovato e Kleyn, 1995], Visual Yacc [Yacc, 2006], Parser [Khuri e Sugono, 1998], CUPV [Kaplan e D., 2000] e a ferramenta LRparse [Blythe et al., 1994].

A apresentação de cada ferramenta é feita pela descrição de suas funcionalidades, exibindo-se as telas disponíveis, seguido de uma explanação sobre o seu funcionamento interno. Algumas explicações são realizadas em termos de diagramas de atividades. A sintaxe desses diagramas é resumida no Apêndice A deste documento.

#### 2.1.1 GYacc

GYacc [Lovato e Kleyn, 1995] é uma ferramenta de animação que simula a execução de analisadores sintáticos produzidos pelo Yacc.

Para animar um analisador sintático de uma especificação Yacc, o usuário simplesmente invoca a ferramenta, que exibe a sua tela principal.

A tela principal <sup>1</sup> é composta por dois painéis: o painel da gramática e o painel dos *strings* de teste. Esses painéis permitem respectivamente a entrada/edição de uma gramática Yacc e de *strings* de teste para verificar a linguagem reconhecida pelo analisador. A gramática fornecida é utilizada na geração do analisador sintático a ser simulado.

Para realizar a animação do analisador sintático, o usuário escolhe um dos *strings* de teste contidos no painel correspondente.

Durante a animação, o usuário tem a sua disposição as seguintes visões:

- floresta de derivação: apresenta as árvores obtidas a cada passo de execução do analisador sintático;
- tabelas de controle: apresenta as tabelas *Action* e *Goto*. Nesta visão, as entradas são destacadas à medida em que as mesmas são consultadas pelo analisador;
- pilha: esta visão apresenta a pilha mantida pelo analisador sintático. A cada passo de execução, a pilha cresce, no caso de ações de empilhamento, ou diminui, no caso de ações de redução. A coloração da pilha a cada passo de execução indica o tipo da operação realizada, se empilhamento ou redução;
- transições de estados: apresenta um diagrama com as transições realizadas até o momento. Cada estado é representado por uma linha horizontal. Uma transição é desenhada como uma seta unidirecional partindo do estado de origem para o estado destino. Esta visão pode ser entendida como uma simplificação da visão da pilha, apresentando somente as transições ocorridas;
- reduções: esta visão proporciona ao usuário uma visualização da produção utilizada em cada redução.

Para controlar a simulação da execução do analisador sintático, GYacc permite realizar a animação de duas formas: passo a passo ou direta. No primeiro caso, o usuário sempre solicita a execução do próximo passo. No segundo caso, a execução é realizada de forma ininterrupta, podendo-se determinar a velocidade da animação.

---

<sup>1</sup>As telas da ferramenta GYacc não são apresentadas neste trabalho devido à péssima qualidade das figuras disponibilizadas no artigo que descreve a ferramenta [Lovato e Kleyn, 1995]. Tentou-se obter as telas via execução da ferramenta, mas o código fonte da mesma não foi encontrado.



### Funcionamento Interno

Dada uma especificação Yacc, GYacc simula a execução do analisador sintático correspondente. Anterior a simulação, a ferramenta realiza os passos apresentados no diagrama de atividades da Figura 2.1, descritos a seguir:

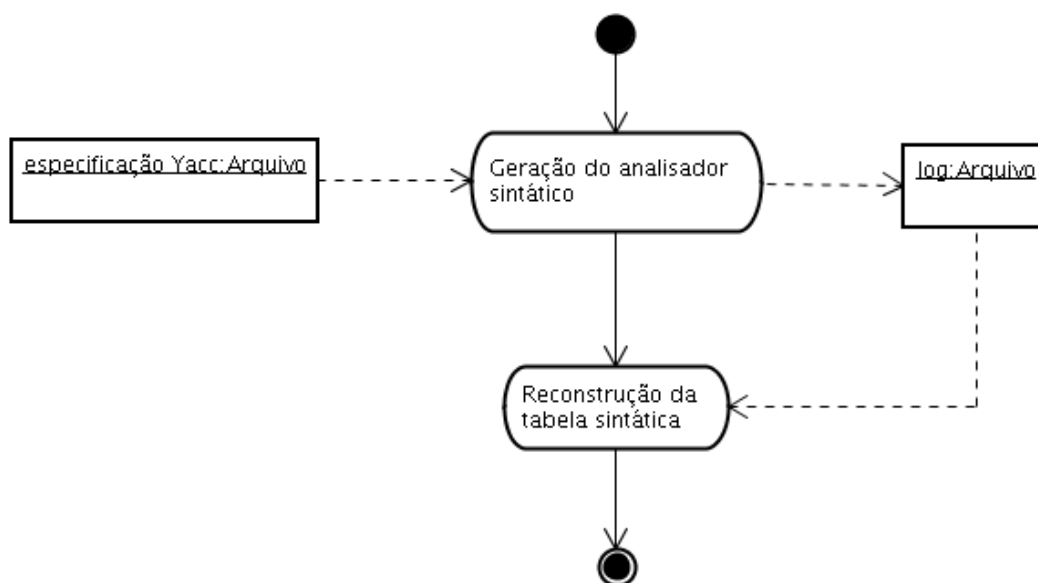


Figura 2.1: Pré-animação - obtenção de informações do analisador sintático.

- processamento da especificação: este passo consiste no processamento do arquivo de especificação escrito na linguagem do Yacc. Neste ponto, a especificação é passada ao Yacc, que a processa e grava um arquivo de *log*. Este arquivo contém, dentre outros dados, o autômato LALR(1). Se forem encontrados erros sintáticos ou conflitos *shift/reduce* ou *reduce/reduce*, o usuário deve corrigi-los sem a assistência do Gyacc, pois este não disponibiliza qualquer recurso neste sentido;
- reconstrução da tabela sintática: este passo processa o arquivo de *log* produzido pelo Yacc e reconstrói em memória a tabela sintática do analisador. Isto permite ao GYacc simular a execução do analisador especificado.

### 2.1.2 Visual Yacc

Visual Yacc [Yacc, 2006] é uma ferramenta para produção de analisadores sintáticos ilustrados a partir de especificações escritas para o Yacc [Johnson, 1979].

Na produção de um analisador ilustrado, a ferramenta recebe um arquivo de especificação Yacc, que é então modificado de modo a conter ações semânticas responsáveis pela ilustração do analisador em questão.

Durante a execução do analisador ilustrado, dado um *string* de entrada, exibe-se uma tela composta basicamente por dois painéis principais – painel da pilha sintática e painel da árvore de derivação, conforme ilustrado pela Figura 2.2. A pilha sintática apresentada não é anotada por padrão; o usuário indica a anotação explicitamente em sua especificação. Para isto, cada tipo declarado deve ser precedido pelo caractere “i”, embora somente tipos inteiros sejam suportados. Como exemplo de uma especificação com um tipo ilustrado, considere o seguinte fragmento de uma especificação escrita para o Yacc:

```

...

%union
{
    int ival;
    char * nome;
}

%token <nome>      NOME
%token <valor>     NUMERO
%type <ival>      expr

...

```

No fragmento acima, quaisquer elementos do tipo `ival` que apareçam na pilha terão os seus valores exibidos.

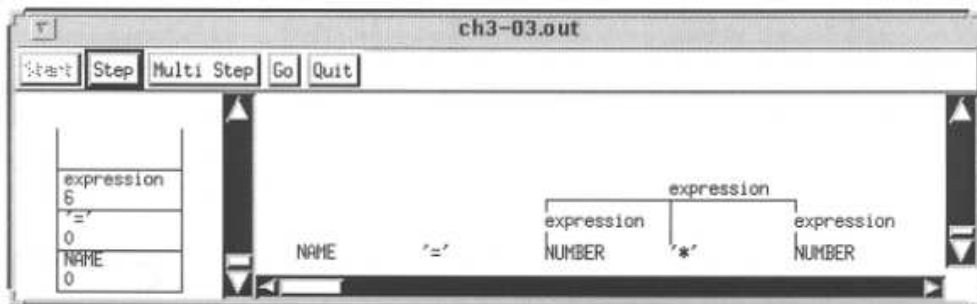


Figura 2.2: Janela do Visual Yacc. Extraído de [Yacc, 2006].

Visual Yacc apresenta um sistema simples de controle da animação, onde é possível:

- executar um passo (botão *step*): realiza um passo no processo de análise sintática;
- executar vários passos (botão *multiple step*): realiza um certo número de passos na análise sintática. Este número é informado pelo usuário;

- executar animação até o fim (botão *go*): analisa todo o *string* de entrada, não permitindo qualquer intervenção do usuário;
- finalizar a animação (botão *quit*): termina a animação em execução.

### Funcionamento Interno

Para gerar um analisador sintático Yacc ilustrado, Visual Yacc realiza dois passos, apresentados na Figura 2.3 e explicados a seguir:

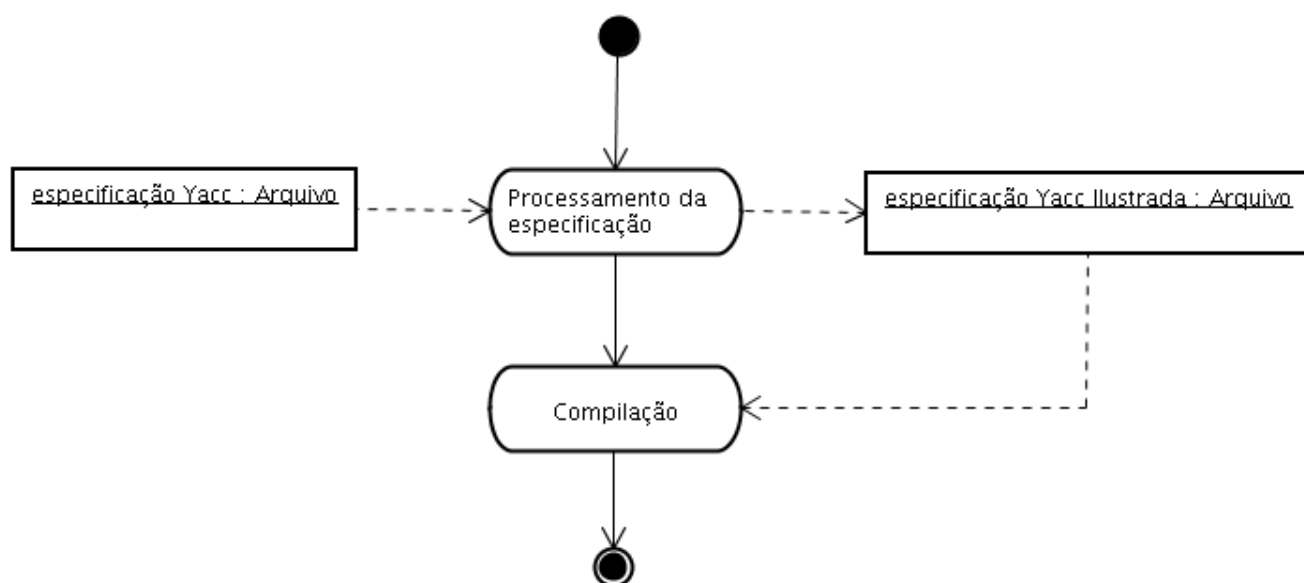


Figura 2.3: Geração de um analisador ilustrado gerado pela ferramenta Visual Yacc.

- processamento da especificação: processa a especificação Yacc de modo a inserir chamadas a rotinas de atualização na forma de ações semânticas. Essas rotinas são responsáveis pela atualização da pilha e árvore sintática exibidas pelo analisador ilustrado;
- compilação: o arquivo de especificação alterado no passo anterior é repassado ao Yacc, onde então são produzidos os fontes C referentes ao analisador sintático ilustrado. Esses fontes são compilados e ligados com os módulos do Motif [Motif, 2007], *toolkit* utilizado para construção das janelas.

### 2.1.3 Parser

Parser [Khuri e Sugono, 1998] é uma ferramenta de animação de analisadores sintáticos LL(1) e SLR(1). Como o foco deste trabalho é em ferramentas voltadas à análise

sintática *bottom-up*, somente a ferramenta de animação de analisadores SLR(1) é apresentada.

A ferramenta inicia sua execução quando o usuário a invoca pela linha de comando, onde é exibido um menu. Neste, o usuário solicita a carga do arquivo com a gramática e a tabela sintática SLR(1), codificadas em um formato específico à ferramenta, e fornece também um *string* de entrada. A tela principal da ferramenta é então apresentada – conforme mostrado na Figura 2.4.

Nessa tela a ferramenta Parser exibe quatro componentes: *string* de entrada, caixa da ação sintática realizada, pilha e a árvore sintática, cada um representando uma visão. O relacionamento entre esses componentes é indicado por setas.

Além das quatro visões mencionadas, uma quinta é exibida na janela do console onde a ferramenta foi invocada. Essa visão é apresentada na forma de uma tabela, dividida em três colunas: pilha sintática (*Parsing Stack*), entrada (*Input*) e ação (*Action*). As linhas dessa tabela estão dispostas em ordem crescente do instante de execução, ou seja, a primeira linha reflete a pilha, a entrada e a ação sintática no instante  $t_1$  de execução. Já a segunda linha refere-se ao instante  $t_2$ , e assim por diante. Isto é ilustrado na Figura 2.5.

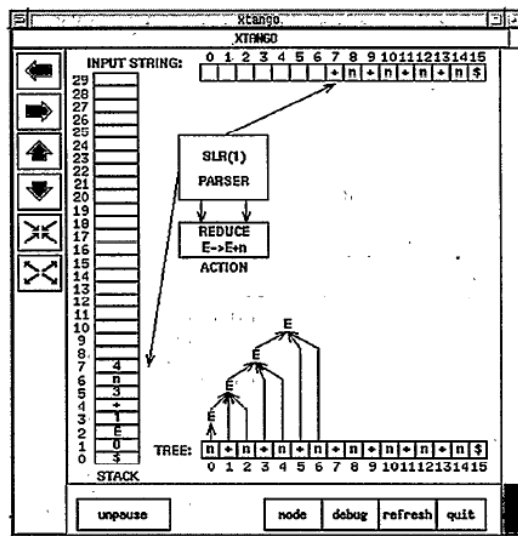
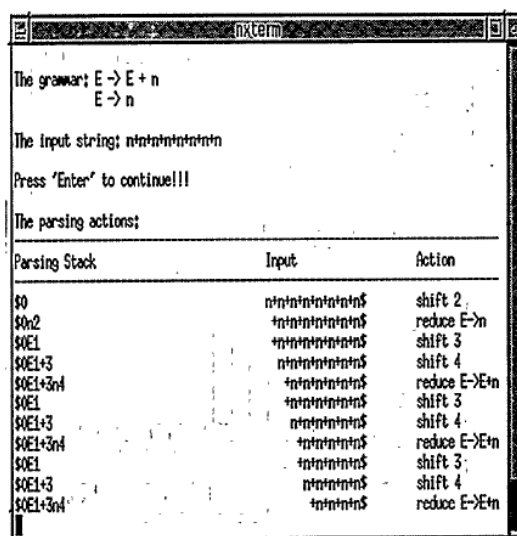


Figura 2.4: Janela principal da ferramenta Parser. Extraído de [Khuri e Sugono, 1998].

Para controlar a animação, a ferramenta Parser fornece meios de se definir a velocidade de animação e realizar pausas.

### Funcionamento Interno

Parser utiliza as três entradas fornecidas pelo usuário: a gramática, a tabela sintática e o *string* de entrada. A partir da carga e validação dessas entradas, a ferramenta executa



The grammar:  $E \rightarrow E + n$   
 $E \rightarrow n$

The input string: n+n+n+n+n

Press 'Enter' to continue!!!

The parsing actions:

Parsing Stack	Input	Action
\$0	n+n+n+n+n\$	shift 2
\$0n2	+n+n+n+n+n\$	reduce $E \rightarrow n$
\$0E1	+n+n+n+n+n\$	shift 3
\$0E1+3	n+n+n+n+n\$	shift 4
\$0E1+3n4	+n+n+n+n+n\$	reduce $E \rightarrow E+n$
\$0E1	+n+n+n+n+n\$	shift 3
\$0E1+3	n+n+n+n+n\$	shift 4
\$0E1+3n4	+n+n+n+n+n\$	reduce $E \rightarrow E+n$
\$0E1	+n+n+n+n+n\$	shift 3
\$0E1+3	n+n+n+n+n\$	shift 4
\$0E1+3n4	+n+n+n+n\$	reduce $E \rightarrow E+n$

Figura 2.5: Console da ferramenta Parser. Extraído de [Khuri e Sugono, 1998].

um algoritmo de análise sintática codificado internamente, responsável por atualizar os dados exibidos ao usuário durante a execução do analisador.

### 2.1.4 LRParse

LRparse [Blythe et al., 1994] é uma ferramenta visual e interativa para construção de analisadores LR(1), dada uma gramática correspondente.

A construção do analisador é dividida em fases, cada uma apresentada em uma tela distinta. O usuário prossegue para a próxima tela somente quando o passo corrente é finalizado com sucesso. A única exceção ocorre quando o usuário fornece a gramática da linguagem. Se ela não for LR(1), o usuário é devidamente alertado e se depara com dois possíveis caminhos: realizar as alterações necessárias a torná-la LR(1), ou então seguir até o ponto de preenchimento da tabela sintática de forma a identificar mais de uma ação para uma mesma entrada da tabela, o que indica a ocorrência de um conflito.

Na janela inicial de LRparse, apresentada na Figura 2.6, o usuário fornece uma gramática LR(1) com no máximo 15 regras.

Fornecida a gramática, uma nova janela é exibida (vide Figura 2.7) com conjuntos *FIRST* vazios, de modo que o usuário os preencha devidamente. Após a entrada correta desses conjuntos, uma janela similar é exibida para o fornecimento dos conjuntos *FOLLOW*. No quarto passo, o usuário constrói o autômato LR(1) em questão (o número de estados é limitado a 25) e define o conjunto de itens correspondente. Isto é apresentado nas Figuras 2.8a e 2.8b.

O passo final na concepção de um analisador sintático LR(1) é o preenchimento da tabela sintática, apresentada em branco ao usuário. Um exemplo de preenchimento é

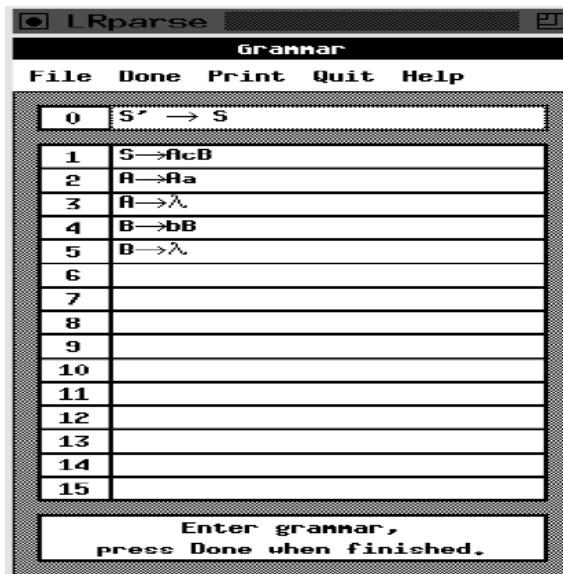


Figura 2.6: Janela principal da ferramenta LRparse. Extraído de [Blythe et al., 1994].

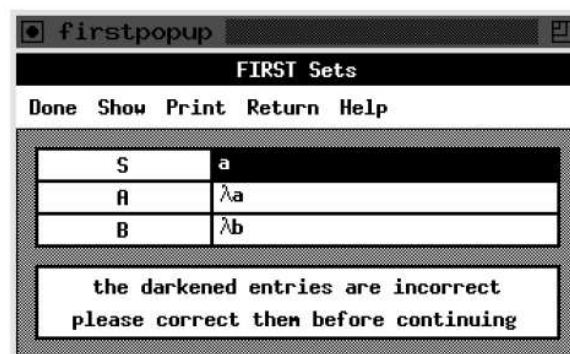


Figura 2.7: Janela para a entrada do conjunto *FIRST*. Extraído de [Blythe et al., 1994].

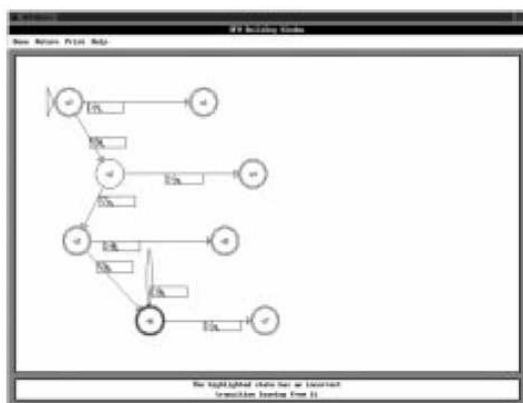
apresentado na Figura 2.9.

Para validar a entrada fornecida a cada etapa, o usuário sempre clica no botão *Done* disponível em todas as janelas. Ao fazer isto, a informação fornecida é verificada e erros, se ocorrerem, são apresentados e indicados visualmente. Do contrário, a janela da etapa seguinte é exibida.

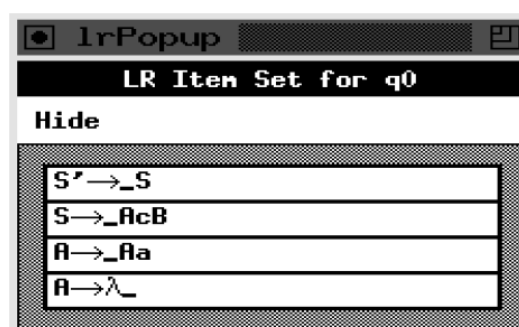
Realizados os cinco passos necessários à definição do analisador sintático LR(1), o usuário acompanha a análise sintática para *strings* de exemplo por ele definidos. Um exemplo de execução passo a passo é mostrado na Figura 2.10.

### 2.1.5 CUPV

CUPV [Kaplan e D., 2000] é uma ferramenta para produção de analisadores sintáticos ilustrados a partir de especificações CUP [CUP, 2007].



(a) Definição da máquina de estados finita.



(b) Definição do conjunto de itens para um dado estado do autômato LR.

Figura 2.8: Janela para a criação do autômato LR(1). Extraído de [Blythe et al., 1994].

	c	a	b	\$	S	A	B
0	r3	r3			0	1	2
1				acc	1		
2	s3	s4			2		
3				r5	3		5
4	r2	r2			4		
5				r1	5		
6			s6	r5	6		7
7				r4	7		

there are errors in your parse table  
please correct them before continuing

Figura 2.9: Janela para o preenchimento da tabela sintática. Extraído de [Blythe et al., 1994].

Para produzir um analisador ilustrado, o usuário invoca a ferramenta passando a ela um arquivo de especificação CUP, que é então modificado de modo a conter chamadas específicas a rotinas responsáveis pela ilustração do analisador em questão.

Ao executar um analisador gerado por CUPV, dado um *string* de entrada, é apresentada uma tela, conforme mostrado na Figura 2.11.

Nessa tela, destaca-se a pilha sintática, onde cada elemento é um botão rotulado pelo símbolo e pelo estado correspondente. Outras informações também são apresentadas pelo analisador ilustrado, tais como o *token*, o estado corrente e um *log* de ações ocorridas até o momento. A partir dos itens apresentados, o usuário pode visualizar

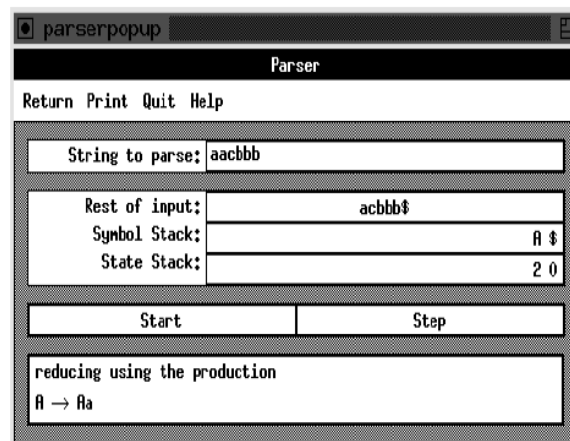


Figura 2.10: Janela de análise sintática de um *string* de entrada. Extraído de [Blythe et al., 1994].

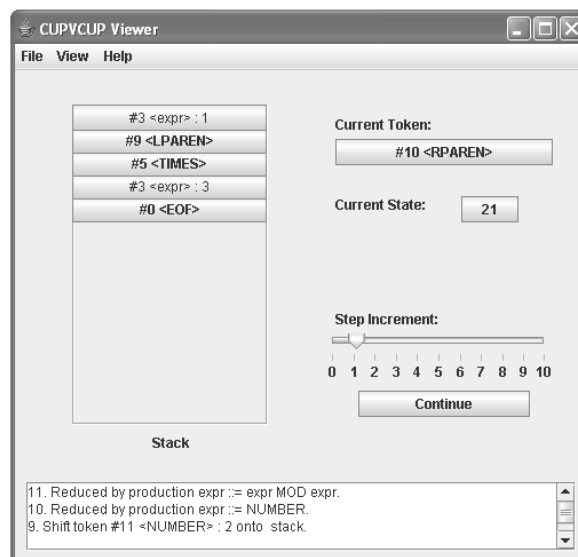


Figura 2.11: Tela principal de um analisador sintático ilustrado gerado por CUPV. Adaptado de [Kaplan e D., 2000].

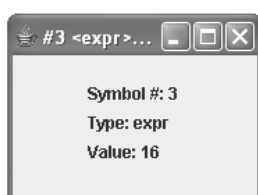
as seguintes informações:

- valores semânticos: como mencionado anteriormente, os símbolos da pilha são apresentados na forma de botões. Quando existe um valor semântico associado a um elemento na pilha, o mesmo é colorido de forma a diferenciá-lo dos demais. Sabendo disso, o usuário clica no botão referente ao elemento, sendo exibido uma tela com as informações semânticas correspondentes. Um exemplo dessa janela é apresentada na Figura 2.12a;
- detalhamento de uma redução em execução: quando uma ação de redução é executada, os símbolos a serem retirados do topo da pilha são devidamente co-

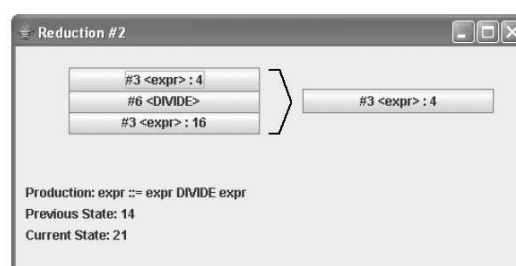


loridos. O *log* de ações é atualizado, acrescido da mensagem “*Getting ready to reduce*”. Quando o usuário clica no botão *Continue* da tela principal, o analisador ilustrado mostra uma janela contendo a subpilha da pilha original, com os símbolos que formam o lado direito da produção utilizada na redução, qual símbolo não terminal irá substituí-los, a produção utilizada, o estado antecessor e o estado corrente. A Figura 2.12b ilustra esta situação. Todos os símbolos nessa janela também são apresentados na forma de botões; quando um deles é clicado, a ferramenta exibe o valor semântico correspondente, caso este exista;

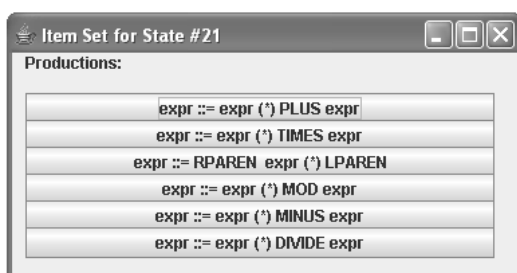
- visualização do conjunto de itens do estado corrente: na tela principal é exibido, na forma de um botão, o estado corrente. Ao clique desse botão, a ferramenta mostra uma tela com os itens LALR(1) do respectivo estado, apresentados também como botões. Quando um deles é clicado, a ferramenta exibe os *lookaheads* correspondentes. Estes dois casos são ilustrados pelas Figuras 2.12c e 2.12d, respectivamente.



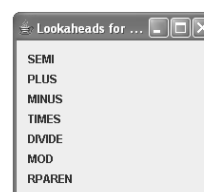
(a) Janela de exibição de informações semânticas.



(b) Janela de exibição de informações de redução.



(c) Janela de exibição do conjunto de itens do estado corrente.



(d) Janela de exibição do conjunto de *lookaheads* de um item.

Figura 2.12: Janelas complementares da ferramenta CUPV. Adaptado de [Kaplan e D., 2000].

Para controle da execução do analisador sintático, o usuário estabelece, por meio da barra *Step Increment* da tela principal, o número de passos a serem realizados

antes que lhe seja solicitado a continuação da animação. Considere o caso em que o usuário estabelece quatro como o número de passos. Inicialmente, o usuário possui como única opção a inicialização da animação, via botão *Start*. Ao clicar nesse botão, o rótulo do mesmo é alterado para *Pause*, sendo permitido a realização de pausa em qualquer instante dos próximos quatro passos de execução. Se isto ocorrer, o rótulo do botão *Pause* é alterado para *Continue*, onde o analisador ilustrado espera até que o usuário retome a animação. No final dos quatro passos, o botão *Pause* é alterado para *Continue*, podendo o usuário reconfigurar ou não o número de passos a ser executado. Em ambos os casos, tem-se a posterior retomada de animação.

### Funcionamento Interno

O artigo que descreve o gerador CUPV não menciona o funcionamento interno necessário à geração de analisadores sintáticos ilustrados, restringindo-se a reportar somente a linguagem de programação e arcabouço utilizados.

CUPV é implementado em Java, com a utilização de um arcabouço descrito em [Shoup, 1999]. Esse arcabouço é usado para definição e personalização de visualizações de analisadores sintáticos, permitindo dentre outras coisas, a customização da forma de exibição dos valores semânticos.

## 2.2 Ferramentas Comerciais

As ferramentas comerciais disponíveis para construção de analisadores sintáticos visam prover uma ambiente mais produtivo para a construção e depuração desses programas.

São apresentadas duas ferramentas: AnaGram [AnaGram, 2006] e Visual Parse++ [Parse++, 2005], utilizadas na confecção de analisadores sintáticos LALR(1) e LALR( $k$ ) respectivamente.

A apresentação de cada ferramenta é feita da seguinte forma: inicialmente o ambiente de desenvolvimento e suas principais funcionalidades são descritos. Em seguida, mostra-se a geração do analisador referente à especificação e como o mesmo é integrado às aplicações. Detalhes de implementação e/ou arquitetura não são discutidos, já que essas informações não são disponibilizadas pelos fabricantes.

### 2.2.1 AnaGram

O ambiente de trabalho da ferramenta AnaGram permite a análise, depuração e realização de testes sobre arquivos escritos em sua linguagem de especificação.

Na análise, a ferramenta apresenta ao usuário um conjunto de estatísticas referentes ao arquivo de especificação, tais como o número de linhas lido, número de *tokens*

lidos, número de estados criados no autômato LALR(1), número de conflitos, número de mensagens de alerta, dentre outros dados. Nesse momento, o usuário pode ainda solicitar a geração de outras informações, tais como a listagem dos procedimentos utilizados nas ações semânticas, conflitos resolvidos via precedência, tabela de produções, tabela de estados, tabelas de *tokens*, visualização do arquivo de especificação, etc.

Para depuração de gramáticas, AnaGram disponibiliza um mecanismo denominado *Rastreamento de gramática*. Este mecanismo disponibiliza duas visões e um painel para seleção de *tokens*. Esses itens são apresentados na Figura 2.13 e explicados a seguir:

- visão da pilha sintática: são exibidos a seqüência de *tokens* obtidos da entrada e os estados em que foram lidos. Cada item da pilha possui um índice à sua esquerda, o que permite identificar a ordem de empilhamento. Quando da redução por uma produção, os elementos no topo da pilha constituintes do *handle* são devidamente desempilhados e o símbolo não terminal referente ao lado esquerdo da produção envolvida é empilhado;
- visão da pilha de regras sintáticas: exibe as regras ativas em qualquer nível da pilha sintática. No caso de redução, esta pilha é devidamente atualizada;
- painel de *tokens* permitidos<sup>2</sup>: para explorar a gramática durante o rastreamento, pode-se selecionar um *token* de entrada dentre os disponibilizados no painel de entradas permitidas. Este painel exibe os *tokens* consentidos no estado corrente da gramática e a ação resultante da escolha de cada um. O campo de ação de cada *token* indica, no caso de sua seleção, a ocorrência de um empilhamento (>>) ou redução (<<), com o respectivo estado destino ou produção a ser reduzida. As visões da pilha e regras sintáticas são devidamente atualizadas para refletir a escolha feita.

O acompanhamento da depuração é feito passo a passo, podendo-se retroceder ou avançar na execução.

Para a realização de testes, AnaGram permite o rastreamento de arquivos. Isto consiste na utilização de arquivos de testes para validação da gramática construída. A tela disponibilizada ao usuário é similar à apresentada no rastreamento de gramáticas. Um exemplo é exibido na Figura 2.14. Essa tela possui as visões da pilha sintática e de regras sintáticas, idênticas às mencionadas anteriormente, e um painel que contém o arquivo em teste. Neste arquivo, a área consumida pelo analisador léxico é indicada visualmente ao usuário. Como no rastreamento de gramática, o rastreamento de arquivo é realizado a passo a passo, sendo permitido o retrocesso ou avanço a um ponto

---

<sup>2</sup>Uma alternativa à escolha de *tokens* é digitar a entrada a ser utilizada. Isto é feito na *combo box* abaixo da visão de regras sintáticas.

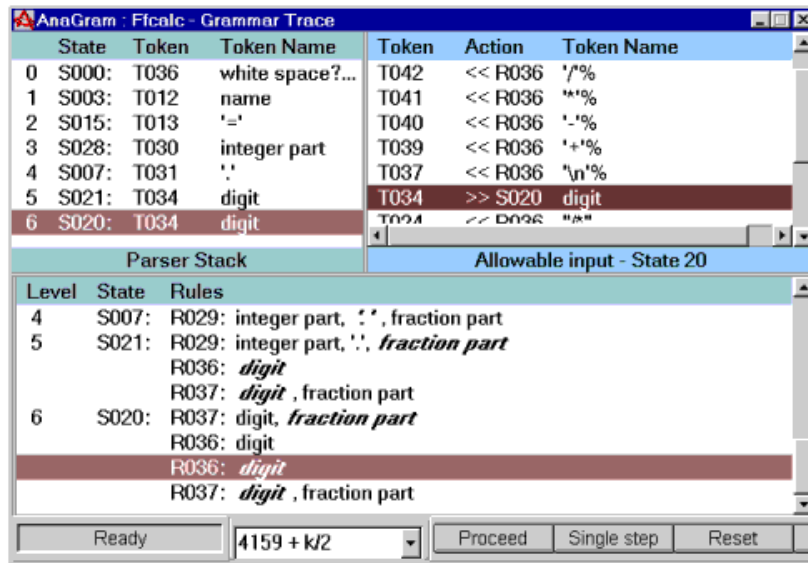


Figura 2.13: Rastreamento de gramática. À esquerda tem-se a pilha sintática, à direita o painel de *tokens* válidos e na parte de baixo da tela, a visão da pilha de regras sintáticas. Extraído de [AnaGram, 2006].

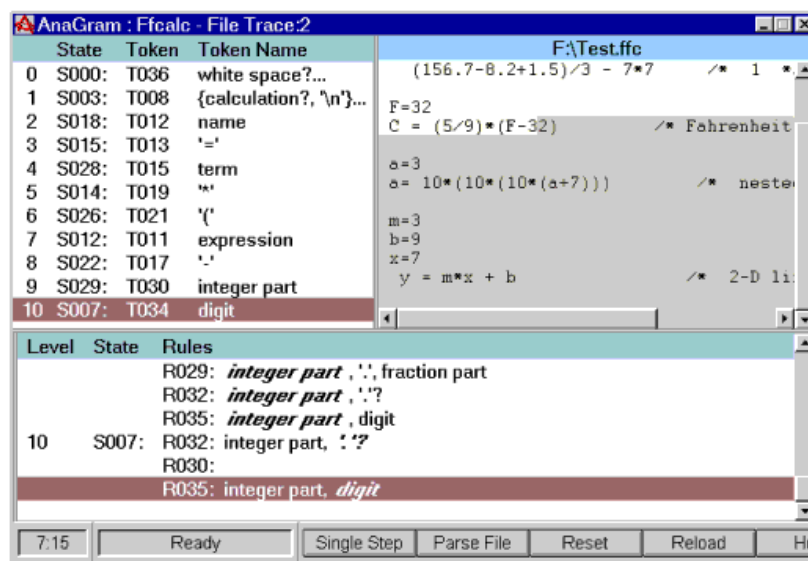


Figura 2.14: Rastreamento de arquivo. À esquerda tem-se a pilha sintática, à direita o painel do arquivo de teste e na parte de baixo da tela, a visão da pilha de regras sintáticas. Extraído de [AnaGram, 2006].

específico da execução. No fim do rastreamento, a ferramenta exibe ao usuário um mensagem de sucesso ou falha resultante da análise sintática realizada.

## Geração do Analisador Sintático

AnaGram gera, a partir da especificação fornecida, analisadores sintáticos em C ou C++.

O usuário pode configurar algumas características do analisador sintático produzido via diretivas incluídas na própria especificação. Dentre essas opções, pode-se estabelecer o tamanho máximo da pilha sintática, definir a forma de comunicação com o analisador sintático (se orientado ou não a eventos), se *thread safe*, etc.

### 2.2.2 Visual Parse++

Visual Parse++ [Parse++, 2005] é uma IDE comercial utilizada na criação de analisadores sintáticos LALR( $k$ ).

A IDE, cuja tela principal é apresentada na Figura 2.15, permite a edição, compilação, execução e depuração do analisador sintático em produção. Exceto pela edição, todas esses recursos são similares aos disponibilizados na ferramenta AnaGram.

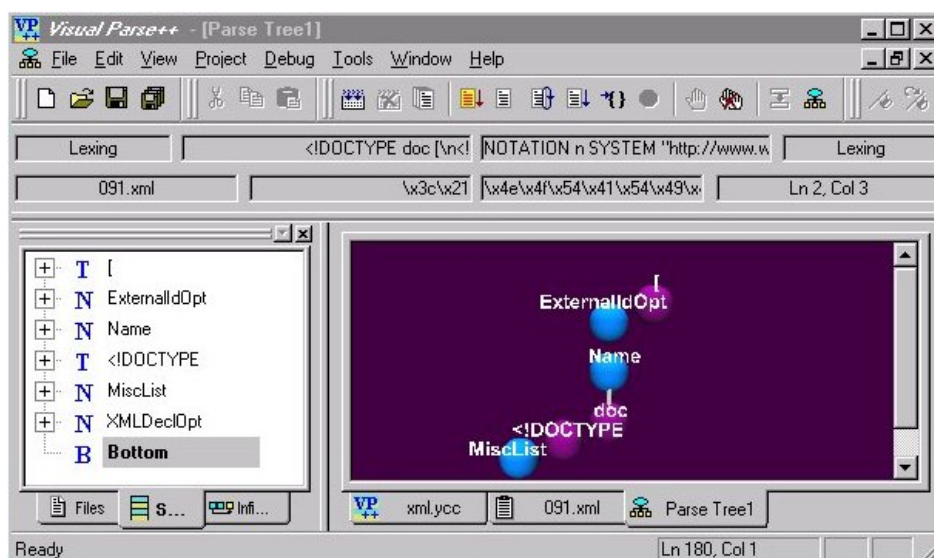


Figura 2.15: Tela principal da ferramenta Visual Parse++.

O grande diferencial desta ferramenta é o suporte à remoção de conflitos. Durante a compilação, cada conflito *shift/reduce* ou *reduce/reduce* é reportado da mesma forma que em outros geradores LALR(1). Selecionado um conflito, a ferramenta exibe a tela *Inflouk* (*infinite lookahead*), conforme mostrado na Figura 2.16. Nessa janela pode-se solicitar a remoção do conflito, embora a remoção efetiva nem sempre seja possível de ser realizada. O indicativo de falha na remoção é não determinístico, pois depende da intervenção do usuário; se a mensagem de confirmação da remoção demora a ser

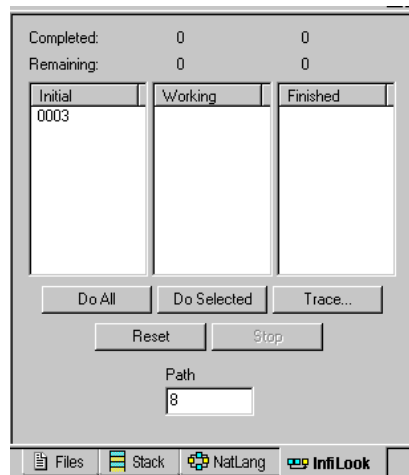


Figura 2.16: Visual Parse++: Janela de diálogo *Inflook*.

exibida, o usuário assume a impossibilidade da remoção e solicita à ferramenta que pare a execução do algoritmo.

No caso de uma remoção manual, o usuário a realiza via estabelecimento de precedência e/ou associatividade, ou reescreve algumas produções da gramática. Para auxiliar neste processo, Visual Parse++ disponibiliza o rastreamento do conflito, que consiste na apresentação do conflito na forma de árvores de derivação. Um exemplo disto é apresentado na Figura 2.17.

Nessa figura, o conflito apresentado é decorrente da derivação  $unique \Rightarrow \lambda$ , anterior ao nodo *cluster*, que deriva **IDENTIFIER**, o símbolo de conflito. Para cada item envolvido no conflito, a ferramenta apresenta a árvore de derivação correspondente. Neste caso, a árvore de derivação exibida corresponde ao item  $cluster \rightarrow \lambda \bullet$  do conflito.

Visual Parse++ possui uma linguagem própria de especificação de gramática, similar à notação BNF. Especificações escritas para o YACC também são aceitas.

## Geração do Analisador Sintático

A ferramenta gera analisadores sintáticos divididos em módulos, codificados em C/C++, C#, Java, Delphi ou Visual Basic. Esse código é integrado a uma aplicação pela inclusão das bibliotecas que definem as interfaces de cada módulo. O conjunto de classes gerado, permite dentre outras coisas, carregar as tabelas referentes ao analisador léxico e sintático de forma a iniciar a análise sintática.

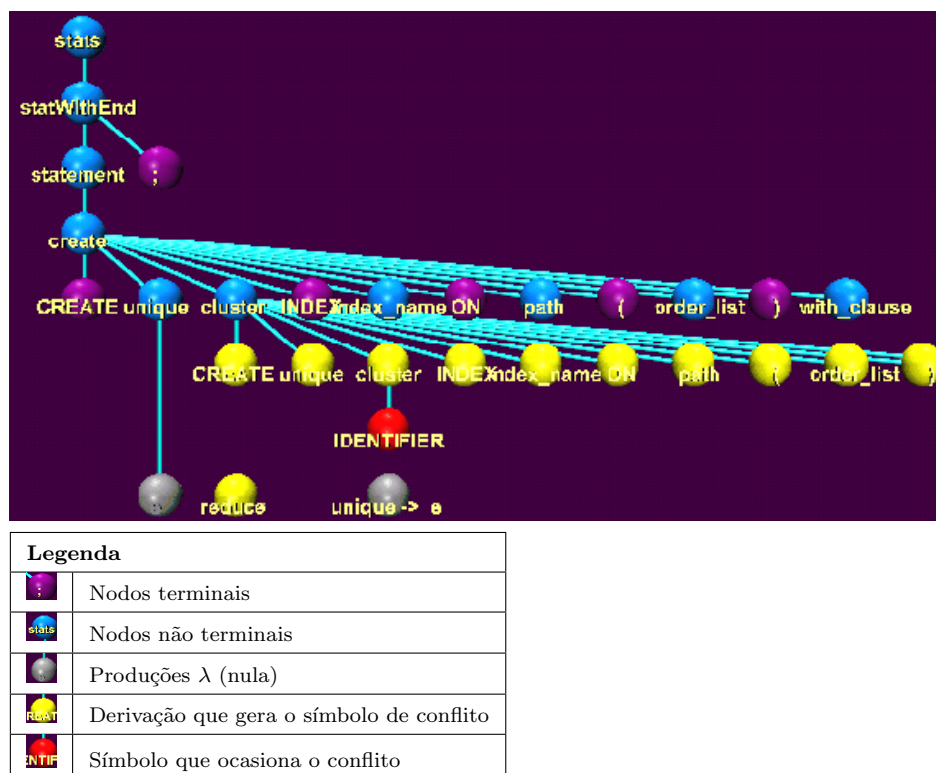


Figura 2.17: Visual Parse++: Árvore de derivação de um item de conflito.

## 2.3 Avaliação das Ferramentas

Como pôde ser observado, todas as ferramentas apresentadas enfocam a depuração do analisador sintático produzido. A depuração realizada nas ferramentas atendem a três propósitos:

- i) facilitar o relacionamento entre a especificação gramatical com o analisador sintático gerado. Isto ocorre porque a especificação, fornecida em uma linguagem que evidencia as estruturas da linguagem para a qual o analisador sintático é construído, é convertida em um código escrito em C++, Java, etc., onde são realizados inúmeros controles que acabam obscurecendo tal relacionamento;
- ii) evidenciar como as estruturas de dados utilizadas pelo analisador sintático funcionam e como elas colaboram para a aceitação/rejeição de uma *string*. Isto ameniza a dificuldade encontrada por projetistas iniciantes na compreensão da análise sintática LR/LALR/SLR [Vegdahl, 2000], [Blythe et al., 1994], [Stasko, 1992] e [Kaplan e D., 2000];
- iii) evitar que o projetista inclua código de depuração nas ações semânticas da especificação. Esse código geralmente inclui a impressão de mensagens de texto que permitem acompanhar os passos de execução do analisador sintático produzido.

Gramática	Produções	Terminais	Não terminais	Conflitos	Conflitos/ Produção
Algol-60	131	58	67	61	0.47
Scheme	175	42	83	78	0.45
Oberon-2	213	75	112	32	0.15
Notus	236	77	110	575	2.44

Tabela 2.1: Número de conflitos encontrados em algumas gramáticas de teste.

Embora útil, a inserção desse tipo de código polui a especificação e gera um duplo esforço dado pela sua codificação e posterior remoção.

Os dois primeiros propósitos são enfocados pelas ferramentas educacionais, ao passo que o último é comum às ferramentas comerciais e educacionais.

Ocorre, no entanto, que exceto pela ferramenta Visual Parse++, todas as outras não se preocupam com um momento anterior à depuração – a remoção de conflitos, pressupondo a inexistência dos mesmos.

No caso das ferramentas educativas, o objetivo de garantir o entendimento do funcionamento de analisadores LR/SLR/LALR pode ser prejudicado, pois projetistas iniciantes, que constituem o público alvo desse tipo de ferramenta, não dispõem de recursos, tampouco experiência para remover eventuais conflitos decorrentes da especificação. Se consideradas as ferramentas comerciais, a ausência de mecanismos para auxílio de remoção de conflitos acarreta em um maior tempo de desenvolvimento do analisador sintático. Isto ocorre devido ao fato de que conflitos são freqüentes em especificações gramaticais, e muitas vezes de difícil remoção. Para demonstrar a alta freqüência em que conflitos são obtidos, a Tabela 2.1 mostra o número de conflitos em gramáticas de algumas linguagens de programação. Estes números foram obtidos a partir de experimentos realizados com a ferramenta SAIDE, apresentada no Capítulo 5. As gramáticas utilizadas foram codificadas utilizando-se a especificação oficial de cada linguagem. Observe que na gramática de Notus, tem-se mais de dois conflitos por produção. Nas gramáticas de Algol-60 e Scheme aproximadamente a cada duas produções da gramática, um conflito é reportado.

Outro problema identificado nas ferramentas estudadas é a inflexibilidade para se trabalhar com mais de um tipo de linguagem de especificação. Assim, se o usuário possui uma gramática escrita em uma linguagem específica a um gerador e deseja usufruir das facilidades oferecidas por uma outra ferramenta, necessariamente deve reescrever sua especificação na linguagem aceita pelo gerador acoplado à ferramenta a qual deseja migrar, o que é um processo custoso, muitas vezes realizado de forma manual.



## 2.4 Conclusão

Este capítulo apresentou um conjunto de ferramentas dividido segundo dois enfoques não exclusivos entre si: um que prioriza o aspecto educativo e outro com maior ênfase na diminuição do tempo gasto na construção de analisadores sintáticos LALR pela utilização de ambientes integrados de desenvolvimento.

Do estudo realizado, conclui-se que embora o número de ferramentas visuais relacionadas ao desenvolvimento e entendimento de analisadores LR/SLR/LALR seja considerável, existe pouca preocupação por parte dessas ferramentas em prover mecanismos para remoção de conflitos. Esta desconsideração, no entanto, não é justificada, pois conforme argumentado, a ocorrência de conflitos é extremamente recorrente e sua remoção é responsável por grande parte do tempo gasto na produção de tais analisadores.

Para suprir esta necessidade, a ferramenta SAIDE enfoca a resolução de conflitos, via suporte a uma metodologia que sistematiza o processo de remoção em um ambiente independente de linguagens de especificação. As funcionalidades disponibilizadas por SAIDE incluem a exibição modularizada e propriamente ligada dos elementos relacionados à análise LALR (autômato característico, a gramática e os conflitos obtidos), visualização dos conflitos na forma de árvores de derivação, apresentação de exemplos clássicos de ambigüidade e suas possíveis soluções de forma a permitir que o projetista adapte-as a seu contexto, resolução automática de uma subclasse de conflitos e listagem dos conflitos segundo uma ordenação de prioridade. A ferramenta dispõe ainda de geração de analisadores sintáticos LALR( $k$ ) cujas tabelas sintáticas são compactadas de acordo com o perfil da aplicação alvo. Estas funcionalidades são explicadas em detalhes no Capítulo 5 e comparadas às oferecidas por Visual Parse++, ferramenta que mais se assemelha a SAIDE.



# Capítulo 3

## Análise Sintática LALR( $k$ )

Este capítulo apresenta o formalismo da análise sintática LALR( $k$ ) necessário à posterior compreensão dos algoritmos utilizados na ferramenta SAIDE. A partir da definição dos conceitos iniciais e da terminologia adotada, apresenta-se as situações que resultam em conflitos, o que permite classificá-los. Em seguida discute-se a fronteira entre os conflitos que aparecem na obtenção de analisadores LR( $k$ ) em relação aos obtidos na construção de analisadores LALR( $k$ ). No restante do capítulo, descreve-se o algoritmo de DeRemer e Penello [DeRemer e Pennello, 1982] e o proposto por Charles [Charles, 1991] para cômputo de *lookaheads* de tamanho igual ou maior ou igual a 1 respectivamente, utilizados na geração do autômato LALR.

As formulações e definições apresentadas são baseadas naquelas utilizadas nos estudos feitos por [Aho et al., 1986], [Aho e Ullman, 1972], [Kristensen e Madsen, 1981], [DeRemer e Pennello, 1982] e [Charles, 1991].

### 3.1 Conceitos e Terminologia Adotada

Uma gramática livre de contexto é uma quádrupla  $G = (N, \Sigma, P, S)$ , onde  $N$  é o conjunto finito de não terminais,  $\Sigma$  o conjunto finito de símbolos terminais,  $P$  o conjunto finito de produções e  $S$  o símbolo de partida. O vocabulário  $V$  de  $G$  é  $V = N \cup \Sigma$ . Presume-se que toda gramática esteja em sua forma estendida  $(N', \Sigma', P', S')$ , onde  $N' = N \cup \{S'\}$ ,  $\Sigma' = \Sigma \cup \{\$\}$ ,  $P' = P \cup \{S' \rightarrow S\$\}$ , desde que  $S' \notin N$  e  $\$ \notin V$ .

Letras gregas minúsculas, como  $\alpha, \beta, \gamma, \dots$ , denotam *strings* em  $V^*$ ; letras românicas do início do alfabeto ( $a, b, c$ ), dígitos, símbolos de operação ( $+, -, *, /$ , etc.), *strings* em negrito (**id**, **if**, etc.) ou contidos entre aspas duplas indicam símbolos em  $\Sigma$ ; letras minúsculas do final do alfabeto ( $x, y, z$ ) representam símbolos em  $\Sigma^*$ ; letras maiúsculas do início do alfabeto ( $A, B, C$ ) e *strings* escritos em caixa baixa em itálico, como *expr* e *stmt*, representam símbolos em  $N$ ; letras maiúsculas do final do alfabeto ( $X, Y, Z$ )

denotam elementos em  $V$ . O *string* vazio é dado por  $\lambda$ , o comprimento de um *string*  $\gamma$  por  $|\gamma|$  e  $\Omega$  representa a constante de indefinição.

**Definição 1.**

$$FIRST_k(\alpha) = \{x \mid (\alpha \xrightarrow[*]{lm} x\beta \text{ e } |x| = k) \text{ ou } (\alpha \xrightarrow[*]{} x \text{ e } |x| < k)\}$$

Dada uma gramática livre de contexto,  $FIRST_k(\alpha)$  consiste em todos os prefixos de terminais de tamanho menor ou igual a  $k$  deriváveis a partir de  $\alpha$ .

**Definição 2.** Seja  $G$  uma gramática livre de contexto. O autômato LR( $k$ ) referente a  $G$  é uma sêxtupla  $LRA_k = (M_k, V, P, IS, GOTO_k, RED_k)$ , onde  $M_k$  é um conjunto finito de estados,  $V$  e  $P$  são como em  $G$ ,  $IS$  é o estado inicial,  $GOTO_k : M_k \times V^* \rightarrow M_k$  é a função de transição e  $RED_k : M_k \times \Sigma_k^* \rightarrow \mathcal{P}(P)$  é a função de redução, com  $\Sigma_k^* = \{w \mid w \in \Sigma^* \wedge 0 \leq |w| \leq k\}$ .

**Definição 3.** Um estado em  $M_k$  é composto por itens LR( $k$ ). Esses itens são elementos em  $N \times V^* \times V^* \times \mathcal{P}(\Sigma_k^*)$  e são escritos de duas formas:

- a)  $(A \rightarrow \alpha \bullet \beta, \{w_1, w_2, \dots, w_n\})$ , onde  $w_1, w_2, \dots, w_n$  são *strings* de *lookaheads*. Esta forma é usada no caso de  $k \geq 1$ ;
- b)  $A \rightarrow \alpha \bullet \beta$ , caso contrário.

O primeiro componente de um item LR( $k$ ) é denominado *núcleo*.

**Definição 4.** Seja  $K$  um conjunto de itens LR( $k$ ). O fechamento de  $K$ , definido pela função  $CLOSURE$ , é dado por:

$$CLOSURE(K) = K \cup \{(A \rightarrow \bullet \omega, \{x_1, x_2, \dots, x_n\}) \mid \begin{array}{l} (B \rightarrow \alpha \bullet A\beta, \{w_1, w_2, \dots, w_n\}) \in K \\ \wedge A \rightarrow \omega \in P \\ \wedge x_1 = FIRST_k(\beta w_1), \\ \wedge x_2 = FIRST_k(\beta w_2), \\ \dots \\ \wedge x_n = FIRST_k(\beta w_n) \end{array}\}$$

**Definição 5.** A função  $GOTO_k$  é definida pelas equações:

$$\begin{aligned} GOTO_k(p, \lambda) &= p \\ GOTO_k(p, X) &= F^{-1}(CLOSURE(ADVANCE(p, X))) \\ GOTO_k(p, X\alpha) &= GOTO_k(GOTO_k(p, X), \alpha), \forall \alpha \neq \lambda \end{aligned}$$

onde  $ADVANCE(p, X)$  é dado por

$$\begin{aligned} ADVANCE(p, X) = & \{(A \rightarrow \alpha X \bullet Y \beta, \{x_1, x_2, \dots, x_n\}) \mid \\ & (A \rightarrow \alpha \bullet XY \beta, \{w_1, w_2, \dots, w_n\}) \in p \\ & \wedge w_1 = FIRST_k(\beta x_1), \\ & \wedge w_2 = FIRST_k(\beta x_2), \\ & \dots \\ & \wedge w_n = FIRST_k(\beta x_n)\} \cup \\ & \{(A \rightarrow \alpha X \bullet, lookaheads) \mid (A \rightarrow \alpha \bullet X, lookaheads) \in p\} \end{aligned}$$

e  $F$  é uma função bijetora que mapeia um estado no respectivo conjunto de itens, excluindo o conjunto vazio.

**Definição 6.**

$$PRED(q, \alpha) = \{p \mid GOTO_k(p, \alpha) = q\}$$

Considerando  $\alpha = X_1 X_2 \dots X_n$ ,  $PRED$  retorna os estados predecessores de  $q$  con- tanto que  $GOTO_k(\dots(GOTO_k(p, X_1), X_2), \dots), X_n)$  seja definido. No caso de  $n = 0$ ,  $PRED(q, \lambda) = \{q\}$ .

**Definição 7.** O autômato  $LRA_0$  é composto pelos seguintes componentes:

$$\begin{aligned} M_0 = & \{F^{-1}(CLOSURE(\{S' \rightarrow \bullet S\}))\} \cup \\ & \{F^{-1}(CLOSURE(F(q))) \mid q \in SUCC(p) \wedge p \in M_0\} \end{aligned}$$

A função  $SUCC$  é definida por

$$SUCC(p) = \{F^{-1}(ADVANCE(p, X)) \mid X \in V\}$$

O estado inicial  $IS$  é dado por  $IS = F^{-1}(CLOSURE(\{S' \rightarrow \bullet S\}))$  e  $\forall w \in \Sigma^*$  e  $X \in V$ , tem-se:

$$RED_k(q, w) = \{A \rightarrow \gamma \mid A \rightarrow \gamma \bullet \in F(q)\}$$

Observe que a função de redução é definida independentemente de  $w$ , o que está em concordância com o valor de  $k = 0$ . A função  $F$  garante uma correspondência um para um entre estados e conjunto de itens. Para facilitar a leitura, todas as ocorrências de  $F$  e  $F^{-1}$  serão omitidas e a distinção entre um estado e seu conjunto de itens dar-se-á pelo contexto em que são utilizadas.

**Definição 8.** O autômato de um analisador  $LALR(k)$  é uma sêxtupla:

$$LALRA_k = (M_0, V, P, IS, GOTO_0, RED_k)$$

onde  $M_0$ ,  $IS$ ,  $GOTO_0$  são como em  $LRA_0$  e  $V$  e  $P$  como em  $G$ . A função de redução é dada por

$$RED_k(q, w) = \{A \rightarrow \gamma \mid w \in LA_k(q, A \rightarrow \gamma \bullet)\}$$

onde  $LA_k$  é definida pelo seguinte teorema:

**Teorema 1.**

$$LA_k(q, A \rightarrow \gamma) = \{w \in FIRST_k(z) \mid S \xrightarrow[rm]{*} \alpha Az \wedge \alpha \gamma \text{ access } q\}$$

Diz-se que uma forma sentencial  $\xi$  *access*  $q$  quando  $PRED(q, \xi) \neq \emptyset$ . A prova deste teorema é fornecida por DeRemer e Pennello em [DeRemer e Pennello, 1982].

## 3.2 Classificação de Conflitos

Conflitos em gramáticas não LALR( $k$ ) surgem em um estado  $q$  quando pelo menos umas das seguintes condições é satisfeita:

1.  $RED_k(q, w) \geq 2$ : dado o *string*  $w$  de *lookahead*, o autômato  $LALRA_k$  possui mais de uma produção possível de ser reduzida. Isto caracteriza um conflito *reduce/reduce*;
2.  $RED_k(q, w) \geq 1 \wedge \exists A \rightarrow \alpha \bullet \beta \wedge w \in FIRST_k(\beta)$ : na presença de  $w$ , o autômato  $LALRA_k$  possui uma ação de redução e existe pelo menos um item em  $q$  com o marcador anterior a uma forma sentencial  $\beta$ , tal que  $FIRST_k(\beta)$  contém  $w$ . Quando isto ocorre tem-se um conflito *shift/reduce*.

**Definição 9.** Um estado que possui pelo menos um conflito é denominado *inconsistente*.

**Definição 10.** Uma gramática é LALR( $k$ ) se o autômato correspondente não possui estados inconsistentes.

Um conflito em uma gramática não LALR( $k$ ) é causado por ambigüidade ou falta de contexto à direita, resultando em cinco possíveis situações, conforme mostrado na Figura 3.1.

Os conflitos de ambigüidade definem a classe de conflitos oriundos da utilização de regras gramaticais que resultam em uma ou mais árvores de derivação para um dado *string*. Estes conflitos simplesmente não podem ser resolvidos aumentando-se o valor de  $k$ , pois não existe um valor de  $k$  (ou  $k = \infty$ ) tal que a gramática se torne LALR( $k$ ).

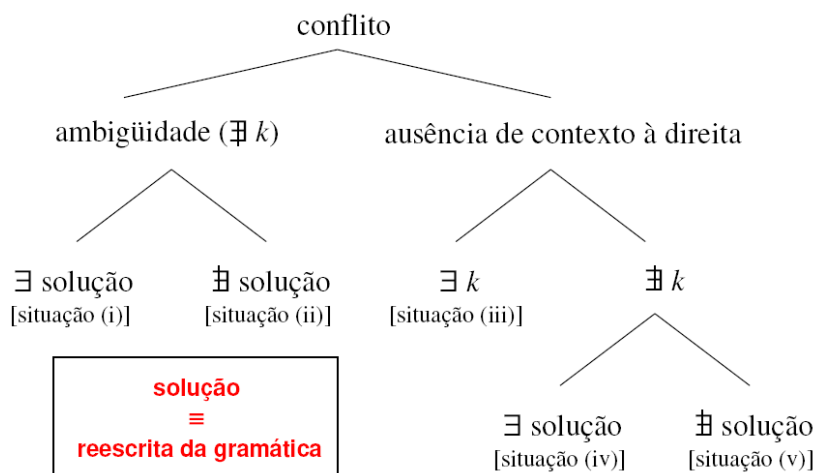


Figura 3.1: Situações em que um dado conflito ocorre.

Mas alguns dos conflitos de ambigüidade podem ser resolvidos pela reescrita de algumas regras da gramática de modo a torná-la LALR( $k$ ), de acordo com o valor de  $k$  utilizado pelo gerador de analisador sintático em questão. Esses conflitos formam a primeira situação da classificação proposta. Um exemplo de conflito bastante conhecido que pertence a esta categoria é a construção ambígua *dangling-else*, exibida a seguir:

$$\begin{array}{lcl}
 stmt & \rightarrow & if-stmt \\
 & | & other-stmt \\
 if-stmt & \rightarrow & \mathbf{if} \ exp \ \mathbf{then} \ stmt \ else-stmt \\
 else-stmt & \rightarrow & \mathbf{else} \ stmt \\
 & | & \lambda
 \end{array}$$

Sabe-se que essa construção sintática pode ser expressa utilizando-se um conjunto de regras não ambíguas [Aho e Ullman, 1972] equivalentes às anteriores, consultando-se apenas um *lookahead*:

$$\begin{array}{lcl}
 smt & \rightarrow & matched-stmt \\
 & | & unmatched-stmt \\
 matched-stmt & \rightarrow & matched-if \\
 & | & other-stmt \\
 matched-if & \rightarrow & \mathbf{if} \ exp \ \mathbf{then} \ matched-stmt \ \mathbf{else} \ matched-stmt \\
 unmatched-stmt & \rightarrow & \mathbf{if} \ exp \ \mathbf{then} \ stmt \\
 & | & \mathbf{if} \ exp \ \mathbf{then} \ matched-stmt \ \mathbf{else} \ unmatched-stmt
 \end{array}$$

Outros conflitos de ambigüidade, ao contrário, não podem ser removidos de uma gramática sem a alteração da linguagem correspondente. Isto se dá devido a existência de construções sintáticas inerentemente ambíguas. Conflitos deste tipo constituem a

segunda situação dentre as definidas pela classificação. Um exemplo de conflito nesta categoria é o conjunto de regras para descrição da linguagem  $L = \{a^m b^n c^k \mid m = n \vee n = k\}$ :

$$\begin{array}{l}
 S \quad \rightarrow \quad S_1 C_1 \\
 \quad \quad | \quad A_1 S_2 \\
 S_1 \quad \rightarrow \quad A_2 S_1 B \\
 \quad \quad | \quad \lambda \\
 A_1 \quad \rightarrow \quad A_1 \mathbf{a} \\
 \quad \quad | \quad \mathbf{a} \\
 A_2 \quad \rightarrow \quad \mathbf{a} \\
 B \quad \rightarrow \quad \mathbf{b} \\
 S_2 \quad \rightarrow \quad B S_2 C_2 \\
 \quad \quad | \quad \lambda \\
 C_1 \quad \rightarrow \quad C_1 \mathbf{c} \\
 \quad \quad | \quad \mathbf{c} \\
 C_2 \quad \rightarrow \quad \mathbf{c}
 \end{array}$$

Os conflitos causados pela falta de contexto à direita ocorrem devido a uma quantidade insuficiente de *lookaheads*. Uma solução direta é aumentar o valor de  $k$  utilizado pelo gerador. Conflitos que podem ser resolvidos desta forma constituem a terceira situação da classificação. Como exemplo de um conflito removido pelo aumento do valor de  $k$ , considere o fragmento da gramática de Notus mostrado na Figura 3.2:

<i>declaration</i>	$\rightarrow$	<i>visibility exportable-declaration</i>
		<i>non-exportable-declaration</i>
<i>non-exportable-declaration</i>	$\rightarrow$	<i>function-definition</i>
<i>visibility</i>	$\rightarrow$	<b>public</b>
		<b>private</b>
		$\lambda$
<i>exportable-declaration</i>	$\rightarrow$	<i>syntatic-domain</i>
<i>syntatic-domain</i>	$\rightarrow$	<b>domain-id</b> = <i>domain-exp</i>
<i>function-definition</i>	$\rightarrow$	<i>temp4</i>
<i>temp4</i>	$\rightarrow$	<i>temp5 id</i>
<i>temp5</i>	$\rightarrow$	<b>domain-id</b> “.”

Figura 3.2: Fragmento da linguagem Notus.



Para  $k = 1$ , tem-se um conflito *shift/reduce* entre os itens:

$$\begin{aligned} temp5 &\rightarrow \bullet\text{domain-id} \text{ ".} \\ visibility &\rightarrow \lambda\bullet \end{aligned}$$

contidos em um estado  $q$ . O conflito ocorre pois **domain-id** pertence a  $LA_1(q, visibility \rightarrow \lambda)$ . No entanto, a gramática é LALR(2), já que os *tokens* após **domain-id** são:

1. = (igual): este símbolo é originário da produção

$$syntactic-domain \rightarrow \text{domain-id} = domain-exp$$

2. "." (ponto): este símbolo provém da produção

$$temp5 \rightarrow \text{domain-id} = \text{"."}$$

Entretanto, mesmo quando não existe ambigüidade, ocorrem casos para os quais uma quantidade infinita de *lookaheads* é necessária - situação (iv). Para conflitos decorrentes desta situação, a solução é a reescrita da gramática. Considere, por exemplo, a linguagem  $L = (b^+a) \cup (b^+b)$ . Uma possível gramática não ambígua para  $L$  é:

$$\begin{aligned} S &\rightarrow A \\ &\quad | B \\ A &\rightarrow B_1 \mathbf{a} \\ B &\rightarrow B_2 \mathbf{b} \\ B_1 &\rightarrow B_1 \mathbf{b} \\ &\quad | \mathbf{b} \\ B_2 &\rightarrow B_2 \mathbf{b} \\ &\quad | \mathbf{b} \end{aligned}$$

Para esta gramática, o gerador LALR( $k$ ) reporta um conflito *reduce/reduce* envolvendo os itens:

$$\begin{aligned} B1 &\rightarrow \mathbf{b}\bullet \\ B2 &\rightarrow \mathbf{b}\bullet \end{aligned}$$

sendo  $\mathbf{b}$  o símbolo de conflito. Aumentando-se o valor de  $k$  por um fator  $n$ , o conflito permanece para o *string* de *lookahead*  $\mathbf{b}^{k*n}$ . Assim, não existe um  $k$  finito capaz de resolver o conflito. Para este exemplo simples, a gramática pode ser reescrita de forma a eliminar o conflito. Ressalta-se, no entanto, que este tipo de solução não é sempre possível. Neste caso, o conflito em questão é decorrente da situação (v).

### 3.3 Fronteira entre Conflitos em LR e LALR

As situações mencionadas na Seção 3.2 cobrem todo o leque de possíveis causas de conflitos na obtenção de analisadores LALR( $k$ ). No entanto, é importante ressaltar que existem conflitos que ocorrem somente na obtenção de analisadores LALR( $k$ ), não reportados na construção de analisadores LR( $k$ ).

Para explicar esses conflitos específicos a LALR, é necessário antes apresentar um algoritmo para a construção do autômato  $LALRA_k$ . Tal algoritmo inicialmente constrói o autômato  $LRA_k$ , pela aplicação da função  $CLOSURE$ , conforme definido na Seção 3.1. Em seguida, faz a junção dos estados que possuem o mesmo núcleo, resultando no conjunto de estados  $M_0$ . Em cada estado  $q$  resultante da junção dos estados  $q_1q_2\dots q_n$  em  $LRA_k$ , o conjunto de *lookaheads* de um item  $A \rightarrow \omega \bullet$  em  $q$  é dado por:

$$LA_k(q, A \rightarrow \omega \bullet) = \bigcup_{i=1}^n \{w \mid w \in la \wedge (A \rightarrow \omega \bullet, la) \in q_i\}$$

No autômato  $LALRA_k$  resultante podem existir conflitos *reduce/reduce* originalmente não presentes em  $LRA_k$ . Esses conflitos, específicos a analisadores LALR, são identificados pela inexistência de um mesmo contexto à esquerda  $\xi$ , isto é, uma forma sentencial obtida pela concatenação de cada símbolo de entrada dos estados no caminho de  $IS_{LALRA_k}$  a  $q$ , o estado inconsistente. Disto resulta o teorema a seguir:

**Teorema 2.** *Para todo conflito reduce/reduce em um estado inconsistente  $q$  no autômato  $LALRA_k$ , para  $k \geq 1$ , resultante da junção de dois ou mais estados do autômato  $LRA_k$ , tem-se pelo menos dois caminhos em  $LALRA_k$  a partir do estado inicial  $IS_{LALRA_k}$  até  $q$  e todos eles são diferentes entre si. Isto resulta em diferentes contextos à esquerda.*

**Prova 1.** *Seja  $p$  um estado no autômato  $LALR_k$  resultante da junção de dois estados  $p_1$  e  $p_2$  em  $M_k$ , com  $k \geq 1$ . Pela propriedade de que  $LRA_k$  é um autômato finito determinístico (AFD) que reconhece os prefixos viáveis de uma gramática  $G$  [Knuth, 1965], segue pela propriedade de AFDs que todos os caminhos do estado inicial  $IS_{LRA_k}$  até  $p_1$  e  $p_2$  são diferentes entre si. Como todo estado em  $M_k$  é alcançável a partir de  $IS_{LRA_k}$ , existe pelo menos dois caminhos  $\alpha_1$  e  $\alpha_2$  de  $IS_{LRA_k}$  a  $p_1$  e  $p_2$  respectivamente. Sabendo-se que o conjunto das linguagens LALR( $k$ ) é subconjunto próprio das LR( $k$ ), o autômato  $LALRA_k$  deve reconhecer a mesma linguagem que o autômato  $LRA_k$ . Assim, percorrendo-se  $\alpha_1$  e  $\alpha_2$  a partir de  $IS_{LALRA_k}$  o único destino possível é o estado  $p$ , o que resulta em contextos à esquerda diferentes entre si, pois  $\alpha_1 \neq \alpha_2$ .*

Desse teorema, resulta o seguinte corolário:

**Corolário 1.** *Conflitos reduce/reduce exclusivos a LALR não representam conflitos de ambigüidade.*

**Prova 2.** *Pelo teorema anterior, a existência de pelo menos dois contextos a esquerda diferentes entre si faz com que seja impossível a obtenção de strings iguais para duas árvores de derivação distintas entre si.*

### 3.4 Cômputo da Função $LA_k$

Ao divulgar a técnica de análise sintática LALR( $k$ ), DeRemer não definiu um algoritmo para cálculo dos *lookaheads* a partir do autômato  $LRA_0$ . Dois anos mais tarde, Lalonde [LaLonde et al., 1971] apresentou um algoritmo para o caso de  $k = 1$ . Muitos outros algoritmos para cômputo de  $LA_1$  foram posteriormente publicados, incluindo o método utilizado pelo gerador YACC [Johnson, 1979]. No entanto, esses algoritmos eram muito ineficientes.

Na década de 80, um enorme progresso foi obtido. Em 1982, DeRemer e Pennello [DeRemer e Pennello, 1982] publicaram um eficiente algoritmo para cômputo de *lookaheads* de tamanho igual a 1 [Charles, 1991]. No entanto, os autores não o generalizaram para o caso de  $k > 1$ . Alguns anos mais tarde, Charles [Charles, 1991] propôs um conjunto de equações e algoritmos para o caso de  $k \geq 1$ , baseando-se em parte nas relações definidas por DeRemer e Pennello. Essas duas técnicas são descritas a seguir.

#### 3.4.1 Algoritmo de DeRemer e Pennello ( $k = 1$ )

O cálculo dos *lookaheads* pelo método de DeRemer e Pennello basea-se em relações estabelecidas nas transições sobre não terminais do autômato  $LRA_0$ . A prova de todos os teoremas apresentados são discutidas e apresentadas em [DeRemer e Pennello, 1982].

Transições em  $LRA_0$  são triplas em  $(M_0 \times \Sigma \times M_0) \cup (M_0 \times N \times M_0)$ , onde o primeiro componente representa o estado de origem, o segundo o símbolo de transição e o terceiro o estado de destino. As triplas pertencentes ao conjunto  $M_0 \times \Sigma \times M_0$  são denominadas *transições sobre terminais*, enquanto o conjunto  $M_0 \times N \times M_0$  contém as *transições sobre não terminais*. Por questões de simplicidade, transições serão apresentadas como duplas quando o estado de destino não for relevante.

Como o valor de  $k$  é fixado em 1, a função  $RED_1$  é redefinida da seguinte forma:

**Definição 11.**

$$RED_1(q, a) = \{A \rightarrow \gamma \mid a \in LA_1(q, A \rightarrow \gamma \bullet)\}$$

Os símbolos que podem seguir um não terminal  $A$  em uma forma sentencial de prefixo  $\alpha$ , que precede  $A$  e acessa o estado  $p$ , são expressos pela função  $FOLLOW_1$ :

**Definição 12.**

$$FOLLOW_1(p, A) = \{t \in \Sigma \mid S \xrightarrow{\pm} \alpha Atz \text{ e } \alpha \text{ access } p\}$$

O cálculo de  $FOLLOW_1(p, A)$  é dado em função de três situações possíveis. A primeira delas ocorre quando se tem um caminho de um estado  $p$  a um estado  $q$  sobre  $\beta$  ( $p \in PRED(q, \beta)$ ),  $p$  contém um item  $C \rightarrow \theta \bullet B\eta$  e  $B \rightarrow \beta A\gamma \in P$ . A partir deste ponto, o analisador sintático alcança em algum momento o estado  $q = GOTO_k(p, \beta)$ . Se neste estado existir um item  $B \rightarrow \beta \bullet A\gamma$ , tal que  $\gamma \xrightarrow{*} \lambda$ ,  $B$  deriva  $\beta A$ , o que implica que os *lookaheads* de  $B$  em  $FOLLOW(p, B)$  estão contidos em  $FOLLOW(q, A)$ . Assim,

$$FOLLOW_1(p, B) \subseteq FOLLOW_1(q, A)$$

se

$$C \rightarrow \theta \bullet B\eta \in p, p \in PRED(q, \beta), B \rightarrow \beta A\gamma \in P, \text{ e } \gamma \xrightarrow{*} \lambda$$

Disto resulta a próxima definição.

**Definição 13.**

$$(q, A) \text{ includes } (p, B) \text{ sss } FOLLOW(p, B) \subseteq FOLLOW(q, A)$$

A Figura 3.3 esquematiza a relação de inclusão.

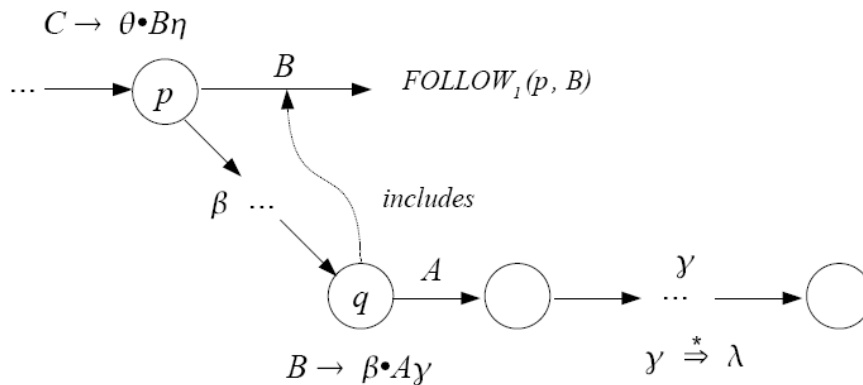


Figura 3.3: Relacionamento entre os conjuntos  $FOLLOW_1$  via relação *includes*.

A segunda possibilidade ocorre quando se tem um item em  $C \rightarrow \theta A \bullet t\eta$  em  $GOTO_0(p, A)$ . Neste caso, diz-se que  $t$  é lido diretamente. Disto, resulta a função  $DR$ , dada pelo teorema 3.

**Teorema 3.**

$$DR(p, A) = \{t \in \Sigma \mid GOTO_0(GOTO_0(p, A), t) \neq \Omega\}$$

A terceira e última situação ocorre quando se tem duas transições  $(p, A, r_0)$  e  $(r_0, B, r_1)$ , tal que  $B \xrightarrow{*} \lambda$ . Neste caso, os *lookaheads* de  $B$  também são *lookaheads* de  $A$ , pois  $A \Rightarrow B$ . Para modelar isto, apresenta-se a relação *reads*, responsável por capturar as leituras realizadas por estados alcançados por  $p$  via seqüência de transições sobre símbolos não terminais anuláveis:

**Definição 14.**

$$(p, A) \text{ reads } (r, B) \text{ sss } GOTO_0(p, A) = r \text{ e } B \xrightarrow{*} \lambda$$

A Figura 3.4 esquematiza esta relação.

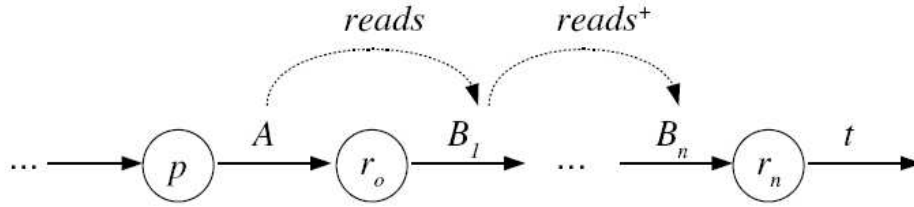


Figura 3.4: Representação da relação *reads*.

A função  $READ_1(p, A)$  obtém os *lookaheads* passíveis de leitura direta ou indireta, a partir do estado  $GOTO_0(p, A)$ :

**Teorema 4.**

$$READ_1(p, A) = DR(p, A) \cup \bigcup \{READ_1(q, B) \mid (p, A) \text{ reads } (q, B)\}$$

Com isto posto, os dois teoremas a seguir fornecem a fórmula fechada para o cômputo de  $FOLLOW_1$  e  $LA_1$ .

**Teorema 5.**

$$FOLLOW_1(p, A) = READ_1(p, A) \cup \bigcup \{FOLLOW_1(q, B) \mid (p, A) \text{ includes } (q, B)\}$$

**Teorema 6.**

$$LA_1(q, A \rightarrow \alpha) = \bigcup \{FOLLOW_1(p, A) \mid p \in PRED(q, \alpha)\}$$

Dos teoremas e definições apresentados, percebe-se que o problema de cômputo de  $LA_1$  foi dividido em quatro subproblemas. Em ordem inversa: os conjuntos  $LA_1$  são

obtidos a partir de  $FOLLOW_1$ , estes computados a partir dos conjuntos  $READ_1$ , que por sua vez contém os valores definidos em  $DR$ .

Em 1977, Eve e Kurki-Suonio [Eve e Kurki-Suonio, 1977] apresentaram um algoritmo para cálculo do fecho transitivo de uma relação arbitrária, baseado no trabalho realizado por Tarjan [Tarjan, 1972] para cálculo de componentes fortemente conectados (CFC) em dígrafos. Um componente fortemente conectado em um grafo dirigido é um subgrafo maximal no qual todos os vértices são alcançáveis entre si. Se o número de vértices no subgrafo for igual a 1, diz-se que o componente fortemente conectado é *trivial*. DeRemer e Pennello adaptaram o algoritmo de Eve e Kurki para o cômputo de  $FOLLOW_1$  e  $READ_1$ . Antes de apresentar a solução, no entanto, segue a formulação geral do problema.

Seja  $R$  uma relação sobre um conjunto  $X$ . Sejam  $F$  e  $F'$  duas funções definidas sobre os elementos em  $X$ , tal que  $\forall x \in X$ , tem-se:

$$F(x) = F'(x) \cup \bigcup \{F(y) \mid xRy\}$$

onde  $F'(x)$  é sempre definida.

Se o grafo correspondente à relação  $R$  não contiver ciclos, um algoritmo para calcular  $F$  é facilmente obtido. Se, no entanto, existirem CFCs, incorre-se no risco de recursão infinita, além de um alto custo computacional, devido ao cômputo de uniões desnecessárias. Para diminuir o número de uniões, faz-se uso da seguinte propriedade:

*Propriedade 1.* Se  $x$  e  $y$  são membros de um CFC, então  $xR^*y$  e  $yR^*x$ . Com isto, se  $F(x)$  for da forma  $F(x) = F'(x) \cup \bigcup \{F(y) \mid xRy\}$ , tem-se que  $F(x) \subseteq F(y)$  e  $F(y) \subseteq F(x)$ ; portanto  $F(x) = F(y)$ .

O objetivo do algoritmo proposto por DeRemer e Pennello é realizar duas tarefas. A primeira delas é construir um novo dígrafo  $G'$ , de forma que os vértices de um CFC  $C$  no grafo original  $G$  sejam agregados em um único vértice  $\sigma$  em  $G'$ , tal que  $F(\sigma) = \bigcup \{F'(x) \mid x \in C\}$ . Como todos os CFCs em  $G$  agora constituem vértices em  $G'$ , este é garantidamente livre de ciclos e portanto de CFCs. A segunda tarefa é a realização de uma busca recursiva em  $G'$ , propagando os valores de  $F(\sigma)$  a todo vértice  $x$  em  $C$  agregado a  $\sigma$  em  $G'$ . A eficiência do algoritmo de DeRemer e Pennello é obtida pela realização de ambas as tarefas em uma única busca em  $G$ , sem a construção explícita de  $G'$ . O algoritmo para o cálculo de  $F$  é apresentado na Figura 3.5.

Assume-se a existência das variáveis globais  $F, F', stack$  e  $N$ . Os vetores  $F, F', N$  são indexados por elementos em  $X$ , onde inicialmente  $F[x] \leftarrow F'[x] \leftarrow \emptyset$  e  $N[x] \leftarrow \emptyset$ ,  $\forall x \in X$ ;  $S$  é uma pilha vazia. Para uma dada pilha  $stack$ , tem-se as operações de

```

DIGRAPH()
1  for  $x \in X$ 
2  do  $N[x] \leftarrow 0$ 
3      $F[x] \leftarrow F'[x] \leftarrow \emptyset$ 
4  for  $y \in X$ 
5  do if  $N[x] = 0$ 
6     then TRAVERSE( $y$ )

TRAVERSE( $x$ )
1  PUSH( $stack, x$ )
2   $d \leftarrow |stack|$ 
3   $N[x] \leftarrow d$ 
4   $F[x] \leftarrow F'[x]$ 
5  for  $y \in X \mid xRy$ 
6  do if  $N[y] = 0$ 
7     then TRAVERSE( $y$ )
8      $N[x] \leftarrow MIN(n[x], n[y])$ 
9      $F[x] \leftarrow F[x] \cup F[y]$ 
10 if  $N[x] = d$ 
11 then  $y \leftarrow POP(stack)$ 
12      $N[y] \leftarrow \infty$ 
13     while  $y \neq x$ 
14     do  $F[y] \leftarrow F[y] \cup F[x]$ 
15          $y \leftarrow POP(stack)$ 
16      $N[y] \leftarrow \infty$ 

```

Figura 3.5: Algoritmo de cômputo de  $F$ .

empilhamento (+), consulta ao topo ( $TOP$ ), remoção do topo ( $POP$ ) e obtenção do número de elementos empilhados ( $|stack|$ ).

$TRAVERSE$  é um procedimento recursivo que recebe como parâmetro a pilha e o vértice  $x$  para o qual se deseja calcular o valor de  $F$ . O algoritmo inicia pela inserção de  $x$  em  $S$ . Em seguida, armazena em  $N[x]$  o tamanho da pilha e atribui  $F'[x]$  a  $F[x]$ . O vetor  $N$  serve a três propósitos: (i) indica se um vértice não foi visitado ( $N[x] = 0$ ); (ii) associa um número inteiro positivo aos vértices cujo o cálculo de  $F$  ainda não finalizado ( $0 < N[x] < \infty$ ); (iii) marca quais vértices já tiveram o valor de  $F$  calculado ( $N[x] = \infty$ ). No próximo passo, o algoritmo chama recursivamente o procedimento  $TRAVERSE$  para todo vértice  $y$  relacionado a  $x$ , tal que  $y$  é um vértice não visitado. Na linha 8,  $N[y]$  é necessariamente diferente de zero. Se  $N[y] < N[x]$ , então a relação  $R$  contém um ciclo que inclui os vértices  $x$  e  $y$ , pois neste caso,  $y$  foi visitado em um momento anterior ao de  $x$  e a computação de  $x$  não pode terminar antes da computação de  $y$ . Com isto,  $x$  só poderá ser desempilhado quando o valor

de  $F(y)$  estiver disponível. A verificação na linha 10 do procedimento *TRAVERSE* garante isto. Se, no entanto, o valor de  $N[x]$  não tiver sido alterado,  $x$  é o primeiro nodo visitado do CFC, seja ele trivial ou não. Neste ponto, o valor  $F(x)$  está disponível, pois todos os outros vértices nos quais  $x$  se relaciona já foram visitados. Procedese então, desempilhando cada vértice  $y$  do CFC armazenado na pilha, propagando o valor de  $F(x)$  a todos eles, até que se tenha  $y$  seja igual a  $x$ .

O algoritmo *TRAVERSE* é utilizado no cálculo de  $READ_1$ , tomando-se  $X$  igual ao conjunto de transições não terminais definidas em  $GOTO_0$ ,  $F'$  igual a  $DR$  e  $R$  igual à relação *reads*. No caso de  $FOLLOW_1$ , tem-se  $F'$  igual a  $READ_1$  e a relação *includes* como  $R$ .  $X$  permanece como o conjunto de transições sobre não terminais.

Para exemplificar o cálculo de  $LA_1$ , considere a seguinte gramática da Figura 3.6

$$\begin{array}{l} exp \rightarrow exp + exp \\ | \quad exp - exp \\ | \quad ( exp ) \\ | \quad \mathbf{num} \end{array}$$

Figura 3.6: Gramática de expressões.

Os estados do autômato LR(0) referente à gramática apresentada, seus itens e suas transições são dados a seguir:

$S_0$ )

$$\begin{array}{l} S' \rightarrow \bullet exp \$ \\ exp \rightarrow \bullet exp + exp \\ exp \rightarrow \bullet exp - exp \\ exp \rightarrow \bullet ( exp ) \\ exp \rightarrow \bullet \mathbf{num} \end{array}$$

Transições:

- \* sobre *exp* vai para o estado 1
- \* sobre ( vai para o estado 2
- \* sobre **num** vai para o estado 3

$S_1$ )

$$\begin{array}{l} S' \rightarrow exp \bullet \$ \\ exp \rightarrow exp \bullet + exp \\ exp \rightarrow exp \bullet - exp \end{array}$$

Transições:



\* sobre  $\$$  vai para o estado 4

\* sobre  $-$  vai para o estado 5

\* sobre  $+$  vai para o estado 6

$S_2$ )

$exp \rightarrow (\bullet exp)$

$exp \rightarrow \bullet exp - exp$

$exp \rightarrow \bullet exp + exp$

$exp \rightarrow \bullet \mathbf{num}$

$exp \rightarrow \bullet (exp)$

Transições:

\* sobre  $($  vai para o estado 2

\* sobre  $\mathbf{num}$  vai para o estado 3

\* sobre  $\mathbf{exp}$  vai para o estado 7

$S_3$ )

$exp \rightarrow \mathbf{num} \bullet$

$S_4$ )

$S' \rightarrow exp \$ \bullet$

$S_5$ )

$exp \rightarrow exp - \bullet exp$

$exp \rightarrow \bullet exp - exp$

$exp \rightarrow \bullet exp + exp$

$exp \rightarrow \bullet \mathbf{num}$

$exp \rightarrow \bullet (exp)$

Transições:

\* sobre  $($  vai para o estado 2

\* sobre  $\mathbf{num}$  vai para o estado 3

\* sobre  $exp$  vai para o estado 8

$S_6$ ) $exp \rightarrow exp + \bullet exp$  $exp \rightarrow \bullet exp - exp$  $exp \rightarrow \bullet exp + exp$  $exp \rightarrow \bullet \mathbf{num}$  $exp \rightarrow \bullet ( exp )$ 

\* sobre ( vai para o estado 2

\* sobre **num** vai para o estado 3\* sobre  $exp$  vai para o estado 9 $S_7$ ) $exp \rightarrow ( exp \bullet )$  $exp \rightarrow exp \bullet + exp$  $exp \rightarrow exp \bullet - exp$ 

Transições:

\* sobre  $-$  vai para o estado 5\* sobre  $+$  vai para o estado 6\* sobre  $)$  vai para o estado 10 $S_8$ ) $exp \rightarrow exp - exp \bullet$  $exp \rightarrow exp \bullet + exp$  $exp \rightarrow exp \bullet - exp$ 

Transições:

\* sobre  $-$  vai para o estado 5\* sobre  $+$  vai para o estado 6 $S_9$ ) $exp \rightarrow exp + exp \bullet$  $exp \rightarrow exp \bullet + exp$  $exp \rightarrow exp \bullet - exp$ 

Transições:

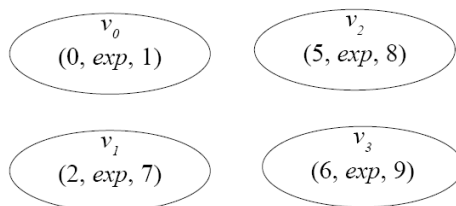


Figura 3.7: Grafo da relação *reads* construído a partir do autômato  $LRA_0$  referente à gramática da Figura 3.6.

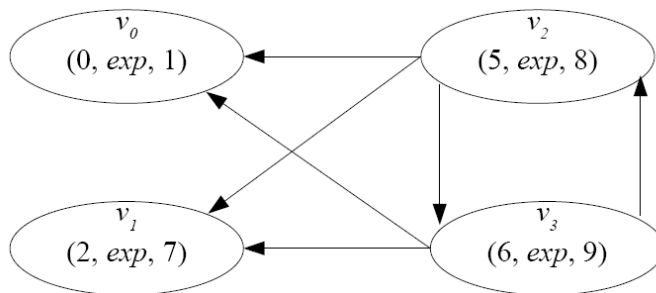


Figura 3.8: Grafo da relação *includes* construído a partir do autômato  $LRA_0$  referente à gramática da Figura 3.6

\* sobre  $-$  vai para o estado 5

\* sobre  $+$  vai para o estado 6

$S_{10}$ )

$exp \rightarrow (exp) \bullet$

O valor da função  $DR$  para cada transição sobre não terminal é facilmente obtida a partir do autômato  $LR(0)$ .

$$\begin{aligned} DR(0, exp, 1) &= \{-, +, \$\} & DR(5, exp, 8) &= \{-, +\} \\ DR(2, exp, 7) &= \{-, +, ,\} & DR(6, exp, 9) &= \{-, +\} \end{aligned}$$

O grafo referente à relação *reads* é exibido na Figura 3.7. A não existência de arestas está em conformidade com a gramática, pois esta não possui produções que derivam  $\lambda$ . Por este fato, ao aplicar o algoritmo *DIGRAPH* ao grafo da relação *reads*, obtêm-se os mesmos valores que os definidos em  $DR$ .

Para a relação de inclusão, cujo grafo é exibido na Figura 3.8, o algoritmo *DIGRAPH* retorna os seguintes conjuntos  $FOLLOW_1$ :

$$\begin{aligned} FOLLOW_1(0, exp, 1) &= READ_1(0, exp, 1) \\ &= \{-, +, \$\} \end{aligned}$$

$$\begin{aligned} FOLLOW_1(2, exp, 7) &= READ_1(2, exp, 7) \\ &= \{-, +, )\} \end{aligned}$$

$$\begin{aligned} FOLLOW_1(5, exp, 8) &= READ_1(5, exp, 8) \cup \\ &FOLLOW_1(0, exp, 1) \cup \\ &FOLLOW_1(2, exp, 7) \cup \\ &FOLLOW_1(6, exp, 9) \cup \\ &= \{-, +, \$, )\} \end{aligned}$$

$$\begin{aligned} FOLLOW_1(6, exp, 9) &= READ_1(6, exp, 9) \cup \\ &FOLLOW_1(0, exp, 1) \cup \\ &FOLLOW_1(2, exp, 7) \cup \\ &FOLLOW_1(5, exp, 8) \cup \\ &= \{-, +, \$, )\} \end{aligned}$$

Observe a existência de um CFC formado pelos vértices  $v_2$  e  $v_3$  no grafo da relação *includes*. Ao executar o algoritmo *DIGRAPH* sobre esse grafo, no momento em que  $v_2$  é desempilhado (linha 11 do procedimento *TRAVERSE*), atribui-se o valor de  $FOLLOW_1(5, exp, 8)$  a  $FOLLOW_1(6, exp, 9)$  e faz-se  $N[v_2] = N[v_3] = \infty$ , o que impede que *TRAVERSE* seja posteriormente chamado para calcular os *lookaheads* de  $v_3$ .

Para obter o valor de  $LA_1$  utiliza-se a fórmula enunciada no Teorema 6. Antes, no entanto, é necessário os valores de *PRED*.

$$\begin{aligned} PRED(3, \mathbf{num}) &= \{0, 1, 4, 5\} & PRED(4, exp\$) &= \emptyset \\ PRED(8, exp - exp) &= \{0, 1, 5\} & PRED(8, exp + exp) &= \{0, 1, 4\} \\ PRED(10, (exp)) &= \{0, 4, 5\} \end{aligned}$$

Com isto, tem-se:

$$\begin{aligned} LA_1(3, exp \rightarrow \mathbf{num}) &= \{-, +, ), \$\} & LA_1(4, S' \rightarrow exp\$) &= \emptyset \\ LA_1(8, exp \rightarrow exp - exp) &= \{-, +, ), \$\} & LA_1(9, exp \rightarrow exp + exp) &= \{-, +, ), \$\} \\ LA_1(9, exp \rightarrow (exp)) &= \{-, +, ), \$\} \end{aligned}$$

### 3.4.2 Algoritmo de Charles ( $k \geq 1$ )

Como parte de sua tese de doutorado, Charles [Charles, 1991] propôs um algoritmo para cômputo de  $LA_k$  utilizando em parte a proposta de DeRemer e Pennello.

As funções  $READ_1$  e  $FOLLOW_1$  são alteradas de forma a acomodar valores de  $k \geq 1$ , conforme definido pelos dois lemas a seguir:

**Lema 1.**

$$READ_k(p, A) = \{w \mid w \in FIRST_k(\beta), |w| = k, B \rightarrow \alpha \bullet A\beta \in p\}$$

**Lema 2.**

$$\begin{aligned} FOLLOW_k(p, A) = & READ_k(p, A) \\ & \cup \cup \{FOLLOW_k(p', B) \mid (p, A) \text{ includes } (p', B)\} \\ & \cup \cup \{\{w\}.FOLLOW_{k-|w|}(p', B) \mid \\ & \quad (w, B \rightarrow \alpha \bullet A\beta) \in SHORT_k(p, A), \\ & \quad p' \in PRED(p, \alpha), \\ & \quad B \neq S, \\ & \quad w \neq x\$\} \end{aligned}$$

onde

$$SHORT_k(p, A) = \{(w, B \rightarrow \alpha \bullet A\beta) \mid w \in FIRST_k(\beta), 0 < |w| < k, B \rightarrow \alpha \bullet A\beta \in p\}$$

O operador infixado “.” é definido por:

$$la_1.la_2 = \{xy \mid x \in la_1 \wedge y \in la_2\}$$

$READ_k$  define os *strings* em  $FIRST_k$  com tamanho exatamente igual a  $k$ . A partir disto,  $FOLLOW_k(p, A)$  é dado pelo resultado de  $READ_k(p, A)$  com a união dos conjuntos  $FOLLOW_k(p', B)$ , desde que exista uma aresta entre  $(p, A)$  e  $(p', B)$  no grafo de inclusão. Adicionalmente, os *strings* em  $FIRST_k$  cujo tamanho é menor que  $k$ , são concatenados com os *strings* em  $FOLLOW_{k-|w|}$  das transições  $(p', B)$ , tal que  $p' \in PRED(q, \alpha)$ , de forma a serem incluídos em  $FOLLOW_k(p, A)$ . Este último caso é esquematizado na Figura 3.9.

Quando  $\alpha A\beta$  se torna o *handle*, faz-se a redução via o desempilhamento de  $|\alpha A\beta|$  estados. Neste ponto,  $p'$  aparece no topo da pilha. Como o *string*  $w$  derivado a partir de  $\beta$  possui tamanho menor que  $k$ , os *strings* de *lookahead* de tamanho  $k - |w|$  que seguem  $B$  a partir de  $p'$  são calculados e concatenados a  $w$ .

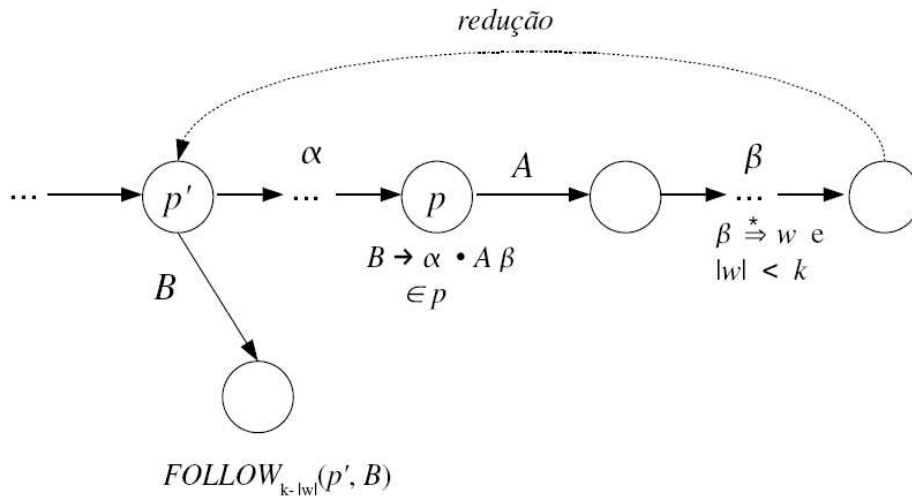


Figura 3.9: Esquemática da concatenação de  $w$  aos strings de lookaheads em  $FOLLOW_{k-|w|}(p', B)$ .

O autor define uma fórmula para cálculo dos conjuntos  $READ_k$ , a partir da qual um algoritmo pode ser diretamente obtido. A fórmula é baseada na simulação dos possíveis passos do autômato  $LRA_0$ . Antes de apresentá-la, no entanto, segue a definição de configuração:

**Definição 15.** Uma configuração do autômato  $LRA_0$  é uma dupla pertencente ao domínio  $M_0^+ \times V$ , constituída pela pilha de estados e um símbolo gramatical  $X$ . Se  $ts$  é o estado no topo da pilha, então  $GOTO_0(ts, X) \neq \Omega$ .

Como o autômato  $LRA_0$  de uma GLC  $G$  é um analisador sintático correto para  $G$ , embora possa não ser determinístico, qualquer string  $w$  lido no contexto de uma configuração  $(p_1 p_2 \dots p_n, X)$ , com  $n \geq 1$ , pode ser obtido pela seqüência de transições realizadas a partir de  $[p_1 p_2 \dots p_n q]$ , onde  $q = GOTO_0(p_n, X)$ . No estado  $q$ , três possibilidades são consideradas [Charles, 1991]:

1.  $q$  contém transições sobre símbolos terminais. Cada terminal  $a$  que é diretamente lido em  $q$  é um possível primeiro símbolo dos strings que podem ser lidos a partir do contexto  $(p_1 p_2 \dots p_n, X)$ . O conjunto de todos os sufixos de tamanho menor ou igual a  $k - 1$  que podem seguir  $a$  podem ser calculados de forma recursiva considerando o par  $(stack + [q], a)$ ;
2.  $q$  contém transições sobre símbolos não terminais anuláveis. Após  $q$  se tornar o estado no topo da pilha, o autômato  $LRA_0$  pode realizar uma transição sobre um não terminal sem realizar o consumo de nenhum token. Todos os string de tamanho menor ou igual a  $k$  que podem ser lidos após tal transição devem ser incluídos no resultado final;

3.  $p_n$  contém um ou mais itens da forma  $C \rightarrow \gamma \bullet X$ . Para cada um desses itens, após a transição sobre  $X$ ,  $LRA_0$  remove  $|\gamma X|$  elementos da pilha e realiza a transição sobre  $C$  a partir do estado no topo da pilha. Ao executar a redução, nenhum *token* é consumido. Assim, todo *string* de tamanho menor ou igual a  $k$  possível de ser lido a partir da nova configuração deve ser considerado.

Com isto, os *strings* de tamanho menor ou igual a  $k$ , indicados por  $k^*$ , lidos a partir de uma configuração  $(stack, X)$  são obtidos por:

$$\begin{aligned}
READ_{0^*}(stack, X) &= \{\lambda\} \\
READ_{k^*}(stack, X) &= \\
&\cup \{ \{a\} \cdot READ_{(k-1)^*}(stack + [q], a) \mid a \in DR(ts, X), q = GOTO_0(ts, X) \} \\
&\cup \{ \{READ_{k^*}(stack + [q], Y) \mid (ts, X) \text{ reads } (q, Y) \} \\
&\cup \{ \{READ_{k^*}(stack(1..(|stack| - |\gamma|)), C) \mid C \rightarrow \gamma \bullet X \in ts, |\gamma| + 1 < |stack| \}
\end{aligned} \tag{3.1}$$

onde  $ts$  é o estado no topo da pilha e  $stack(1..n)$  é a subpilha composta pelos  $n$  elementos a partir da base de  $stack$ . O algoritmo para cômputo dessa equação, apresentado na Figura 3.10, calcula os *strings* de *lookaheads* de forma incremental. A  $i$ -ésima execução do laço em  $READ_*$  calcula o  $i$ -ésimo símbolo de um *string* de *lookahead*  $w$ , tal que  $w \leq |k|$ . Cada símbolo é obtido por uma chamada a *READ-STEP*, que fornecida uma configuração de partida  $(p_1 p_2 \dots p_n, X)$ , simula os passos do autômato  $LRA_0$  de forma a encontrar todos os símbolos que podem ser lidos a partir de  $GOTO_0(p_n, X)$ . Todo símbolo  $a_{i+1}$  é obtido a partir da configuração que resultou no aparecimento de  $a_i$ .

Observe que os valores em  $SHORT_k$  podem ser facilmente obtidos. Se em algum momento, a partir de uma chamada inicial  $READ_*([p], A)$ , a simulação atingir uma configuração  $([pp_1 p_2 \dots p_n, X])$ , onde  $C \rightarrow \gamma \bullet X \in p_n$  e  $|\gamma|$  é da forma  $\eta A \theta$ , com  $n+1 < |\gamma|$  (*stack underflow*) ou  $|\gamma| = n$ , então o string  $w$  obtido possui tamanho menor que  $k$ . Neste caso, o item do par  $(w, item)$  em  $SHORT(p, A)$  é obtido movendo-se o ponto em  $C \rightarrow \gamma \bullet X \in p_n$   $n$  posições à esquerda.

O novo algoritmo para calcular  $READ_{k^*}$ , que permite a obtenção dos pares  $(w, item)$  em  $SHORT_k$  é mostrado na Figura 3.11.

Com isto,  $SHORT_k(p, A)$  e  $READ_k(p, A)$  são redefinidos por:

$$\begin{aligned}
SHORT_k(p, A) &= \{(item, w) \mid (item, w) \in READ_{2^*} \wedge item \neq nil\} \\
READ_k(p, A) &= \{w \mid (item, w) \in READ_{2^*} \wedge item = nil\}
\end{aligned}$$

Em [Charles, 1991], o autor argumenta que a equação do Lema 2 não pode ser calculada pelo algoritmo *DIGRAPH*, pois não está no formato  $F(x) = F'(y) \cup \{F(y) \mid xRy\}$ ,

```

READ*(stack, X, k)
1  if  $k = 0 \vee X = \$$ 
2    then return  $\{\lambda\}$ 
3   $rd \leftarrow \emptyset$ 
4  for  $(stk, a) \in \text{READ-STEP}(stack, X)$ 
5  do  $rd \leftarrow rd \cup \{ax \mid x \in \text{READ}_*(stk, a, k - 1)\}$ 
6  return  $rd$ 

READ-STEP(stack, X)
1   $configs \leftarrow \emptyset$ 
2   $ts \leftarrow \text{TOP}(stack)$ 
3   $q \leftarrow \text{GOTO}_0(ts, X)$ 
4  for  $Y \in V \mid \text{GOTO}_0(q, Y) \neq \Omega$ 
5  do if  $Y \xrightarrow{*} \lambda$ 
6    then  $configs \leftarrow configs \cup \text{READ-STEP}(stack + [q], Y)$ 
7    else if  $Y \in \Sigma$ 
8      then  $configs \leftarrow configs \cup \{(stack + [q], Y)\}$ 
9  for  $C \rightarrow \gamma \bullet X \in ts \mid C \neq S \wedge |\gamma| + 1 < |stack|$ 
10 do  $configs \leftarrow configs \cup \text{READ-STEP}(stack(1..(|stack| - |\gamma|)), C)$ 
11 return  $configs$ 

```

Figura 3.10: Algoritmo para cômputo de  $READ_{k*}$ .

o que é questionado em [Passos et al., 2007]. Charles descarta essa equação como base de um algoritmo para cômputo de  $FOLLOW_k$  e propõe um outro Lema, a partir do qual deriva um algoritmo:

**Lema 3.**

$$FOLLOW_k(p, A) = FOLLOW_{k*}([p], A)$$

onde

$$FOLLOW_{0*}(stack, X) = \{\lambda\}$$

$$FOLLOW_{k*}(stack, X) =$$

$$\begin{aligned} & \cup \{ \{a\}.FOLLOW_{k*}(stack + [q], a) \mid a \in DR(ts, X), q = \text{GOTO}_0(ts, X) \} \\ & \cup \{ FOLLOW_{k*}(stack + [q], Y) \mid (ts, X) \text{ reads } (q, Y) \} \\ & \cup \{ FOLLOW_{k*}(stack(1..(|stack| - |\gamma|)), C) \mid C \rightarrow \gamma \bullet X \in ts, |\gamma| + 1 < |stack| \} \\ & \cup \{ FOLLOW_{k*} * ([q], C) \mid C \rightarrow \gamma_1 \gamma_2 \bullet X \in ts, |\gamma_2| + 1 = |stack|, q \in PRED(stack(1), \gamma_1) \} \end{aligned}$$

O algoritmo para cômputo de  $FOLLOW_{k*}$ , apresentado na Figura 3.12, assim como feito em  $READ_{k*}$ , utiliza uma função para cálculo de um único símbolo em um *string*



```

READ2*(stack, X, k)
1  if k = 0 ∨ X = $
2    then return {(nil, λ)}
3  rd ← ∅
4  for (stk, a, item) ∈ READ-STEP-2(stack, X)
5  do if item ≠ nil
6    then rd ← rd ∪ {(item, λ)}
7    else rd ← rd ∪ {(item', ax) | (item', x) ∈ READ2*(stk, a, k - 1)}
8  return rd

READ-STEP2(stack, X)
1  configs ← ∅
2  ts ← TOP(stack)
3  q ← GOTO0(ts, X)
4  for Y ∈ V | GOTO0(q, Y) ≠ Ω
5  do if Y  $\xRightarrow{*}$  λ
6    then configs ← configs ∪ READ-STEP2(stack + [q], Y)
7    else if Y ∈ Σ
8      then configs ← configs ∪ {(stack + [q], Y, nil)}
9  for C → γ • X ∈ ts | C ≠ S
10 do if |γ| + 1 < |stack|
11   then configs ← configs ∪ READ-STEP2(stack(1..(|stack| - |γ|)), C)
12   else ASSERT(γ = αAβ), onde |Aβ| = n
13       configs ← configs ∪ (nil, nil, C → α • AβX)
14 return configs

```

Figura 3.11: Novo algoritmo para cômputo de  $READ_{k*}$ .

de *lookahead*. Este símbolo é retornado pela chamada a *FOLLOW-STEP*.

Anterior a toda chamada a *FOLLOW-STEP*, o conjunto global *visited* é iniciado como vazio. Como *FOLLOW-STEP* pode chamar-se recursivamente passando uma pilha constituída de um único estado (linha 17), o conjunto *visited* armazena pares da forma  $(p, X)$ , onde  $p$  é um estado em  $M_0$  e  $X \in V$ , de modo a evitar que mais de uma chamada a *FOLLOW-STEP*( $[p], X$ ) ocorra.

Os algoritmos *READ-STEP* e *FOLLOW-STEP* possuem muitas semelhanças. Existe, no entanto, uma diferença crucial entre eles: *FOLLOW-STEP* permite a obtenção de símbolos de *lookheads* fora do contexto da configuração recebida como parâmetro, enquanto que em *READ-STEP* isto não é permitido. Em [Charles, 1991], argumenta-se, no entanto, que se existirem um ou mais ciclos em  $LRA_0$  formados por símbolos não terminais anuláveis ou a gramática utilizada contiver regras que envolvam derivações da forma  $A \xRightarrow[rm]{*} A$ , tanto *READ-STEP* quanto *FOLLOW-STEP* podem não terminar. Como qualquer uma dessas situações resulta em uma gramática não  $LALR(k)$

[Charles, 1991], a verificação da não ocorrência delas permite a geração de analisadores sintáticos LALR( $k$ ).

Pela obtenção dos *lookaheads* de tamanho  $k*$  retornados pela função  $FOLLOW_*$ , a aplicação da equação referente a  $LA_k$  é direta. Isto, no entanto, não sugere que o problema da geração de analisadores sintáticos LALR( $k$ ) esteja resolvido.

O autômato  $LALRA_k$  é armazenado no programa do analisador sintático LALR( $k$ ) correspondente como duas tabelas, representadas na forma de matrizes: *Action* e *Goto*.

As linhas da tabela *Action* são indexadas pelos estados do autômato  $LALRA_k$  e as colunas por elementos em  $\Sigma^k$ , o que totaliza  $|M_0| \times |\Sigma|^k$  entradas.

A Tabela 3.1 mostra o número de entradas da tabela *Action* conforme o valor de  $k$  aumenta. Os resultados foram calculados a partir das análise do arquivo de *log*

$FOLLOW_*(stack, X, k)$

```

1  if  $k = 0 \vee X = \$$ 
2    then return  $\{\lambda\}$ 
3   $flw \leftarrow \emptyset$ 
4  for  $(stk, a) \in FOLLOW\text{-STEP}(stack, X)$ 
5  do  $flw \leftarrow flw \cup \{ax \mid x \in FOLLOW_*(stk, a, k - 1)\}$ 
6  return  $flw$ 

```

$FOLLOW\text{-STEP}(stack, X)$

```

1   $ts \leftarrow TOP(stack)$ 
2  if  $|stack| = 1$ 
3    then if  $(ts, X) \in visited$ 
4          then return  $\emptyset$ 
5           $visited \leftarrow visited \cup \{(ts, X)\}$ 
6   $configs \leftarrow \emptyset$ 
7   $q \leftarrow GOTO_0(ts, X)$ 
8  for  $Y \in V \mid GOTO_0(q, Y) \neq \Omega$ 
9  do if  $Y \xrightarrow{*} \lambda$ 
10     then  $configs \leftarrow configs \cup FOLLOW\text{-STEP}(stack + [q], Y)$ 
11     else if  $Y \in \Sigma$ 
12           then  $configs \leftarrow configs \cup \{(stack + [q], a)\}$ 
13  for  $C \rightarrow \gamma \bullet X \in ts \mid C \neq S$ 
14  do if  $|\gamma| + 1 < |stack|$ 
15     then  $configs \leftarrow configs \cup FOLLOW\text{-STEP}(stack(1..(|stack| - |\gamma|)), C)$ 
16     else ASSERT( $\gamma = \gamma_1\gamma_2$ ), onde  $|\gamma_2| + 1 = |stack|$ 
17           for  $q \in PRED(stack(1), \gamma_1)$ 
18           do  $configs \leftarrow configs \cup FOLLOW\text{-STEP}([q], C)$ 
19  return  $configs$ 

```

Figura 3.12: Algoritmo para cômputo de  $FOLLOW_{k*}$ .

criado pelo gerador CUP [CUP, 2007]. As gramáticas utilizadas foram obtidas em <http://www.devincook.com/goldparser/grammars/index.htm> e convertidas para a linguagem de especificação desse gerador. Pelo experimento realizado, percebe-se que

Gramática	$ M_0 $	$ \Sigma $	$k = 1$	$k = 2$	$k = 3$
C	352	85	29.920	2.543.200	216.172.000
C#	807	141	113.787	16.043.967	2.262.199.347
HTML	348	129	44.892	5.791.068	747.047.772
Java	632	107	67.624	7.235.768	774.227.176
Visual Basic .NET	636	144	91.584	13.188.096	1.899.085.824

Tabela 3.1: Tamanho da tabela *Action* conforme o valor de  $k$  aumenta.

a utilização da representação da tabela *Action*, indexada por estados e *strings* de *lookahead*, não é adequada. De fato, ao analisar uma tabela *Action*, nota-se que muitas das entradas são desnecessárias, pois representam seqüências inválidas de *tokens*. Considere, por exemplo, uma gramática LALR(2) de definições BNF e sua tabela *Action* correspondente, apresentadas respectivamente nas Figuras 3.13 e 3.14. O analisador sintático dessa tabela sempre consulta dois *lookaheads* para realizar qualquer decisão. Com exceção das entradas no estado 6, iniciadas pelo *token* **s**, a utilização de dois *lookaheads* nas demais entradas da tabela é totalmente desnecessária, uma vez que a

$$\begin{array}{lcl}
 bnf & \rightarrow & rlist \\
 rlist & \rightarrow & rlist \text{ rule} \\
 & | & \lambda \\
 rule & \rightarrow & \mathbf{s} \text{ “}\rightarrow\text{” } slist \\
 slist & \rightarrow & slist \mathbf{s} \\
 & | & \lambda
 \end{array}$$

Figura 3.13: Gramática LALR(2) para a notação BNF.

<i>Action</i>										
	“ $\rightarrow$ ” “ $\rightarrow$ ”	“ $\rightarrow$ ” <b>s</b>	“ $\rightarrow$ ” “\$”	<b>s</b> “ $\rightarrow$ ”	<b>ss</b>	<b>s</b> “\$”	“\$” “ $\rightarrow$ ”	“\$” <b>s</b>	“\$” “\$”	
0				R3						R3
1				S3						R1
2										ACC
3		S5	S5							
4				R2						R2
5				R6	R6	R6				R6
6				R4	S7	S7				R4
7				R5	R5	R5				R5

Figura 3.14: Tabela *Action* da gramática 3.13.

consulta o primeiro *token* de cada *string* de *lookahead* define univocamente a ação a ser realizada. No estado 6, se o sucessor de **s** for um outro **s** ou **\$**, o analisador sintático empilha o estado estado 7; do contrário, se “ $\rightarrow$ ” seguir “**s**” ou o *lookahead* corrente é **\$**, o analisador sintático reduz. Para os demais casos, o analisador sinaliza erro. Se fosse utilizado somente um símbolo de *lookahead*, ter-se-ia um conflito *shift/reduce*, pois  $Action[6, s]$  seria igual a  $\{R4, S7\}$ .

Charles [Charles, 1991] propõe um método de cômputo do conjunto mínimo de *lookaheads* de tamanho 1 a  $k$ , de forma incremental: inicialmente, calcula-se o autômato  $LALRA_1$  a partir de  $LRA_0$ . Se forem encontrados conflitos em  $LALRA_1$ , cada símbolo de conflito é estendido de forma a compôr um conjunto de *lookaheads* de tamanho 2. Se o conflito persistir para mais de um desses *strings*, cada um é novamente estendido com um novo símbolo de *lookahead*, resultando em *strings* de tamanho 3, e assim por diante até que um limite pré-estabelecido ( $k_{max}$ ) seja alcançado ou os conflitos tenham sido removidos. No fim desse processo, cada item final em um estado inconsistente está associado a um ou mais *lookaheads*  $w$ , onde  $1 \leq |w| \leq k_{max}$ . Desta forma, os *strings* de *lookaheads* possuem tamanho variável. O analisador sintático gerado segundo esta abordagem se comporta com um analisador LALR(1) quando somente um *lookahead* é suficiente para a escolha de qual decisão tomar. Quando isto não for possível, o analisador consulta até  $k_{max}$  *lookaheads* de modo a decidir qual ação realizar. Para representar as ações do analisador, o autor utiliza um esquema análogo ao da tabela *Action*, exceto pela existência de um novo tipo de valor armazenado: ações de *lookahead shift*.

Neste esquema, uma ação de *lookahead shift* é necessária a um analisador LALR( $k$ ) quando em um estado  $q$  e *token* corrente igual a  $a$ , a linha do estado  $q$  em *Action* contém uma ou mais entradas diferentes para *strings* de *lookaheads* iniciados em  $a$ . Considere, por exemplo, um alfabeto  $\Sigma' = \{a, b, c, d\}$  subconjunto do alfabeto  $\Sigma$  de uma gramática  $G$ . Seja  $Action[q, a] = \{S2, R5, R6\}$  e  $L = \{abc, adb, adc, acb\}$  os strings de tamanho 3 formados a partir de  $q$ . A Figura 3.4.2 mostra um autômato finito determinístico, denominado AFD de *lookahead* (AFDL), para consultar os possíveis símbolos que podem seguir  $a$ , dado o estado  $q$ . Neste caso, a entrada  $Action[q, a]$  aponta para  $q_1$  e  $q$  é dito ser o estado inicial do AFDL. Os demais estados no autômato são denominados *estados de lookahead*. Um *string* em  $L$  é reconhecido seguindo-se o caminho do estado inicial a uma das ações sintáticas indicadas.

As entradas nas tabelas *Action* e nas tabelas dos AFDLs criados armazenam ações de empilhamento, redução, erro ou ações de transição para um estado de um AFDL – ação de *lookahead shift*. A tabela de um AFDL é uma matriz cujas linhas são estados de *lookahead* e as colunas são símbolos em  $\Sigma$ . Por questões de uniformidade da análise sintática, as tabelas dos AFDLs são todas concatenadas no fim de *Action*, onde cada

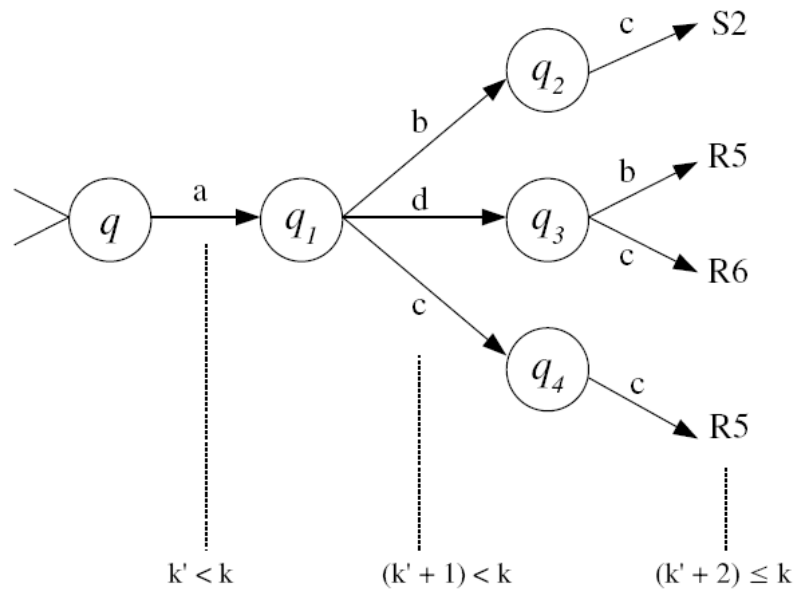


Figura 3.15: AFD para determinação das ações sintáticas a partir do percorrimento dos possíveis *lookaheads*.

linha concatenada é uma unidade superior em relação à última linha da tabela. As linhas concatenadas formam a tabela de *lookaheads*. Como exemplo, a Figura 3.16 mostra a tabela *Action* referente ao analisador LALR, com  $k_{max} = 2$ , da gramática da Figura 3.13. Note que a tabela apresentada é equivalente à da Figura 3.14, mas seu tamanho é aproximadamente 63% menor.

Um analisador sintático nessa nova representação, quando em execução, consulta a tabela *Action* normalmente. Se, dado um estado  $q$  e símbolo corrente  $a$ , o analisador encontra uma ação de *lookahead shift*  $q'$ , a consulta é transferida à tabela do AFDL apropriado. Em seguida, o analisador tenta obter a partir da leitura dos símbolos de entrada posteriores a  $a$ , um caminho no AFDL que leve a uma única ação sintática. Ao consultar esses símbolos de entrada, o analisador realiza ações de *lookahead shift*, que diferem das ações de empilhamento habituais no sentido de que os *tokens* consultados não são consumidos. Tão logo o analisador encontre uma combinação ilegal de *lookaheads* na tabela do AFDL, uma ação de erro é sinalizada.

Na seção seguinte são apresentados os algoritmos definidos em [Charles, 1991] para cômputo dos *strings* de *lookahead* de tamanho variável e como os mesmos são utilizados na construção das tabelas *Action* e de *lookaheads*.

### 3.4.2.1 Cômputo de *lookaheads* de tamanho variável

A geração de *lookaheads* de tamanho variável é feita para todo estado inconsistente no autômato  $LALRA_1$ . Um estado  $q$  é inconsistente se a interseção do conjunto  $LA_1$

<i>Action</i>		“→”	<b>s</b>	“\$”
	0	R3		R3
	1	S3		R1
	2			ACC
	3		S5	
	4	R2		R2
	5	R6		R6
	6	L8		R4
	7	R5		R5
<i>Lookaheads</i>	8	R4	S8	S8

Figura 3.16: Representação da tabela *Action* do analisador LALR(2) da gramática da Figura 3.13.

com o conjunto de terminais empilhados em  $q$  for diferente de vazia. Na tentativa de remover o conflito, todo símbolo  $a$  pertencente a essa interseção é estendido com um novo símbolo de *lookahead*, formando um *string* de tamanho 2. Se símbolos de conflitos forem encontrados no conjunto de símbolos concatenados a  $a$ , então o processo é repetido para formar *strings* de tamanho 3, e assim por diante até que um valor  $k_{max}$  pré-definido seja atingido.

De forma a estender um símbolo  $a$  de conflito no estado  $q$  em um *lookahead* de tamanho maior que um, as ações sintáticas em  $q$  associadas a  $a$  são devidamente identificadas. Isto inclui uma ou mais ações de redução e possivelmente uma ação de empilhamento. Em seguida, determina-se as fontes nas quais o símbolo  $a$  é lido em cada ação. Se a ação é de empilhamento, a fonte é um caminho constituído unicamente do estado  $q$ . Em uma ação de redução, as fontes são os caminhos do autômato que fizeram com que  $a$  aparecesse como símbolo de *lookahead* em  $LA_1(q, A \rightarrow \omega)$ . Neste caso, as fontes formam o conjunto de pilhas definido por:

$$stacks = \{stack \mid (stack, a) \in FOLLOW-STEP([p], A) \wedge p \in PRED(q, \omega)\}$$

A partir desse conjunto,  $a$  é estendido em *strings* de tamanho 2 por:

$$\{ab \mid (stack, b) \in FOLLOW-STEP(stk, a) \wedge stk \in stacks\}$$

Pela equação acima, durante a obtenção de  $b$ , *FOLLOW-STEP* já calcula o novo conjunto de pilhas a partir das quais pode-se facilmente obter os símbolos  $c$  de forma a produzir *lookaheads* de tamanho 3. Por razões a serem explicadas mais adiante, do ponto de vista de desempenho isto não é desejável. A função *FOLLOW-STEP* é, portanto, dividida em duas funções: *FOLLOW-SOURCES* e *NEXT-LA*. A função *FOLLOW-*

```

FOLLOW-SOURCES(stack, X, a)
1  ts ← TOP(stack)
2  if |stack| = 1
3      then if (ts, X) ∈ visited
4          then return ∅
5  stacks ← ∅
6  q ← GOTO0(ts, X)
7  for Y ∈ V | GOTO0(q, Y) ≠ Ω
8  do if Y = a
9      then stacks ← stacks ∪ {stack + [q]}
10     else if Y  $\xrightarrow{*}$  λ
11         then stacks ← stacks ∪ FOLLOW-SOURCES(stack + [q], Y, a)
12 for C → γ • X ∈ ts | C ≠ S
13 do if |γ| + 1 < |stack|
14     then stacks ← stacks ∪ FOLLOW-SOURCES(stack(1..(|stack| - |γ|)), C, a)
15     else ASSERT(γ = γ1γ2), onde |γ2| + 1 = |stack|
16         for q ∈ PRED(stack(1), |γ1|)
17         do stacks ← stacks ∪ FOLLOW-SOURCES([q], C, a)
18 return stacks

```

Figura 3.17: Função *FOLLOW-SOURCES*.

*SOURCES*, apresentada na Figura 3.17, recebe uma configuração  $cfg = (stack, X)$  e um símbolo  $a$  e retorna as pilhas obtidas pela simulação do autômato  $LRA_0$  a partir de  $cfg$ . A função *NEXT-LA* retorna o conjunto de terminais que podem ser lidos a partir de uma configuração inicial recebida como parâmetro. Ao invés de simular os passos do autômato  $LRA_0$ , *NEXT-LA* utiliza as funções  $READ_1$  e  $FOLLOW_1$ , prontamente disponíveis para cômputo de *lookaheads* de tamanho 1. A Figura 3.18 exhibe o algoritmo que define esta função.

Definidos esses dois blocos de construção, o cômputo do conjunto mínimo de *lookaheads* de tamanho variável é calculado pelo procedimento *LOOKAHEAD*, mostrado na Figura 3.19.

A função *LOOKAHEAD* inicia pela identificação das entradas na tabela *Action* que possuem conflito, isto é, as entradas com um número de ações maior que um. Para cada terminal  $a$  e ação  $act$  em  $Action[q, a]$ , tal que  $|Action[q, a]| > 1$ , o procedimento armazena em  $sources[act]$  as pilhas que resultaram no aparecimento de  $a$  como símbolo de *lookahead*. Determinadas as pilhas de cada ação, o procedimento *RESOLVE-CONFLICTS* é chamado de forma a estender  $a$  em um conjunto de *strings* de *lookahead* que não ocasionem conflitos.

O procedimento recursivo *RESOLVE-CONFLICTS* é invocado com quatro argumentos: o estado inconsistente  $q$ , o símbolo de conflito  $t$ , o dicionário *sources*, indexado

```

NEXT-LA( $stack, t$ )
1   $ts \leftarrow TOP(STACK)$ 
2   $q \leftarrow GOTO_0(ts, X)$ 
3   $la \leftarrow READ_1(ts, X)$ 
4  for  $C \rightarrow \gamma \bullet X\delta \in ts \mid \delta \xrightarrow{*} \lambda \wedge C \neq S$ 
5  do if  $|\gamma| + 1 < |stack|$ 
6      then  $la \leftarrow la \cup NEXT-LA(stack(1..(|stack| - |\gamma|)), C)$ 
7      else  $ASSERT(\gamma = \gamma_1\gamma_2)$ , onde  $\gamma_2 = |stack|$ 
8          for  $q \in PRED(stack(1), \gamma_1)$ 
9          do  $la \leftarrow la \cup FOLLOW_1(q, C)$ 
10 return  $la$ 

```

Figura 3.18: Função *NEXT-LA*.

```

LOOKAHEAD( $q$ )
1  for  $a \in \Sigma \mid |Action[q, a]| > 1$ 
2  do  $sources \leftarrow \emptyset$ 
3      for  $act \in Action[q, a]$ 
4      do if  $act$  é uma ação de empilhamento
5          then  $sources[act] \leftarrow \{[q]\}$ 
6          else  $ASSERT(act = \text{redução por } A \rightarrow \omega)$ 
7               $sources[act] \leftarrow \emptyset$ 
8              for  $p \in PRED(q, \omega)$ 
9              do  $visited \leftarrow \emptyset$ 
10                  $sources[act] \leftarrow sources[act] \cup FOLLOW-SOURCES([p], A, a)$ 
11  RESOLVE-CONFLICTS( $q, a, sources, 2$ )

```

Figura 3.19: Procedimento *LOOKAHEAD*.

por ações, e o inteiro  $n$  para controlar a posição dos símbolos que irão estender cada *string* de *lookahead* obtido. O procedimento inicia pela verificação se  $n > k_{max}$  ou o símbolo de conflito é  $\$$ . Em qualquer um dos casos, não é possível obter nenhum símbolo de extensão e portanto, o procedimento retorna. Do contrário, aloca-se uma nova linha  $p$  na tabela de *lookaheads*<sup>1</sup>, atribui a  $Action[p, a]$ , para todo  $a \in \Sigma$ , o conjunto vazio, e a ação de  $\{la-shift\ p\}$  é colocada em  $Action[q, t]$ . Em seguida, para toda ação  $act$  que indexa o dicionário *sources*, utiliza-se cada pilha  $stk$  em  $sources[act]$  na chamada a *NEXT-LA* de forma a determinar os *tokens* que devem ser lidos a partir da configuração  $(stk, t)$ . Para um *token*  $a$  retornado, o procedimento atribui  $act$  a  $Action[p, a]$ . A função *NEXT-LA* retorna os novos símbolos de *lookahead*, mas não a

<sup>1</sup>A alocação do primeiro estado após a última linha da tabela *Action* marca o início da tabela de *lookaheads*.



```

RESOLVE-CONFLICTS( $q, t, sources, n$ )
1  if  $n > k_{max} \vee t = \$$ 
2    then return
3  alocar uma nova linha  $p$  em Action
4  for  $a \in \Sigma$ 
5    do  $Action[p, a] \leftarrow \emptyset$ 
6   $Action[q, t] \leftarrow \{la-shift\ p\}$ 
7  for  $act \in INDEX(sources)$ 
8    do for  $stk \in sources[act]$ 
9      do for  $a \in NEXT-LA(stk, t)$ 
10         do  $Action[p, a] \leftarrow Action[p, a] \cup \{act\}$ 
11 for  $a \in \Sigma \mid Action[q, a] > 1$ 
12 do  $newSources \leftarrow \emptyset$ 
13   for  $act \in Action[q, a]$ 
14     do  $newSources[act] \leftarrow \emptyset$ 
15       for  $stk \in sources[act]$ 
16         do  $visited \leftarrow \emptyset$ 
17            $newSources \leftarrow newSources \cup FOLLOW-SOURCES(stk, k, a)$ 
18   RESOLVE-CONFLICTS( $p, a, newSources, n+1$ )

```

Figura 3.20: Procedimento *RESOLVE-CONFLICTS*.

pilha que resulta no seu aparecimento. Esta responsabilidade é da função *FOLLOW-SOURCES*. Isto é feito na tentativa de se determinar rapidamente se conflito é ou não removido ao concatenar os novos símbolos  $a$  a  $t$ . No fim disto, as entradas da linha  $p$  são verificadas. Para toda entrada  $Action[p, a]$  com cardinalidade superior a 1, o procedimento calcula o conjunto  $newSources$ , pela chamada a *FOLLOW-SOURCES* e faz uma chamada recursiva a *RESOLVE-CONFLICTS* de modo a repetir o processo para o símbolo de conflito  $a$ .

O conjunto de funções apresentadas para a computação do conjunto mínimo de *strings* de *lookahead* de tamanho variável possuem a garantia de término se e somente se a relação *reads*, *includes* e a gramática de entrada forem livres de ciclos e a gramática não contém não terminais  $A$ , tal que  $A \xRightarrow{*} A$ .

## 3.5 Conclusão

Este capítulo apresentou as definições formais da análise sintática LALR e os algoritmos utilizados no cômputo de *lookaheads*.

Para  $k = 1$ , explicou-se o algoritmo proposto por DeRemer e Pennello, que realiza o cálculo a partir de um único percorrimto do grafo da relação *reads* e *includes*. Conforme será discutido no 5, essas relações, além de base para o algoritmo de cômputo de

*lookaheads* de tamanho variável apresentado em [Charles, 1991], permitem a obtenção de árvores de derivação para apresentação de conflitos.

Na estratégia de  $k$  variável, o tamanho das tabelas *Action* é reduzido se comparado à abordagem tradicional de utilização de *lookaheads* de tamanho  $k$  para indexação das colunas. Além disto, a tabela produzida neste método tem *layout* idêntico a uma tabela LALR(1); a única diferença está na presença de ações de *lookahead shift*. Conforme será discutido no Capítulo 4, essa semelhança de *layout* permitirá que essas tabelas sejam compactadas por métodos de compactação originalmente projetados para tabelas LALR(1).

# Capítulo 4

## Compactação de Tabelas

Um analisador sintático LALR( $k$ ) é representado pelas tabelas *Action* e *Goto*, indexadas respectivamente pelos pares  $(s, t^k)$  e  $(s, A)$ , onde  $s \in M_0$ ,  $t^k \in \Sigma_k^*$  e  $A \in N$ . O conjunto  $\Sigma_k^*$  é dado por  $\{w \mid w \in \Sigma^* \wedge 0 \leq |w| \leq k\}$ . Essas tabelas, quando construídas para linguagens como C#, Java, VB .NET, possuem um número excessivo de entradas, em que muitas são não significativas. Uma entrada não significativa é uma entrada em branco. Na tabela *Action*, essas entradas denotam erro, ao passo que em *Goto* não possuem significado, pois nunca são consultadas. Do experimento realizado na Seção 3.4.2 (vide Tabela 3.1), mostrou-se a inviabilidade da representação da tabela *Action* como uma matriz  $|M_0| \times |\Sigma|^k$ . Levando em consideração a representação da tabela *Goto*, o total de entradas utilizadas na representação do analisador sintático é ainda maior. Ressalta-se, no entanto, que o tamanho da tabela *Goto* é independente do número de *lookaheads* utilizados. O número final de entradas necessárias à representação de analisadores sintáticos LALR( $k$ ) é dado por:

$$n_e = (|M_0| \times |\Sigma|^k) + (|M_0| \times |N|)$$

Métodos de compactação de tabelas são utilizados de forma a reduzir a demanda de memória necessária ao armazenamento das tabelas sintáticas. Ainda que a quantidade de memória primária tenha aumentado significativamente nos últimos anos, a compactação de tabelas dos analisadores sintáticos gerados é extremamente desejável em dispositivos com quantidade reduzida de memória, cujo acréscimo acarreta em perda de autonomia. Como exemplo, analisadores sintáticos projetados para execução em uma rede de sensores devem fazer uso da menor quantidade de memória possível; analisadores sintáticos de aplicações como editores de texto e navegadores Web escritas para dispositivos móveis também são limitados por sua quantidade de memória, ainda que em menor grau. Para o caso particular dos computadores de mesa, onde a disponibilidade de memória é alta, a utilização de métodos de compactação permite a obtenção de

analísadores sintáticos LALR( $k$ ) com valores de  $k$  cada vez maiores.

Este capítulo apresenta 8 métodos de compactação de tabelas LALR(1), categorizados segundo o resultado produzido: *compactação por vetor* e *compactação por matriz*. A compactação por vetor lineariza a tabela de entrada, enquanto a compactação por matriz preserva a estrutura tabular. Cada método é avaliado empiricamente em função das tabelas LALR(1) das gramáticas das linguagens de programação C, C#, HTML, Java e VB .NET. As características dessas gramáticas e de suas tabelas *Action* e *Goto* são exibidas respectivamente nas Tabelas 4.1, 4.2 e 4.3 <sup>1</sup>. No fim da apresentação dos métodos em cada categoria, faz-se uma análise conjunto de cada estratégia de compactação e melhorias são propostas, contrastando-se os novos resultados com os anteriormente obtidos. Por fim, os métodos apresentados são generalizados de forma a dar suporte às tabelas de analisadores LALR( $k$ ), com  $k$  variável. A avaliação realizada neste capítulo foi feita a partir da implementação de cada um dos métodos apresentados.

## 4.1 Codificação

Toda tabela sintática possui uma forma de codificação dependente do gerador de analisador sintático utilizado. A codificação utilizada representa cada entrada da tabela como um número inteiro com sinal. Na tabela *Action*, isto traduz-se nas seguintes possibilidades:

- números negativos no intervalo MIN\_INT ... -1 denotam reduções;
- os números no intervalo 0...MAX\_INT - 1 identificam ações de empilhamento;
- o maior inteiro positivo (MAX\_INT) identifica a ação de erro.

Nessa codificação, a ação de aceitação não é explicitamente representada; o empilhamento do estado cujo símbolo de acesso é \$ marca a aceitação do *string* de entrada.

As constantes MIN\_INT e MAX\_INT são dependentes da quantidade de bytes utilizadas na representação de cada entrada da tabela. Para as tabelas de teste, as entradas em *Action* e *Goto* necessitam de até dois bytes com sinal; portanto, MIN\_INT = -32.768 e MAX\_INT = 32.767.

No caso de utilização de vetores auxiliares por algum método de compactação, verificou-se que 2 bytes também são suficientes à representação de cada entrada. Exceto quando mencionado o contrário, essas entradas não possuem sinal.

---

<sup>1</sup>As gramáticas utilizadas são as mencionadas no Capítulo 3, obtidas do repositório de gramáticas da ferramenta Gold Parser e convertidas para a linguagem de especificação do CUP.

Gramática	Características		
	$ \Sigma $	$ N $	$ P $
C	85	50	174
C#	141	130	407
HTML	129	25	104
Java	107	137	360
VB .NET	144	108	325

Tabela 4.1: Gramáticas LALR(1) utilizadas na avaliação da taxa de compactação dos métodos apresentados.

Gramática	<i>Action</i>					
	$ M_0 $	$ \Sigma $	Entradas	Entradas (Sig.)	% Sig.	Bytes
C	352	85	29.920	5.446	18.2	59.840
C#	807	141	113.787	13.743	12.1	227.574
HTML	348	129	44.892	8.236	18.4	89.784
Java	632	107	67.624	11.604	17.2	135.248
VB .NET	636	144	91.584	8.568	9.4	183.168

Tabela 4.2: Tabelas *Action* das gramáticas de teste. Da esquerda para a direita tem-se o número de estados, o tamanho do alfabeto, o número de entradas, quantas são significativas (não erro), o porcentual que representam e o número total de bytes da tabela não compactada.

Na tabela *Goto* somente os números no intervalo de 0...MAX.INT são de fato utilizados. A constante MAX.INT neste caso, denota uma entrada em branco.

Assume-se ainda que os terminais da gramática são identificados por constantes numéricas contidas em um intervalo seqüencial iniciado em zero. Isto também é suposto para os não terminais e produções.

Gramática	<i>Goto</i>					
	$ M_0 $	$ N $	Entradas	Entradas (Sig.)	% Sig.	Bytes
C	352	51	17.952	1.086	6.1	35.904
C#	807	131	105.717	2.140	2	211.434
HTML	348	26	9.048	309	3.4	18.096
Java	632	139	87.848	4.279	4.9	175.696
VB .NET	636	109	69.324	1.306	1.9	138.648

Tabela 4.3: Tabelas *Goto* das gramáticas de teste. Da esquerda para a direita tem-se o número de estados, o tamanho do alfabeto, o número de entradas, quantas são significativas (diferentes de branco), o porcentual que representam e o número total de bytes da tabela não compactada.

## 4.2 Compactação por Vetor

Os métodos de compactação por vetor linearizam a tabela *Action* e *Goto*, colocando as entradas em um vetor denominado  $value_a$  e  $value_g$  respectivamente. Quando for necessário se referir tanto a  $value_a$  e  $value_g$ , o nome genérico *value* será utilizado. São apresentados os seguintes métodos de compactação por vetor: *Aho Compression Scheme* [Aho e Ullman, 1972], *Bigonha Compression Scheme* [Bigonha e Bigonha, 1983], *Row Displacement Scheme* [Ziegler, 1977], *Significant Distance Scheme* [Beach, 1974], *Row Column Scheme* [Tewarson, 1968] e *Zero Supression Scheme* [Dencker et al., 1984].

### 4.2.1 Método de Compactação Proposto por Aho (ACS)

Este método de compactação [Aho e Ullman, 1972] (*Aho Compression Scheme*) substitui o acesso direto por uma busca linear a um intervalo de *value*.

A compactação da tabela *Action* tem como estratégia a fusão de linhas idênticas, seguida da utilização de *ações padrão*. Uma ação padrão em uma linha é a ação de erro, se a linha não contém reduções, ou a ação de redução que ocorre com maior frequência, caso contrário. Considere, por exemplo, uma possível linha de uma tabela *Action*:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
4	R0		R1	S5		R0		R2	R0		R0			

Ao utilizar ação padrão, a mesma linha é representada por:

$$\text{estado 4: } [(3, S5), (2, R1), (7, R2), (any, R0)]$$

Nesta forma, as entradas de um estado são dadas por uma seqüência de pares  $(t, act)$ , onde  $t$  é um terminal e  $act$  é uma ação sintática. O par  $(any, act)$ , colocado como última entrada do estado em questão, indica que se a entrada referente ao *token* corrente não for encontrada nas posições anteriores a *any*, então *act* deve ser aplicada. Nesta representação, pode ocorrer de ações de erro serem substituídas por ações de redução. No entanto, a corretude do analisador sintático não é prejudicada, embora um número maior de reduções pode ser realizado em relação à representação original; a detecção do erro ocorre antes que um novo símbolo da entrada seja empilhado.

A representação das linhas da tabela segundo este método usa um vetor auxiliar denominado *rowindex*. Esse vetor mapeia cada linha  $i$  em *Action* em uma posição em  $value_a$  correspondente ao início do intervalo onde  $i$  está armazenada. Um par  $(t, act)$  ocupa duas posições contíguas em  $value_a$ . Assim, dado um estado  $i$ , se a partir de  $rowindex[i]$  a entrada para o terminal  $a$  estiver em  $value_a[p]$ ,  $value_a[p + 1]$  contém a ação a ser tomada. A ação padrão é codificada após a última ação não padrão de

forma a demarcar o final da representação de  $i$  em  $value_a$ . Para representar o símbolo  $any$ , utiliza-se um símbolo não pertencente ao conjunto de terminais. A entrada após esse símbolo contém o valor da ação padrão. Neste método de compactação, duas linhas iguais, digamos  $i_1$  e  $i_2$ , são codificadas em um único intervalo em  $value_a$ . Neste caso, tem-se que  $rowindex[i_1] = rowindex[i_2]$ . Para simular o acesso a  $Action[i, j]$  percorre-se  $value_a$  a partir da posição armazenada em  $rowindex[i]$ . Se a entrada  $any$  for alcançada, então a ação padrão, que está armazenada na posição seguinte a  $any$ , é retornada. Caso contrário, uma entrada referente ao símbolo  $j$  foi encontrada. A entrada na posição seguinte a  $j$  contém a ação desejada.

A compactação da tabela *Goto* segue um esquema análogo ao mostrado na compactação da tabela *Action*. Nesta compactação, o vetor  $value_g$  armazena os valores de cada coluna como pares da forma  $(cs, ns)$ , onde  $cs$  é o estado corrente e  $ns$  é o próximo estado. Com isto, deve-se utilizar um vetor auxiliar, denominado *columnindex*, para armazenar a posição em que cada coluna em *Goto* é mapeada em  $value_g$ . As entradas padrão são os estados que ocorrem com maior frequência na coluna em questão; não existem entradas de erro na tabela *Goto*, isto é, as entradas em branco não possuem qualquer significado, pois nunca são consultadas,

O resultado da compactação das tabelas *Action* e *Goto* das gramáticas de teste é apresentado respectivamente nas Tabelas 4.4 e 4.5. O resultado final obtido por este método é exibido na Tabela 4.6.

Gramática	$ rowindex $	$ value_a $	Total (bytes)	Compactação (%)
C	352	1.616	3.936	93,4
C#	807	3.444	8.502	96,3
HTML	348	5.302	11.300	87,4
Java	632	4.636	10.536	92,2
VB .NET	636	2.520	6.312	96,6
<b>Média</b>				93,2

Tabela 4.4: Resultados da compactação da tabela *Action* pelo método ACS.

A utilização de entradas padrão neste método responde por grande parte da compactação obtida, pois elimina um alto percentual de entradas não significativas, que são as entradas mais frequentemente encontradas (vide Tabela 4.2 e 4.3). Em *Action*, entradas não significativas correspondem às entradas erro, enquanto em *Goto* representam a ausência de ação. Na compactação da tabela *Action*, o outro ganho advém da combinação de linhas iguais. Isto permite reduzir em média 21,5% do número de entradas, conforme mostrado na Tabela 4.7.

Gramática	$ columnindex $	$ value_g $	Total (bytes)	Compactação (%)
C	51	358	818	97,7
C#	131	1.002	2.266	98,9
HTML	26	310	672	96,3
Java	139	1.724	3.726	97,9
VB .NET	109	838	1.894	98,6
<b>Média</b>				97,9

Tabela 4.5: Resultados da compactação da tabela *Goto* pelo método ACS.

Gramática	Total (bytes)	Compactação (%)
C	4.754	95
C#	10.768	97,6
HTML	11.972	88,9
Java	14.262	95,4
VB .NET	8206	97,5
<b>Média</b>		94,9

Tabela 4.6: Resultado final da compactação das tabelas *Action* e *Goto* pelo método ACS.

Gramática	<i>Action</i>		
	Linhas	Linhas iguais	Linhas iguais (%)
C	352	78	22,2
C#	807	176	21,8
HTML	348	75	21,6
Java	632	95	15
VB .NET	636	171	26,9
<b>Média</b>			21,5

Tabela 4.7: Linhas repetidas nas tabelas *Action* das gramáticas de teste.

### 4.2.2 Método de Compactação Proposto por Bigonha (BCS)

Este método [Bigonha e Bigonha, 1983] codifica a tabela sintática em  $value^2$ , dividindo-o em intervalos, um para cada estado, onde as entradas correspondem a posições para outros estados no vetor. A primeira entrada de cada intervalo é sempre o símbolo de acesso para o estado em questão; as demais armazenam posições de estados sucessores.

Existem três estados especiais em  $value$ , responsáveis pela indicação de erro, redução e aceitação, ocupando respectivamente as posições  $E$ ,  $R$  e  $F$ . Um estado que contém somente ações de empilhamento tem com último valor em seu respectivo intervalo a

<sup>2</sup>Em [Bigonha e Bigonha, 1983], o vetor  $value$  é denominado LALR.



posição do estado  $E$ . No caso de um estado possuir pelo menos uma ação de redução, a ação de redução mais freqüente é codificada na última posição do intervalo correspondente. Essa codificação é feita armazenando-se  $p_{rn}$  e  $R$  nas duas últimas posições desse intervalo, onde  $p_{rn}$  é a posição do estado em *value* correspondente à ação de redução  $r_n$ .

Os estados de redução sempre ocupam posições posteriores a  $R$ . Uma redução é armazenada utilizando-se duas entradas contíguas: uma para o símbolo de acesso (símbolo de *lookahead*) e a outra para a produção a ser utilizada.

O mecanismo geral do algoritmo de análise sintática neste método de compactação baseia-se em transições. A cada transição para um estado  $k$ , quatro cenários são possíveis: (i) a posição de  $k$  em *value* é menor que  $E$ . A ação representada neste caso é a de empilhamento (*shift k*); (ii) a posição de  $k$  em *value* é maior que  $R$ . Nesta situação tem-se então uma redução de acordo com o par (*token, produção*). O *token* em questão deve ser igual ao *token* corrente; (iii) a posição de  $k$  é igual a  $F$  ou  $E$ , indicando respectivamente aceitação ou erro; (iv) a posição de  $k$  é igual a  $R$ . Neste caso, aplica-se a redução mais freqüente na linha do estado em questão. A posição que armazena o índice do estado dessa redução encontra-se na penúltima posição do intervalo do estado corrente.

A tabela *Goto* é por construção embutida em *value*. Isto é possível a partir do momento em que é permitido símbolos não-terminais como símbolos de acesso. Ressalta-se, no entanto, que os números utilizados na representação de terminais e não terminais deve ser disjunto.

A forma como a compactação é realizada neste método contrasta com a idéia proposta por Aho. Aho elimina listas iguais e codifica as entradas de *value<sub>a</sub>* como pares da forma (*símbolo terminal, ação sintática*). O método BCS substitui esses pares pelos símbolos de acesso e pelas posições dos estados em *value*, o que permite recuperar os dois componentes de um dado par. Esta representação elimina a necessidade de sempre armazenar o símbolo terminal associado a uma ação sintática. Outro aspecto importante é a não utilização de vetores auxiliares, o que também ocasiona maior ganho na compactação.

De acordo com os autores, as condições necessárias à eficiência deste método são:

1. a relação *número de transições/número de estados* deve ser razoavelmente maior que 1. Quanto maior esta relação, maior é a taxa de compactação;
2. a relação *número de linhas repetidas/total de linhas* deve ser maior que  $a/(c*m)$ , onde  $a$  é o espaço necessário ao armazenamento de uma referência a um estado (no caso da codificação adotada,  $a = 2$ ),  $m$  é o número médio de entradas por

linha e  $c$  é o espaço ocupado por um par (*coluna, ação sintática*) utilizado no método proposto por Aho;

3. entradas de erro devem ser a entrada mais freqüente em cada linha.

Por não fazer distinção entre as entradas da tabelas *Action* e *Goto* em *value*, a Tabela 4.8 mostra o resultado final obtido por este método de compactação.

O tamanho compactado da tabela final é dado por

$$\text{Tamanho compactado} = 2 \times (|\text{value}| + 3)$$

O número três adicionado a  $|\text{value}|$  é oriundo da necessidade de se ter três variáveis para armazenar as posições  $E$ ,  $R$  e  $F$ .

Gramática	$ \text{value} $	Total (bytes)	Compactação (%)
C	3.083	6.172	93,4
C#	6.827	13.660	96,9
HTML	3.815	7.636	92,9
Java	6.116	12.238	96,1
VB .NET	4.473	8.952	97,2
<b>Média</b>			95,3

Tabela 4.8: Resultado final da compactação das tabelas *Action* e *Goto* pelo método BCS.

### 4.2.3 Método de Compactação *Row Displacement* (RDS)

Este método de compactação (*Row Displacement Scheme*) [Ziegler, 1977] combina as linhas da tabela de entrada em *value*, inserindo uma linha por iteração. A cada passo, procura-se em *value* uma posição a partir da inicial de forma que as entradas significativas da linha corrente sobrescrevam somente entradas não significativas de linhas anteriormente inseridas. Neste processo, são utilizados dois vetores auxiliares: *rowindex* e *rowpointer*. Uma entrada  $\text{rowindex}[k]$  contém o número de uma linha  $i$  se  $\text{value}[k]$  contém um valor significativo de  $i$ . Um elemento em  $\text{rowpointer}[i]$  armazena a posição em *value* que corresponderia à posição inicial da primeira entrada da linha  $i$ , significativa ou não.

A recuperação de uma entrada  $\text{Action}[i, j]$  neste esquema de compactação é definida por:

$\text{ACTION}(i, j)$

```

1 if  $rowindex_a[rowpointer_a[i] + j] \neq i$ 
2   then return error
3 return  $value[rowpointer_a[i] + j]$ 

```

No caso da tabela *Goto*, como somente entradas significativas são consultadas, o acesso é simplificado pela não utilização do vetor *rowindex*:

GOTO(*i*, *j*)

```

1 return  $value_g[rowpointer_g[i] + j]$ 

```

O grau de compactação deste método depende da forma como as linhas são inseridas em *value*. O problema de ser obter uma seqüência de inserção que maximize a sobreposição de linhas é um problema NP-completo [Ziegler, 1977]. Ziegler propõe uma heurística para tratar esse problema, denominada *first fit decreasing*. O algoritmo de inserção das linhas segundo essa heurística funciona da seguinte forma: inicialmente, as linhas da tabela são ordenadas em ordem decrescente do número de entradas significativas que elas contêm. Em seguida, cada linha resultante da ordenação é inserida em *value*. Para inserir uma linha *i*, percorre-se *value* de forma a encontrar a primeira posição a partir da qual as entradas em *i* sobrescrevam somente entradas não significativas de linhas anteriormente inseridas.

As Tabelas 4.9 e 4.10 mostram o resultado da compactação das tabelas *Action* e *Goto* pela aplicação deste método segundo a heurística *first fit decreasing*. O resultado final é apresentado na Tabela 4.11.

Gramática	$ value_a $	$ rowindex_a $	$ rowpointer_a $	Total (bytes)	Compressão (%)
C	8.733	352	8.733	35.636	40,5
C#	28.085	807	28.085	113.954	49,9
HTML	10.919	348	10.919	44.372	50,6
Java	21.725	632	21.725	88.164	34,8
VB .NET	18.867	636	18.867	76.740	58,1
<b>Média</b>					46,8

Tabela 4.9: Resultados da compactação da tabela *Action* pelo método RDS.

Gramática	$ value_g $	$ rowpointer_g $	Total (bytes)	Compressão (%)
C	1256	352	3.216	91
C#	2489	807	6.592	96,9
HTML	328	348	1.352	92,5
Java	7399	632	16.062	90,9
VB .NET	1594	636	4.460	96,8
<b>Média</b>				93,6

Tabela 4.10: Resultados da compactação da tabela *Goto* pelo método RDS.

Gramática	Total (bytes)	Compactação (%)
C	38.852	59,4
C#	120.546	72,5
HTML	45.724	57,6
Java	104.226	66,5
VB .NET	81.200	74,8
<b>Média</b>		66,2

Tabela 4.11: Resultado final da compactação das tabelas *Action* e *Goto* pelo método RDS.

#### 4.2.4 Método de Compactação por Distância Significativa (SDS)

Este método de compactação (*Significant Distance Scheme*) [Beach, 1974] ignora as entradas não significativas anteriores à primeira e posteriores à última entrada significativa de uma dada linha da tabela sintática. As demais entradas não ignoradas são inseridas seqüencialmente em *value*, linha após linha. São utilizados três vetores auxiliares: *rowpointer*, *first* e *last*. Armazena-se em *first*[*i*] o índice da coluna da primeira e em *last*[*i*] o índice da coluna da última entrada significativa da *i*-ésima linha da tabela original. Uma entrada em *rowpointer*[*i*] contém a posição em *value* que o primeiro elemento de *i* ocuparia, sendo em alguns casos armazenados índices negativos.

A aplicação deste esquema de compactação à tabela *Action* resulta no seguinte algoritmo para obter o valor de uma dada entrada:

```

ACTION(i, j)
1  if  $j < first[i] \vee j > last[i]$ 
2     then return error
3  return  $value[rowpointer_a[i] + j]$ 

```

Para acesso à tabela *Goto*, o algoritmo é simplificado para:

GOTO( $i, j$ )

1 **return**  $value_g[rowpointer_g[i] + j]$

As Tabelas 4.12 e 4.13 mostram os resultados da compactação das tabelas *Action* e *Goto* por este método. As entradas em *rowpointer* são representadas por dois bytes com sinal. A Tabela 4.14 mostra o resultado final obtido.

Gramática	$ value_a $	$ first $	$ last $	$ rowpointer_a $	Total (bytes)	Compactação (%)
C	14.645	352	352	352	31.402	47,5
C#	51.239	807	807	807	107.320	52,8
HTML	15.698	348	348	348	33.484	62,7
Java	38.812	632	632	632	81.416	39,8
VB .NET	44.967	636	636	636	93.750	48,8
<b>Média</b>						50,3

Tabela 4.12: Resultados da compactação da tabela *Action* pelo método SDS.

Gramática	$ value _g$	$ rowpointer _g$	Total (bytes)	Compactação (%)
C	1.397	352	3.498	90,3
C#	7.476	807	16.566	92,2
HTML	980	348	2.656	85,3
Java	15.374	632	32.012	81,8
VB .NET	4.784	636	10.840	92,2
<b>Média</b>				88,3

Tabela 4.13: Resultados da compactação da tabela *Goto* pelo método SDS.

#### 4.2.5 Método de Compactação *Row Column* (RCS)

Este esquema de compactação (*Row Column Scheme*) [Tewarson, 1968] armazena os elementos da tabela original em *value*. Dois vetores auxiliares são utilizados: *columnindex* e *rowpointer*. Dada uma linha  $i$ ,  $rowpointer[i]$  contém a primeira posição livre em *value*, que é preenchido da seguinte forma: a partir de  $rowpointer[i]$ , as entradas significativas das colunas  $j$  da  $i$ -ésima linha da tabela original são inseridas uma a uma. O valor de  $j$  é inserido na posição em *columnindex* correspondente à posição em *value*.

Para obter uma entrada em  $Action(i, j)$ , faz-se:

Gramática	Total (bytes)	Compactação (%)
C	35.604	63,6
C#	125.500	71,8
HTML	36.836	66,5
Java	114.692	63,5
VB .NET	105.862	67,5
<b>Média</b>		66,6

Tabela 4.14: Resultado final da compactação das tabelas *Action* e *Goto* pelo método SDS.

**ACTION**( $i, j$ )

```

1  for  $k \leftarrow \text{rowpointer}_a[i]$  to  $k \leftarrow \text{rowpointer}_a[i + 1] - 1$ 
2    do if  $\text{columnindex}_a[k] = j$ 
3      then return  $\text{value}_a[k]$ 
4    return error

```

O algoritmo para recuperar um valor em  $Goto(i, j)$  é idêntico ao apresentado, exceto que os vetores  $\text{rowpointer}_g$ ,  $\text{columnindex}_g$  e  $\text{value}_g$  são manipulados.

As Tabelas 4.15 e 4.16 apresentam o resultado da compactação das tabelas *Action* e *Goto* das gramáticas de teste. O resultado final da compactação é apresentado na Tabela 4.6.

Gramática	$ \text{value} $	$ \text{rowpointer}_a $	$ \text{columnindex}_a $	Total (bytes)	Compactação (%)
C	5.446	353	5.446	22.490	62,42
C#	13.743	808	13.743	56.588	75,13
HTML	8.236	349	8.236	33.642	62,53
Java	11.604	633	11.604	47.682	64,74
VB .NET	8.568	637	8.568	35.546	80,59
<b>Média</b>					69,1

Tabela 4.15: Resultados da compactação da tabela *Action* pelo método RCS.

#### 4.2.6 Método de Compactação por Supressão de Zero (SZS)

Este método de compactação (*Zero Suppress Scheme*) [Dencker et al., 1984] lineariza a tabela de entrada pela inserção seqüencial de todos os seus elementos em  $\text{value}$ . Quando  $n$  entradas não significativas são encontradas, com  $n \geq 1$ ,  $n$  é armazenada na posição

Gramática	$ value $	$ rowpointer_a $	$ columnindex_a $	Total (bytes)	Compactação (%)
C	1.086	353	1.086	5.050	85,93
C#	2.140	808	2.140	10.176	95,19
HTML	309	349	309	1.934	89,31
Java	4.279	633	4279	18.382	89,54
VB .NET	1.306	637	1.306	6.498	95,31
<b>Média</b>					91,1

Tabela 4.16: Resultados da compactação da tabela *Goto* pelo método RCS.

Gramática	Total (bytes)	Compactação (%)
C	27.540	71,24
C#	66.764	84,79
HTML	35.576	67,02
Java	66.064	78,75
VB .NET	42.044	86,94
<b>Média</b>		77,8

Tabela 4.17: Resultado final da compactação das tabelas *Action* e *Goto* pelo método RCS.

corrente em  $value$ . De forma a diferenciar essa entrada de uma ação sintática ordinária, a mesma é precedida de um marcador, como por exemplo, um número reservado.

Para obter o valor de uma entrada em  $Action[i, j]$  percorre-se  $value_a$  a partir da primeira posição, mantendo-se um contador  $pos$  para armazenar o número de entradas da tabela original inspecionadas até o momento e um índice  $k$  para guardar a posição corrente em  $value_a$ . Quando uma entrada significativa é encontrada e  $pos$  é igual a  $(i \times |\Sigma|) + j$ , então  $value_a[k]$  contém a entrada desejada. Do contrário, se  $pos$  for maior que  $(i \times |\Sigma|) + j$ , tem-se uma entrada correspondente à ação de erro. Senão, a busca não terminou: se  $value_a$  contiver o valor de marcação, então incrementa-se  $k$  em duas unidades e faz-se  $pos$  igual a  $pos + value_a[k + 1]$ . Do contrário, tanto  $k$  quanto  $pos$  são incrementados em uma unidade.

O acesso a  $Goto(i, j)$  é análogo a da tabela *Action*, exceto que a entrada desejada é dada por  $(i \times |N|) + j$  e o vetor manipulado é  $value_g$ . Neste caso, a verificação se  $pos$  ultrapassa esse valor é desnecessária.

**Análise:** o resultado da compactação das tabelas *Action* e *Goto* por este método é mostrado nas Tabelas 4.18 e 4.19. O resultado final é exibido na Figura 4.20.

Gramática	$ value_a $	Total (bytes)	Compactação (%)
C	10.238	20.476	65,8
C#	30.181	60.362	73,5
HTML	13.786	27.572	69,3
Java	23.812	47.624	64,8
VB .NET	19858	39716	78,3
<b>Média</b>			<b>70,3</b>

Tabela 4.18: Resultados da compactação da tabela *Action* pelo método SZS.

Gramática	$ value_a $	Total (bytes)	Compactação (%)
C	1.514	3.028	91,6
C#	3.446	6.892	96,7
HTML	761	1.522	91,6
Java	6.393	12.786	92,7
VB .NET	2.648	5.296	96,2
<b>Média</b>			<b>93,8</b>

Tabela 4.19: Resultados da compactação da tabela *Goto* pelo método SZS.

Gramática	Total (bytes)	Compactação (%)
C	23.504	75,5
C#	67.254	84,7
HTML	29.094	73
Java	60.410	80,6
VB .NET	45.012	86
<b>Média</b>		<b>80</b>

Tabela 4.20: Resultado final da compactação das tabelas *Action* e *Goto* pelo método SZS.

### 4.2.7 Análise

Os métodos de compactação ACS e BCS detêm um alto índice de compactação, iguais a 94,9% e 95,3% respectivamente. Um fator importante na obtenção desses índices é a utilização de ações padrão. Tais ações permitem eliminar grande parte das entradas não significativas nas tabelas sintáticas, que respondem a um alto porcentual em relação ao número total de entradas. Os dois métodos substituem o acesso aleatório por uma busca linear e podem gerar reduções desnecessárias. Entende-se por redução desnecessária uma ação de redução não originalmente executada pelo analisador sintático ao utilizar a tabela original não compactada. Essas ações atrasam a detecção e dificultam o



tratamento de erro.

Os métodos RDS, SDS, RCS e SZS necessariamente detêm uma taxa de compressão menor, pois não fazem uso de ações padrões. Dentre esses quatro métodos, SZS apresenta a melhor taxa de compactação, embora sua complexidade de acesso seja no pior caso  $O(|value| - 1)$ . Uma alternativa a isto é a utilização do método RCS, que apresenta uma taxa de compressão ligeiramente menor e garante, no pior caso, custo  $O(|\Sigma|)$  e  $O(|N|)$  para acesso a uma entrada respectivamente em *Action* e *Goto*. Os métodos RDS e SDS são equivalentes do ponto de vista da taxa de compactação média e do custo de acesso à suas entradas da tabela, que é  $O(1)$ . Com isto, se analisados pela complexidade temporal, RDS e SDS são os métodos mais indicados quando se desejar uma compactação razoável ( $\approx 66\%$ ), mas com a preservação do custo de acesso original.

Os quatro métodos apresentados podem ser melhorados se linhas idênticas na tabela *Action* forem combinadas em uma única linha, já que o porcentual de linhas repetidas é em muitos casos superior a 20%, conforme discutido na apresentação do método ACS. Para isto, utiliza-se o método de compactação *Row Merging Scheme* - RMS. Neste método, a compressão é feita pela divisão das linhas da tabela *Action* em partições, onde cada partição corresponde a um conjunto de linhas idênticas entre si. Para cada partição  $p$ , insere-se uma nova linha  $k$  na matriz  $value_{rms}$  com os valores referentes a cada linha  $i$  em  $p$  e atribui-se  $k$  a  $rowpointer_{rms}[i]$ . Este vetor auxiliar mapeia uma linha  $i$  da tabela original na linha correspondente à partição na qual  $i$  pertence. Ao aplicar este esquema às tabelas *Action*, o acesso a uma entrada é dado por:

```

ACTION( $i, j$ )
1  return  $value_{rms}[rowpointer_{rms}[i], j]$ 

```

A matriz  $value_{rms}$  é então utilizada como entrada para os métodos RDS, SDS, RCS e SZS. O resultado da compactação RMS é mostrado na Tabela 4.22. Nessa tabela, a coluna *Acréscimo* indica a quantidade extra de bytes necessária à composição de RMS com qualquer método de compactação. Esse acréscimo é calculado por  $2 \times |rowpointer_{rms}|$ .

A composição de RMS com os métodos RDS, SDS, RCS e SZS permite um aumento médio de  $\approx 8\%$  na compactação de *Action*, conforme apresentado na Tabela 4.22. Para a tabela *Goto*, a composição não apresenta nenhum ganho.

Gramática	$ rowpointer_{rms} $	Lin. ( $value_{rms}$ )	Col. ( $value_{rms}$ )	Acréscimo (bytes)	Total (bytes)	Compactação (%)
C	352	274	85	704	47.284	21
C#	807	631	141	1.614	179.556	21,1
HTML	348	273	129	696	71.130	20,8
Java	632	537	107	1.264	116.182	14,1
VB .NET	636	465	144	1.272	135.192	26,2
<b>Média</b>						20,6

Tabela 4.21: Resultado da compactação das tabelas *Action* pelo método RMS.

Método	Compactação original (%)	Compactação (c/ RMS) (%)	Ganho (%)
RDS	46,8	57,2	10,4
SDS	50,3	61	10,7
RCS	69,1	73,7	4,6
SZS	70,3	75,6	5,3
<b>Ganho médio</b>			7,8

Tabela 4.22: Ganhos percentuais obtidos pela composição de RMS com os métodos RDS, SDS, RCS e SZS.

### 4.3 Compactação por Matriz

Os métodos de compactação por matriz diminuem o número de entrada da tabela sintática, mas preservam sua estrutura tabular. As entradas resultantes da compactação são armazenadas na matriz *value*. Da mesma forma que na apresentação dos métodos de compactação por vetor, a matriz  $value_a$  refere-se à tabela *Action* e  $value_g$  a *Goto*. Dois métodos desta categoria são apresentados: *Graph Coloring Scheme* [Schmitt, 1979] e *Line Elimination Scheme* [Bell, 1974].

#### 4.3.1 Método de Compactação por Coloração de Grafos (GCS)

Este método (*Graph Coloring Scheme*) [Schmitt, 1979] utiliza como estratégia de compactação a combinação de linhas que não possuem entradas significativas diferentes entre si para uma mesma coluna, o que caracteriza duas linhas não conflitantes. Este problema é modelado por coloração de grafos da seguinte forma: (i) representam-se as linhas da tabela *Action* como vértices do grafo; (ii) para cada linha  $i_1$  conflitante com uma linha  $i_2$ , cria-se uma aresta não direcionada entre os vértices  $i_1$  e  $i_2$ ; (iii) colore-se o grafo; (iv) combinam-se as linhas cujos vértices no grafo possuem a mesma cor. Disto, resulta uma nova tabela com  $gr_{min}$  linhas e mesmo número de colunas originais, onde

$gr_{min}$  é o número cromático necessário à coloração do grafo. Neste ponto, preenche-se um vetor auxiliar *rowmap* para mapear cada linha da tabela original na linha correspondente da tabela gerada. De forma a aumentar a taxa de compactação dessa tabela, faz-se uma nova coloração aplicada às colunas: vértices coloridos com uma mesma cor, indicam colunas não conflitantes entre si que podem ser combinadas. O resultado é a matriz *value*, com  $gr_{min}$  e  $gc_{min}$  entradas, onde  $gc_{min}$  é o número cromático referente à coloração das colunas. Análogo a *rowmap*, utiliza-se um vetor *columnmap* para mapear as colunas da tabela gerada na combinação de linhas para a respectiva coluna em *value*.

A compactação das tabelas *Action* e *Goto* ocorre da mesma forma, exceto que a primeira utiliza uma matriz auxiliar, denominada *Sigmap*, de forma a distinguir entre entradas significativas e não significativas. Isto é necessário, pois ao combinar linhas e colunas não conflitantes da tabela original, entradas não significativas podem sobrescrever entradas significativas em  $value_a$ . Dada uma entrada  $Action[i, j]$ ,  $Sigmap[i, j] = 1$  se e somente se  $Action[i, j]$  é significativa. Do contrário,  $Sigmap[i, j] = 0$ . Para obter uma entrada em  $Action[i, j]$ , procede-se da seguinte forma:

**ACTION**( $i, j$ )

```

1  if SIGMAP[ $i, j$ ]
2    then return  $value_a[rowmap_a[i], columnmap_a[j]]$ 
3  return error

```

Para as entradas em *Goto*, tem-se:

**GOTO**( $i, j$ )

```

1  return  $value_g[rowmap_g[i], columnmap_g[j]]$ 

```

A obtenção de uma coloração ótima é um problema NP-completo [Cormen et al., 2001]. Dürre [Dürre et al., 1976] propõe uma heurística para coloração de grafos que Dencker [Dencker et al., 1984] afirma ser adequada à compactação de tabelas sintáticas:

**GRAPH-COLORING-HEURISTIC**( $G$ )

```

1  while  $\exists$  vértice não colorido
2    do escolha o vértice v não colorido bloqueado pelo maior número de cores
3    colora v com a menor cor possível

```

Um vértice é bloqueado por uma cor se já possuir um vizinho colorido por ela. O algoritmo de compactação utilizado nos experimentos foi implementado segundo essa

heurística.

Um ponto importante neste método de compactação é a necessidade de se armazenar a matriz *Sigma* de forma compactada. Um método possível, proposto por Joliat [Joliat, 1974] para compactação de tabelas binárias, reduz o tamanho de *Sigma* da seguinte forma: duas linhas  $i_1$  e  $i_2$  idênticas entre si são combinadas, resultando em uma única linha  $k$  na matriz *Sigma'*, obtida no fim do processo. Neste caso, utiliza-se um vetor auxiliar *eq* de forma a identificar a combinação realizada:  $eq[i_1] = eq[i_2] = k$ . Repete-se o processo às colunas de *Sigma'*, armazenando-se a combinação de duas colunas em um vetor auxiliar *et*. O acesso a uma entrada da tabela *Sigma''*, obtida pela combinação de colunas a partir de *Sigma'*, é dado por:

SIGMAP( $i, j$ )

1 **return** *Sigma''*[*eq*[ $i$ ], *et*[ $j$ ]]

Os resultados da compressão da tabela *Sigma* utilizando o método de Joliat são apresentados na Tabela 4.23.

Gramática	Linhas	Colunas	<i>eq</i>	<i>et</i>	Total (bytes)	Compactação (%)
C	66	40	352	85	3.514	88,3
C#	133	64	807	141	10.408	90,9
HTML	122	72	348	129	9.738	78,3
Java	103	43	632	107	5.907	91,3
VB .NET	159	85	636	144	15.075	83,5
<b>Média</b>						86,4

Tabela 4.23: Compactação da tabela *Sigma* pelo método de Joliat.

Os resultados da compactação das tabelas *Action* e *Goto* utilizando via GCS são apresentados respectivamente nas Tabelas 4.24 e 4.25. O resultado final da compactação segundo este método é apresentado na Tabela 4.26.

### 4.3.2 Método de Compactação por Eliminação de Linhas (LES)

Este método (*Line Elimination Scheme*) [Bell, 1974] elimina alternadamente linhas e colunas cujas entradas significativas são iguais a um único valor ou possuem somente valores não significativos. Quatro vetores auxiliares são utilizados: *dr*, *r*, *dc* e *c*.

Inicialmente, escaneia-se a tabela de entrada de forma a buscar uma linha  $i$  com as características mencionadas. Se tal linha for encontrada e todas as suas entradas

Gramática	Linhas	Colunas	$ rowmap_a $	$ columnmap_a $	Total (bytes)	Compactação (%)
C	165	73	352	85	28.478	52,4
C#	370	92	807	141	80.384	64,7
HTML	171	103	348	129	45.918	48,9
Java	329	87	632	107	64.631	52,2
VB .NET	223	98	636	144	60.343	67,1
<b>Média</b>						57

Tabela 4.24: Resultados da compactação da tabela *Action* pelo método GCS.

Gramática	Linhas	Colunas	$ rowmap_g $	$ columnmap_g $	Total (bytes)	Compactação (%)
C	74	40	352	51	6.726	81,3
C#	88	74	807	131	14.900	93
HTML	67	24	348	26	3.964	78,1
Java	130	54	632	139	15.582	91,1
VB .NET	92	64	636	109	13.266	90,4
<b>Média</b>						86,8

Tabela 4.25: Resultados da compactação da tabela *Goto* pelo método GCS.

Gramática	Total (bytes)	Compactação (%)
C	35.204	63,2
C#	95.284	78,3
HTML	49.882	53,8
Java	80.213	74,2
VB .NET	73609	77,1
<b>Média</b>		69,3

Tabela 4.26: Resultado final da compactação das tabelas *Action* e *Goto* pelo método GCS.

corresponderem a um único valor significativo  $v$ , faz-se  $r[i]$  igual a  $v$ . Se as entradas em  $i$  forem todas não significativas, o valor em  $r[i]$  é irrelevante. Em ambos os casos, armazena-se em  $dr[i]$ , o valor do contador de escaneamentos realizados até o presente momento. As entradas em  $dr$  permitem identificar o momento de exclusão de uma dada linha. Em seguida verifica-se a possibilidade de eliminação de uma coluna, processo análogo à eliminação de linhas, com a diferença que os vetores manipulados são  $c$  e  $dc$ . O procedimento de eliminação alternada de linhas e colunas continua até que nenhuma linha possa ser eliminada. A matriz *value* contém as linhas e colunas da tabela original que não puderam ser eliminadas. Para cada linha  $i$  e coluna  $j$  nesta situação, faz-se

$dr[i]$  e  $dc[j]$  igual ao último escaneamento realizado, isto é, igual a  $s_{max}$  e armazena-se em  $r[i]$  e  $c[j]$  a linha e coluna correspondentes em *value*.

A compactação da tabela *Action* neste esquema não permite diferenciar entradas significativas das não significativas, pois os valores em  $r$  e  $c$  armazenam somente as entradas significativas de uma dada linha e coluna. Desta forma, é necessário utilizar a matriz *Sigmap*, tal como no esquema de compactação por coloração de grafos. O acesso a uma entrada em  $Action[i, j]$  é definido por:

**ACTION**( $i, j$ )

```

1  if Sigmap[ $i, j$ ]
2      then if  $dr_a[i] < dc_a[j]$ 
3          then return  $r_a[i]$ 
4          else if  $dc_a[j] < dr_a[i]$ 
5              then return  $c_a[j]$ 
6              else return  $value_a[r_a[i], c_a[j]]$ 
7  else return error

```

O acesso a  $Goto[i, j]$  é dado por:

**GOTO**( $i, j$ )

```

1  if  $dr_g[i] < dc_g[j]$ 
2      then return  $r_g[i]$ 
3  else if  $dc_g[j] < dr_g[i]$ 
4      then return  $c_g[i]$ 
5      else return  $value_g[r_g[i], c_g[j]]$ 

```

As Tabelas 4.27 e 4.28 apresentam os resultados da compactação via LES quando aplicado às tabelas *Action* e *Goto* das gramáticas de teste. O resultado final da compactação obtida é mostrado na Tabela 4.29.

### 4.3.3 Análise

Os métodos LES e GCS, por preservarem a estrutura tabular de *Action* e *Goto*, conservam o tempo de acesso da matriz original e não ocasionam reduções desnecessárias. Quanto à taxa de compactação, o método por eliminação de linhas é superior; nos testes realizados este método obteve vantagem média de  $\approx 6,7$  em relação a GCS.

Os métodos de compactação por matriz da tabela *Action*, assim como os de vetor, podem ser melhorados pela composição com RMS. Isto pode ser observado pelas taxa de

Gramática	Linhas	Colunas	$ dr_a $	$ dc_a $	$ r_a $	$ c_a $	<i>scan</i> <i>max</i>	Total (bytes)	Compactação (%)
C	163	62	352	85	352	85	5	25.474	57,43
C#	301	113	807	141	807	141	9	82.226	63,87
HTML	115	46	348	129	348	129	3	22.226	75,25
Java	230	77	632	107	632	107	7	44.283	67,26
VB .NET	196	129	636	144	636	144	3	65.643	64,16
<b>Média</b>									65,6

Tabela 4.27: Resultados da compactação da tabela *Action* pelo método LES.

Gramática	Linhas	Colunas	$ dr_g $	$ dc_g $	$ r_g $	$ c_g $	<i>scan</i> <i>max</i>	Total (bytes)	Compactação (%)
C	31	12	352	51	352	51	27	2.356	93,4
C#	118	40	807	131	807	131	29	13.192	93,8
HTML	45	4	348	26	348	26	5	18.56	89,7
Java	156	74	632	139	632	139	5	26.172	85,1
VB .NET	108	43	636	109	636	109	7	12.268	91,2
<b>Média</b>									90,6

Tabela 4.28: Resultados da compactação da tabela *Goto* pelo método LES.

Gramática	Total (bytes)	Compactação (%)
C	27.830	70,9
C#	95.418	78,3
HTML	24.082	77,7
Java	70.455	77,3
VB .NET	77.911	75,8
<b>Média</b>		76

Tabela 4.29: Resultado final da compactação das tabelas *Action* e *Goto* pelo método LES.

compactação apresentadas na Tabela 4.30. Para a tabela *Goto*, o ganho é inexpressivo.

Conforme a sugestão feita por Dencker em [Dencker et al., 1984], a composição dos métodos LES e GCS melhora a taxa de compactação tanto da tabela *Action* quanto *Goto*, o que foi comprovado em experimentos realizados. A combinação desses dois métodos é feita aplicando-se primeiramente a compactação por eliminação de linhas. Neste processo, são gerados os vetores  $dr$ ,  $dc$ ,  $r$  e  $c$ , além da matriz  $value_{LES}$ . Esta matriz é fornecida como entrada à compactação por coloração de grafos, que produz  $rowmap$ ,  $columnmap$  e  $value_{GCS}$ . Esses dois vetores, no entanto, podem ser descartados ao fazer  $r[i] = rowmap[r[i]]$  e  $c[i] = colmap[r[i]]$ , onde  $i$  e  $j$  representam linhas e colunas

Método (Action)	Compactação original (%)	Compactação (c/ RMS) (%)	Ganho (%)
LES	65,6	76,8	11,2
GCS	57	64,9	7,9
<b>Ganho médio</b>			<b>9,6</b>

Tabela 4.30: Ganhos percentuais obtidos pela composição de RMS com os métodos LES e GCS.

da tabela não compactada. Utiliza-se uma única matriz *Sigmap*, referente à tabela não compactada. No caso da tabela *Action*, o resultado pode ainda ser melhorada pela aplicação do método RMS anterior à eliminação de linhas. As Tabelas 4.31 e 4.32 mostram os resultados obtidos na compactação das tabelas de teste utilizando-se a combinação GCS ◦ LES ◦ RMS para *Action* e GCS ◦ LES para *Goto*. Na compactação de *Action*, a composição com RMS acarreta em uma pequena diminuição do espaço gasto por *Sigmap*, conforme mostrado na Tabela 4.33.

Gramática	Lin.	Col.	$ dr_a $	$ dc_a $	$ r_a $	$ c_a $	Extra	Total (bytes)	Compactação (%)
C	55	59	274	85	274	85	4.062	11.988	80
C#	67	84	631	141	631	141	11.670	26.014	88,6
HTML	45	44	273	129	273	129	10.284	15.852	82,3
Java	91	66	537	107	537	107	6.981	21.569	84,1
VB .NET	41	90	465	144	465	144	16.005	25.821	85,9
<b>Média</b>									<b>84,2</b>

Tabela 4.31: Resultados da compactação da tabela *Action* pela composição GCS ◦ LES ◦ RMS.

Gramática	Linhas	Colunas	$ dr_g $	$ dc_g $	$ r_g $	$ c_g $	Total (bytes)	Compactação (%)
C	25	6	352	51	352	51	1.912	94,7
C#	62	15	807	131	807	131	5.612	97,4
HTML	45	3	348	26	348	26	1.766	90,2
Java	130	30	632	139	632	139	10.884	93,8
VB .NET	80	29	636	109	636	109	7.620	94,5
<b>Média</b>								<b>94,1</b>

Tabela 4.32: Resultados da compactação da tabela *Goto* pela composição GCS ◦ LES.

A coluna “*Extra*” na Tabela 4.31 mostra o total de bytes gasto com o armazenamento de *Sigmap* e o vetor *rowmap* de RMS, cujo tamanho é o mesmo que *eq*.

A média final do esquema proposto é mostrada na Tabela 4.34.



Gramática	Linhas	Colunas	$ eq $	$ et $	Total (bytes)	Compactação (%)
C	66	40	274	85	3.358	88,8
C#	133	64	631	141	10.056	91,2
HTML	122	72	273	129	9.588	78,6
Java	103	43	537	107	5.717	91,6
VB .NET	159	85	465	144	14.733	83,9
<b>Média</b>						86,8

Tabela 4.33: Compactação da tabela *Sigmap* pelo método de Joliat aplicada às tabelas produzidas por RMS.

Gramática	Total (bytes)	Compactação (%)
C	13.900	85,5
C#	31.626	92,8
HTML	17.618	83,7
Java	32.453	89,6
VB .NET	33.441	89,6
<b>Média</b>		88,2

Tabela 4.34: Resultado final da compactação obtida pela composição GCS ◦ LES ◦ RMS (*Action*) e GCS ◦ LES (*Goto*).

O resultado final é melhor quando comparado aos métodos de compactação por vetor RDS, SDS, RCS e SZS, com mais de 8% de vantagem em relação ao melhor desses métodos (SZS). A grande vantagem dessa combinação é prover uma alta taxa de compactação, inferior às obtidas por ACS e BCS, mas com a preservação do tempo de acesso teórico  $O(1)$  e a não realização de ações desnecessárias. Além disso, a combinação apresentada é superior em 18% e 12% respectivamente se contrastada aos métodos GCS e LES aplicados isoladamente.

## 4.4 Compactação de Tabelas LALR– $k$ variável

Uma tabela *Action* gerada para um analisador LALR( $k$ ), com  $k$  variável, conforme descrito na Seção 3.4.2.1, pode ser compactada por todos os métodos descritos neste capítulo, desde que se considere ações de *lookahead shift* como entradas significativas.

Como os analisadores que não fixam  $k$  não alteram a tabela *Goto*, os métodos de compactação apresentados podem operar diretamente nessas tabelas.

## 4.5 Conclusão

Este capítulo apresentou oito métodos de compactação de tabelas sintáticas LALR(1). Necessário à realização dos experimentos, definiu-se a codificação das entradas e mostrou-se como a mesma permite a identificação das ações sintáticas presentes nas tabelas *Action* e *Goto*.

Os métodos apresentados foram divididos em dois conjuntos, cada um segundo a forma de representação da tabela compactada: os que linearizam a tabela em um grande vetor (ACS, BCS, RDS, SDS, RCS e SZS), denominados métodos de compactação por vetor, e os que a representam em uma matriz compacta (GCS e LES), denominados métodos de compactação por matriz.

Dos testes realizados com cada um desses métodos, verificou-se que o método BCS é o melhor em termos de compactação (superior a 95%), apesar da substituição do acesso aleatório pela busca linear e à possível realização de reduções desnecessárias, o que dificulta o processo de recuperação de erro.

Foi mostrado que o método RMS é útil na melhoria dos métodos, tanto os de vetor quanto os de matriz, pois elimina linhas idênticas em *Action*, fato recorrente em tabelas LALR(1).

Dentre os métodos de compactação por matriz, o método LES mostrou a melhor taxa de compactação. A vantagem de se utilizar os métodos de matriz é na verdade o ganho proporcionado pela combinação deles. Mostrou-se que pela composição dos métodos GCS ◦ LES ◦ RMS (compactação de *Action*) e GCS ◦ LES (compactação de *Goto*) é possível obter uma compressão de  $\approx 87\%$ , com a preservação do tempo original de acesso e detecção de erro na mesma velocidade em que o analisador não compactado.

A utilização dos métodos BCS e a combinação de GCS, LES e RMS permitem obter os níveis de compactação alto e médio mencionados nesta dissertação.

Os métodos descritos neste capítulo podem ser utilizados na compactação de tabelas *Action* de analisadores LALR( $k$ ), onde  $k$  é variável, desde que se considere ações de *lookahead shift* como entradas significativas.

# Capítulo 5

## Ferramenta SAIDE

SAIDE (*Syntax Analyser Integrated Development Environment*) é uma ferramenta para construção e geração de analisadores LALR( $k$ ) com o objetivo de amenizar a dificuldade e diminuir o tempo gasto na remoção de conflitos em gramáticas não LALR. Isto é alcançado por um conjunto de facilidades que permitem ao usuário a aplicação de uma metodologia proposta nesta dissertação. Dois outros pontos importante são a flexibilização da escolha do nível de compactação das tabelas sintáticas produzidas de acordo com o perfil da aplicação e o suporte a múltiplas linguagens de especificação.

A apresentação da ferramenta SAIDE é feita da seguinte forma: a metodologia de suporte à remoção de conflitos é descrita e as funcionalidades e telas da ferramenta são apresentadas. Em seguida, faz-se a explanação sobre sua arquitetura e como analisadores sintáticos LALR( $k$ ) são gerados no ambiente disponibilizado. Feito isto, descrevem-se o funcionamento interno e os algoritmos implementados.

### 5.1 Metodologia Proposta

A metodologia proposta consiste em uma seqüência de passos a serem seguidos durante a remoção de um conflito em uma gramática não LALR. Esta metodologia separa os conflitos em dois grandes conjuntos: os que podem ser automaticamente removidos e os que devem ser solucionados manualmente, conforme a organização apresentada na Figura 5.1.

A remoção automática de conflitos visa diminuir o esforço despendido pelo usuário. Das situações enumeradas na Figura 3.1, sabe-se que parte dos conflitos decorrentes da falta de contexto à direita podem ser automaticamente solucionados aumentando-se o número de *lookaheads* inspecionados, conforme explicado na Seção 3.2.

Para os conflitos que não podem ser automaticamente removidos, a metodologia estabelece que os mesmos sejam solucionados segundo uma ordem de prioridade, que

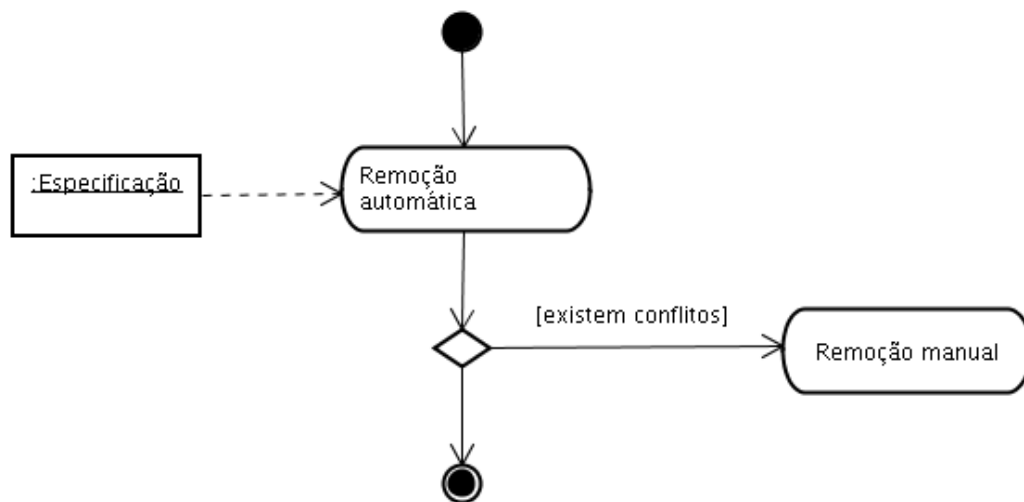


Figura 5.1: Organização da metodologia proposta.

indica a seqüência em que os conflitos devem ser eliminados. Nesta ordenação, um conflito deve ser removido antes de um outro se este for consequência do primeiro. Espera-se com isto, que a remoção de um dado conflito implique na remoção de todos os outros dele decorrentes.

A metodologia define quatro fases para remoção manual de um dado conflito selecionado segundo a ordenação de prioridade: *entendimento*, *classificação*, *alteração da especificação* e *testes*. A Figura 5.2 mostra como a metodologia organiza essas fases. Durante a remoção manual, a metodologia tem o conflito como um conflito em LALR(1). Isto torna o processo de remoção uniforme e faz com que o projetista sempre busque uma solução que utilize o menor número de *lookaheads* possível. Como consequência desta decisão, espera-se que os analisadores sintáticos obtidos sejam sempre eficientes.

**Entendimento:** a fase de entendimento tem por objetivo auxiliar a dedução da causa do conflito. Para amenizar as dificuldades advindas da análise dos dados pelo usuário na leitura do arquivo de *log*, a metodologia estabelece a visualização modularizada desses dados e um mecanismo apropriado para interligá-los. Acredita-se que isto provê ao usuário uma melhor navegação e reduz o tempo gasto no entendimento de um conflito. O conjunto mínimo de dados a ser visualizado é:

- arquivo de especificação sintática;
- autômato LALR, onde para cada estado exibem-se os itens que o constituem e as transições a outros estados;

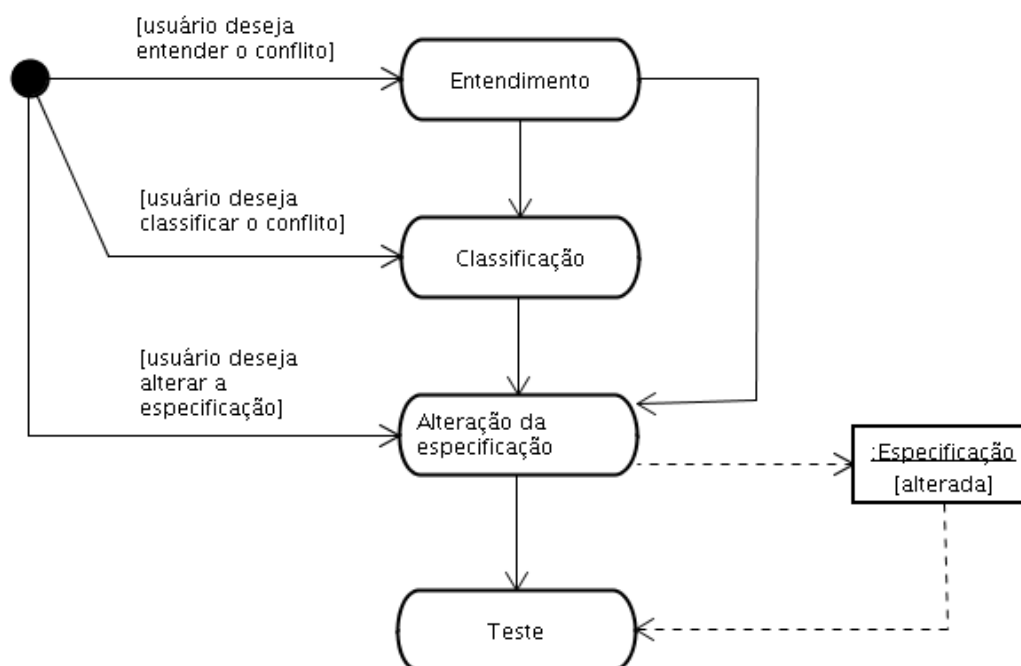


Figura 5.2: Fases da *Remoção manual* da metodologia proposta.

- indicação do total de conflitos encontrados;
- listagem dos conflitos encontrados segundo uma ordem de prioridade. Para cada conflito exibe-se o número do estado em que o mesmo ocorre, os itens envolvidos e o símbolo que o causa.

Muitas vezes, no entanto, ainda que se tenha um mecanismo de visualização modularizada e interligada dos dados do arquivo de *log*, o tempo despendido na análise desse conteúdo ainda é consideravelmente alto, especialmente se o usuário não é proficiente na teoria subjacente a analisadores LALR. Portanto, uma importante característica desta fase é prover estruturas de maior nível de abstração que permitam uma análise mais rápida e apropriada. O uso de árvores de derivação no entendimento de conflitos atendem perfeitamente a este requisito, pois aproximam o usuário de seu objeto real de estudo – a sintaxe da linguagem em questão, ao mesmo tempo em que reduz o montante de conhecimento teórico necessário à remoção de conflitos.

**Classificação:** esta fase da remoção manual visa identificar uma das situações exibidas na Figura 3.1 responsável pela ocorrência do conflito. Isto é justificado pela convicção de que uma solução utilizada na remoção de um conflito anterior, decorrente da situação identificada, pode ser novamente aplicada na remoção do conflito corrente

ou pelo menos propiciar o raciocínio que leve ao planejamento de uma estratégia de remoção.

As situações que ocorrem conflitos são organizadas conforme exibido na Figura 5.1. Se o conflito for identificado como decorrente da situação (i) ou (ii) da figura apresen-

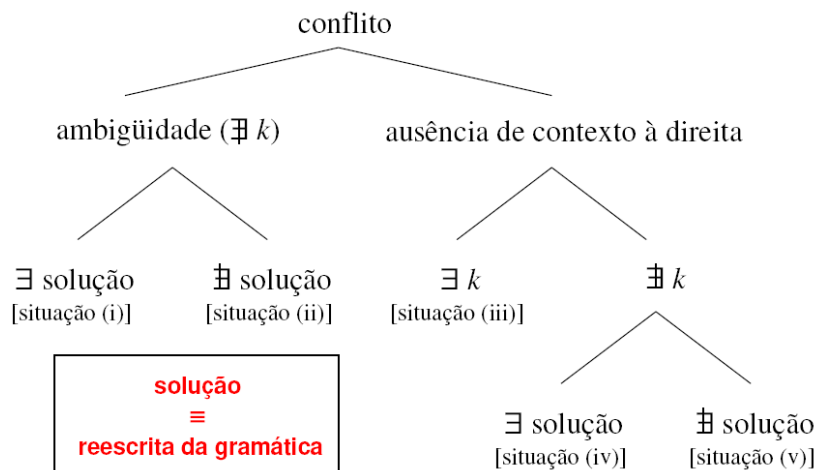


Figura 5.3: Situações em que um dado conflito ocorre.

tada isto é, um conflito de ambigüidade para o qual pode ou não existir uma reescrita da gramática sem alteração da linguagem, então o usuário deve ser assistido com exemplos de ambigüidades recorrentes em gramáticas de linguagens de programação, juntamente com possíveis soluções, de forma a adaptá-las ao seu contexto. Essas soluções não necessariamente preservam a linguagem, embora isto seja sempre desejado.

No caso de conflitos classificados de acordo com a situação (iii), o usuário ou reescreve sua gramática de forma que as ações sintáticas sejam univocamente determinadas pela inspeção de  $k$  tokens ou aumenta o valor de  $k$  na tentativa de remover automaticamente o conflito na primeira parte da metodologia.

A metodologia não dá suporte aos conflitos ocorridos de acordo com as situações (iv) e (v).

**Alteração da especificação:** conhecida a causa do conflito e identificada uma estratégia de remoção, nesta fase o usuário altera a especificação sintática na tentativa de remover o conflito. A estratégia de remoção é utilizada de acordo com a indicação

**Teste:** a última fase da remoção manual consiste na verificação da remoção do conflito. A Figura 5.4 mostra a organização desta fase.

Para testar, o usuário resubmete a especificação sintática ao gerador de analisador sintático. Como a estratégia utilizada pelo usuário pode ter ocasionado novos conflitos, a especificação é reprocessada em uma nova tentativa de remoção automática de

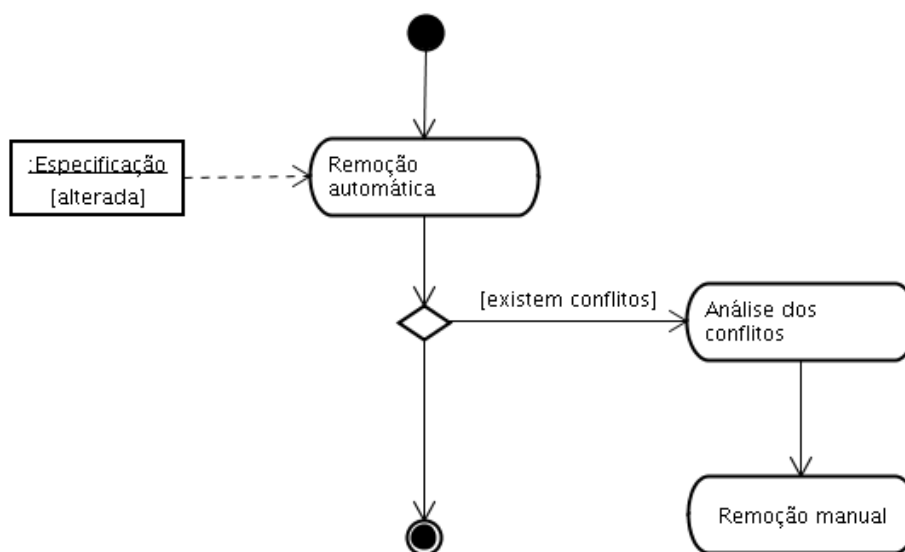


Figura 5.4: Organização da fase de testes.

conflitos. Caso isto não seja possível, retorna-se a listagem dos conflitos encontrados, juntamente com o seu total. Neste ponto, o usuário analisa os resultados de forma a garantir a eliminação do conflito original. Caso a remoção tenha falhado, três ações são possíveis:

- retornar à primeira fase da remoção manual, de forma a reestudar a causa do conflito;
- retornar à classificação para replanejar a estratégia de remoção;
- tentar uma nova estratégia sem passar pelas fases de entendimento e classificação.

## 5.2 Ambiente da Ferramenta

Para construção de analisadores  $LALR(k)$ , a ferramenta disponibiliza o ambiente mostrado na Figura 5.5.

O editor de textos integrado, exibido na Figura 5.6, permite a edição de especificações sintáticas, com suporte a *syntax highlight* e operações comuns tais como posicionamento de linha e coluna, desfazer, refazer, copiar, recortar, colar, etc.

No fim da edição, o usuário solicita o processamento da especificação. Neste ponto, SAIDE verifica a estrutura léxica e sintática da especificação e lista todos os erros decorrentes dessa verificação. Após a correção dos erros apresentados, o usuário resubmete a especificação e caso não sejam encontrados erros léxicos e/ou sintáticos, SAIDE constrói o autômato  $LALRA_k$ , com  $k$  variável, necessário à resolução automática de

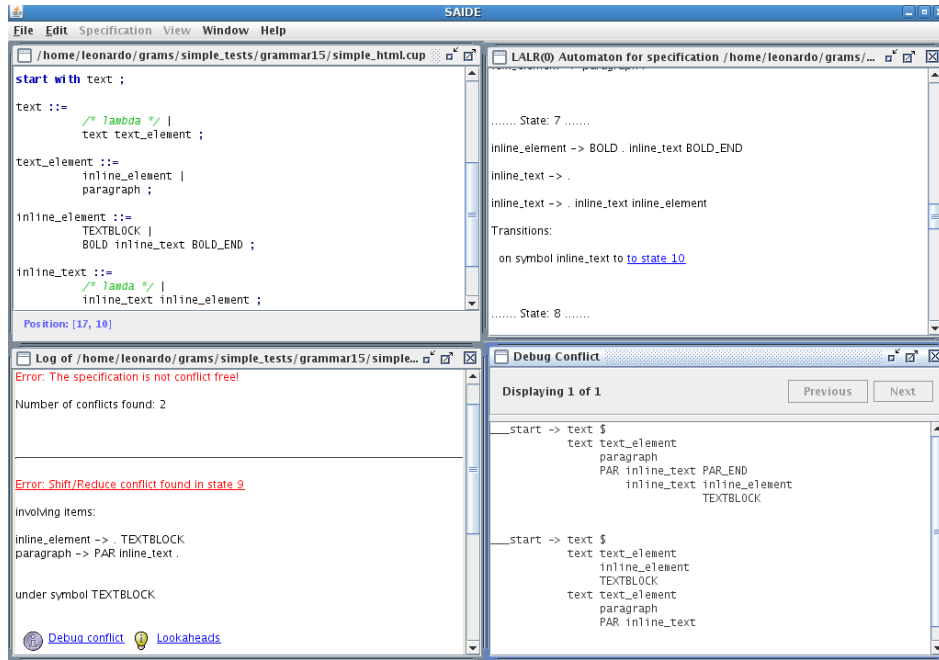


Figura 5.5: Tela principal da ferramenta SAIDE.

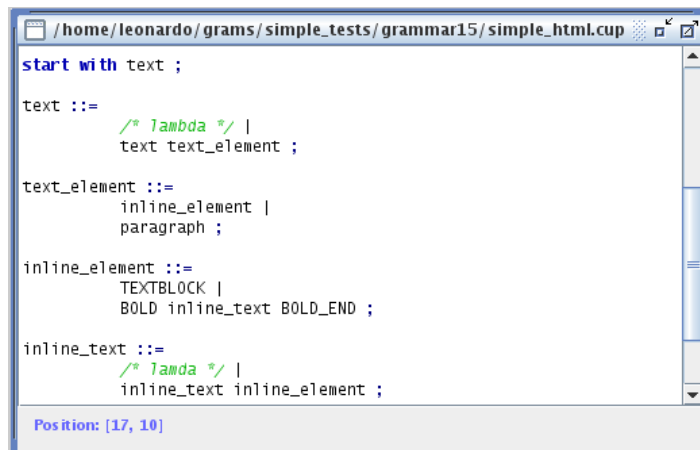


Figura 5.6: Janela de edição da ferramenta SAIDE.

conflitos. O valor de  $k$ , neste caso é local e útil somente à resolução de um dado conflito e é sempre menor ou igual a  $k_{max}$ , uma variável definida no arquivo de configuração da ferramenta e cujo valor padrão é 3. A utilização de  $k$  variável é capaz de resolver parte dos conflitos ocasionados pela falta de contexto à direita quando existe um valor de  $k \leq k_{max}$  para qual o conflito é removido.

Para quaisquer outros conflitos, os mesmos são ordenados e listados, conforme a prioridade em que devem ser resolvidos. Um exemplo de uma possível listagem é apresentado na Figura 5.8. Neste ponto, o usuário remove os conflitos manualmente.

O suporte dado pela ferramenta a cada uma das fases da remoção manual é discutido



a seguir.

### 5.2.1 Entendimento

Para cada conflito listado, SAIDE oferece três opções, exemplificadas em função do conflito no topo da listagem na Figura 5.7:

- visitar o estado do autômato no qual o conflito ocorre. Para isto, o usuário clica no *hiperlink* em vermelho posicionado na primeira linha onde o conflito é reportado. Quando isto ocorre, a janela da Figura 5.8 é exibida;
- depurar o conflito, isto é, visualizar as árvores de derivação de cada item do conflito. A sua ativação é dada pelo hiperlink localizado à esquerda da última linha referente ao conflito, rotulado por *Debug Conflict*. Um exemplo de depuração é mostrado na Figura 5.9;
- visualizar o conjunto de *lookaheads* de tamanho  $k$  utilizados na tentativa de se remover o conflito. Isto é utilizado pelo usuário como parte da fase de classificação, discutida na próxima seção.

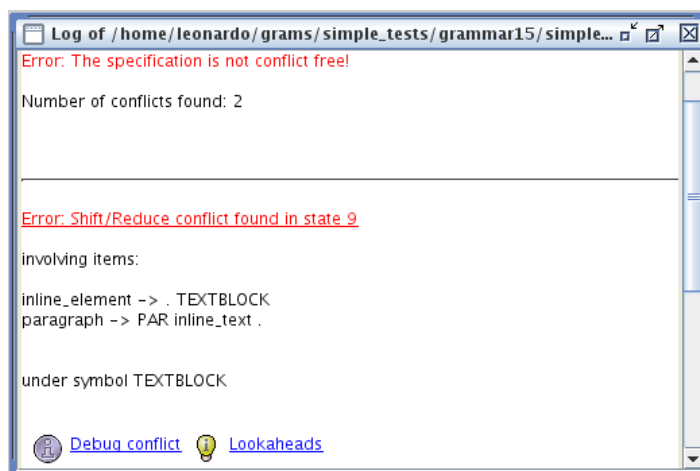


Figura 5.7: Janela de conflitos da ferramenta SAIDE: listagem das mensagens referentes ao processamento da especificação sintática.

### 5.2.2 Classificação

A classificação de um conflito é atualmente feita pelo usuário, a partir de informações fornecidas pela ferramenta. A primeira delas, informada na janela de depuração do conflito, é a afirmação se o conflito é específico ou não a LALR. Isto permite ao usuário identificar se a causa do conflito é devido à presença de ambigüidade, conforme definido

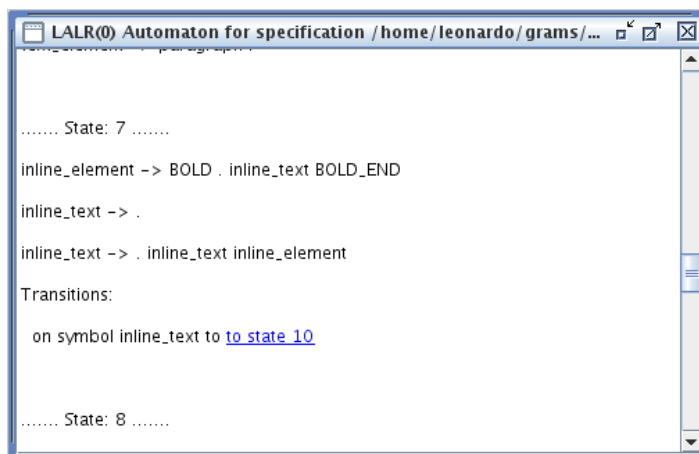


Figura 5.8: Janela de visualização do autômato LALR(0) na ferramenta SAIDE.

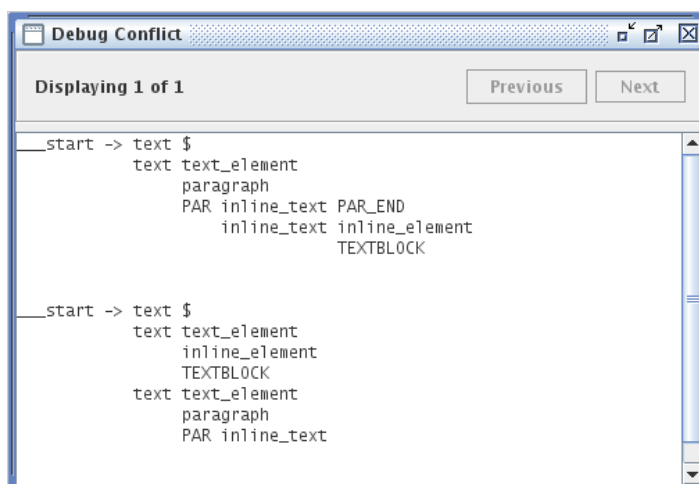


Figura 5.9: Janela de depuração de conflito na ferramenta SAIDE: árvores de derivação para explanação do conflito.

pelo corolário 1. Na janela de *lookaheads* são mostrados os *lookaheads* de tamanho  $k \leq k_{max}$  para os quais o conflito permanece. Com base nestas informações e no conhecimento que o usuário detêm acerca da gramática, espera-se que seja possível a classificação do conflito de acordo com as possíveis situações apresentadas na Figura 3.1. Para o caso específico de conflitos ocasionados por ambigüidade, a ferramenta disponibiliza um conjunto de exemplos de ambigüidades recorrentes em linguagens de programação e suas possíveis soluções, de forma que o usuário possa adaptá-las a seu contexto. Esse conjunto de exemplos, obtido a partir de ambigüidades presentes nas gramáticas das linguagens Notus, Oberon-2, Algol-60 e Scheme, é estático. No entanto, o usuário é encorajado a expandí-lo de forma a conter casos diferente dos originalmente considerados. Os exemplos são indexados por padrões. Um padrão é uma abstração mais geral sobre uma instância específica que representa um conflito de ambigüidade.

O não terminal  $P_i$  é utilizado na definição dos padrões catalogados, que são atualmente quatro (assume-se que  $S \xRightarrow{*} \xi'P\xi''$  e que  $\alpha$  e  $\beta$  não contém o símbolo  $P$ ):

1.  $P \xRightarrow{*} P\alpha P$  e  $P \xRightarrow{*} \beta$ : este padrão captura construções ambíguas como as em

$$\begin{array}{l} \text{exp} \rightarrow \text{exp binop exp} \\ | \\ \text{num} \end{array}$$

2.  $P \xRightarrow{*} \alpha P$  e  $P \xRightarrow{*} \alpha P \beta P$ : este padrão identifica conflitos análogos ao conflito decorrente da construção sintática do *dangling-else*.

3.  $P \xRightarrow{*} \alpha P \beta P$  e  $P \xRightarrow{*} P \beta P$ : este padrão captura construções ambíguas como as em

$$\begin{array}{l} \text{exp} \rightarrow \text{let dcl in exp where exp} \\ | \\ \text{exp where exp} \end{array}$$

4.  $P \xRightarrow{*} \alpha P_1 \beta \xRightarrow{*} \alpha \gamma \beta$  e  $P \xRightarrow{*} \alpha P_2 \beta \xRightarrow{*} \alpha \gamma \beta$ : este padrão captura o caso em que dois não terminais produzem formas sentenciais em comum, isto é:  $\mathcal{L}(P_1) \cap \mathcal{L}(P_2) \neq \emptyset$ , onde  $\mathcal{L}$  denota os *strings* gerados a partir de um não terminal.

Exceto pelo terceiro padrão, as soluções dos conflitos representados pelos padrões disponibilizados ao usuário preservam a linguagem original.

### 5.2.3 Alteração da especificação

Nesta fase o usuário edita a gramática no editor disponibilizado na ferramenta. Isto consiste na codificação da estratégia de remoção planejada a partir da classificação. Para o caso de um conflito de ambigüidade para o qual tem-se um padrão catalogado, a alteração consiste na adaptação das soluções mostradas como exemplos. Para os padrões mencionados, tem-se as seguintes soluções:

**Padrão:**  $P \xRightarrow{*} P\alpha P$  e  $P \xRightarrow{*} \beta$ .

A solução para este tipo de conflito é estabelecer níveis na gramática de forma a definir a precedência de cada construção. No caso de se ter operadores, pode ser necessário também definir a associatividade de cada um deles. Como exemplo deste padrão, considere a gramática de expressões:

$$\begin{array}{l}
 \text{exp} \quad \rightarrow \quad \text{exp binop exp} \\
 \quad \quad | \quad \mathbf{num} \\
 \text{binop} \quad \rightarrow \quad + \\
 \quad \quad | \quad - \\
 \quad \quad | \quad / \\
 \quad \quad | \quad *
 \end{array}$$

Assim como nos demais padrões, as regras apresentadas são diretamente mapeáveis no padrão em questão. Na prática, no entanto, a identificação de qual padrão deve ser utilizado nem sempre é direta. Nestes casos, o usuário deve valer-se da consulta às árvores de derivação obtidas na fase de entendimento.

A solução para remover a ambigüidade dessa gramática é estabelecer a precedência de cada operador. O nível de um operador em uma árvore de derivação é diretamente proporcional à sua precedência, ou seja, quanto maior a precedência de um operador, mais fundo ele deve estar em um árvore de derivação. Pelos operadores mostrados, multiplicação e divisão possuem maior precedência que os operadores de adição e subtração. A precedência entre os operadores de multiplicação e divisão é a mesma; com isto, preserva-se a ordem em que aparecem. Isto também é válido para os operadores de adição e subtração. Por este raciocínio, a nova gramática é expressa por:

$$\begin{array}{l}
 \text{exp} \quad \quad \quad \rightarrow \quad \text{exp additive-op term} \\
 \quad \quad \quad \quad | \quad \text{term} \\
 \text{term} \quad \quad \quad \rightarrow \quad \text{term multiplicative-op num} \\
 \quad \quad \quad \quad | \quad \mathbf{num} \\
 \text{additive-op} \quad \rightarrow \quad + \\
 \quad \quad \quad \quad | \quad - \\
 \text{multiplicative-op} \rightarrow \quad / \\
 \quad \quad \quad \quad | \quad *
 \end{array}$$

A gramática obtida não é ambígua. Note que os operadores utilizados possuem associatividade à esquerda, o que é preservado pela gramática. A associatividade está diretamente ligada à recursão dos não terminais de cada regra.

**Padrão:**  $P \xRightarrow{*} \alpha P$  e  $P \xRightarrow{*} \alpha P \beta P$ . O exemplo clássico deste padrão é a ambigüidade conhecida como *dangling else*, cuja gramática característica é dada pelas regras:

$$\begin{array}{l}
stmt \quad \rightarrow \quad if-stmt \\
\quad \quad \quad | \quad other-stmt \\
if-stmt \quad \rightarrow \quad \mathbf{if} \ exp \ \mathbf{then} \ stmt \ else-stmt \\
else-stmt \quad \rightarrow \quad \mathbf{else} \ stmt \\
\quad \quad \quad | \quad \lambda
\end{array}$$

A ambigüidade é removida ao corresponder cada **else** ao **then** anterior mais próximo ainda não correspondido, que é a interpretação mais comum para enunciados condicionais em linguagens de programação. A idéia da reescrita está em que um *stmt* entre um **then** e um **else** precisa ser “correspondido” (*matched*), isto é, não pode terminar com um **then** ainda não correspondido seguido por qualquer *stmt*, pois, do contrário, o **else** corresponderia ao **then** não correspondido. Um *stmt* correspondido (*matched-stmt*) ou é enunciado *if-then-else* com somente *stmts* correspondidos ou é qualquer tipo de *stmt* incondicional [Aho et al., 1986]. Com isto, tem-se:

$$\begin{array}{l}
smt \quad \quad \quad \rightarrow \quad matched-stmt \\
\quad \quad \quad | \quad unmatched-stmt \\
matched-stmt \quad \rightarrow \quad matched-if \\
\quad \quad \quad | \quad other-stmt \\
matched-if \quad \quad \rightarrow \quad \mathbf{if} \ exp \ \mathbf{then} \ matched-stmt \ \mathbf{else} \ matched-stmt \\
unmatched-stmt \quad \rightarrow \quad \mathbf{if} \ exp \ \mathbf{then} \ stmt \\
\quad \quad \quad | \quad \mathbf{if} \ exp \ \mathbf{then} \ matched-stmt \ \mathbf{else} \ unmatched-stmt
\end{array}$$

**Padrão:**  $P \xRightarrow{*} \alpha P \beta P$  e  $P \xRightarrow{*} P \beta P$ .

A solução para este padrão consiste na utilização de um *token* delimitador<sup>1</sup>. Neste caso, a reescrita da gramática altera a linguagem em questão. Como exemplo, considere a gramática abaixo:

$$\begin{array}{l}
exp \quad \rightarrow \quad \mathbf{let} \ dcl \ \mathbf{in} \ exp \ \mathbf{where} \ exp \\
\quad \quad \quad | \quad exp \ \mathbf{where} \ exp
\end{array}$$

Uma possível solução é restringir que *exp where exp* esteja contido em parênteses quando o lado direito de *exp* for referente a uma expressão *let*. Assim, a nova gramática é dada por:

---

<sup>1</sup>Este padrão foi extraído da gramática de Notus [Tirelo e Bigonha, 2006]. Para remover o conflito em questão, os autores da linguagem alteraram a sintaxe original pela utilização de um *token* delimitador.

$$\begin{aligned} exp &\rightarrow \text{let } "(" dcl \text{ in } exp \text{ where } exp \text{ } ")" \\ &| \quad exp \text{ where } exp \end{aligned}$$

**Padrão:**  $P \xRightarrow{*} P_1 \xRightarrow{*} \alpha$  e  $P \xRightarrow{*} P_2 \xRightarrow{*} \alpha$ .

Como exemplo deste padrão e sua correspondente solução, considere a seguinte gramática:

<sup>2</sup>

$$\begin{aligned} expression &\rightarrow subscripted-variable \\ &| \quad simple-designational-expression \\ subscripted-variable &\rightarrow \text{id } "[" subscript-expression \text{ } "]" \\ simple-designational-expression &\rightarrow \text{id } "[" subscript-list \text{ } "]" \\ subscript-list &\rightarrow subscript-expression \\ &| \quad subscript-list \text{ } "," \text{ } subscript-expression \end{aligned}$$

A solução para remoção da ambigüidade é fornecida pela reescrita:

$$\begin{aligned} expression &\rightarrow subscripted-var-or-simple-desig-exp \\ &| \quad simple-designational-expression2 \\ subscripted-var-or-simple-desig-exp &\rightarrow \text{id } "[" subscript-expression \text{ } "]" \\ simple-designational-expression2 &\rightarrow \text{id } "[" content \text{ } "]" \\ content &\rightarrow subscript-expression \text{ } "," \text{ } subscript-list \end{aligned}$$

Na nova gramática, expandem-se todas as possibilidades no lado direito das produções cujo lado esquerdo é *expression*. Para o exemplo mostrado, o primeiro caso ocorre quando é possível ter tanto um *subscripted-variable* quanto um *simple-designational-expression*. Para isto, tem-se o não terminal *subscripted-var-or-simple-desig-exp*. No segundo caso, a única possibilidade é um *simple-designational-expression*. Se isto ocorrer, então tem-se pelo menos dois *subscript-expression* delimitados por colchetes e separados por vírgula.

Note este padrão captura *alias* entre não terminais, como no fragmento abaixo, extraído de Algol-60:

---

<sup>2</sup>A gramática apresentada é uma simplificação de um conflito de ambigüidade na parte de expressões da gramática de Algol-60.

<i>actual-parameter</i>	→	<i>procedure-identifier</i>
		<i>procedure-identifier</i>
		<i>switch-identifier</i>
		<i>array-identifier</i>
<i>procedure-identifier</i>	→	<b>id</b>
<i>switch-identifier</i>	→	<b>id</b>
<i>array-identifier</i>	→	<b>id</b>

Neste caso, as regras devem ser combinadas:

<i>actual-parameter</i>	→	<i>procedure-or-switch-or-array-identifier</i>
<i>procedure-or-switch-or-array-identifier</i>	→	<b>id</b>

#### 5.2.4 Testes

Para dar suporte à fase de testes o usuário analisa as mensagens resultantes do reprocessamento da especificação, conforme mostrado Figura 5.7, comparando o resultado corrente com o anterior. Isto permite verificar se a remoção foi bem sucedida.

## 5.3 Arquitetura

SAIDE possui uma arquitetura formada por três componentes internos à ferramenta: *saideui.jar*, *alligator.jar* e *plugin.jar*, conforme exibido na Figura 5.10.

O componente *saide.jar* agrupa as classes de fronteira e controle responsáveis pelas interfaces gráficas e sua lógica de controle.

As classes no componente *alligator.jar* detém o código para computação do autômato LALR( $k$ ), da construção das tabelas *Action* e *Goto*, dos algoritmos de compactação de tabela e da ordenação da listagem dos conflitos segundo a prioridade em que devem ser resolvidos. Classes nesse componente são instanciadas pelos objetos de controle em *saideui.jar*.

O componente *plugin.jar* forma um arcabouço a partir do qual permite aos componentes *saide.jar* e *alligator.jar* serem independentes da linguagem de especificação utilizada na escrita de analisadores sintáticos. O arcabouço foi projetado para suportar linguagens de especificação similares ao do YACC. Esta divisão permite a utilização da ferramenta de maneira uniforme e independente da linguagem de especificação, desde que se tenha o *plugin* adequado.

Os *plugins* construídos por usuários constituem classes carregadas dinamicamente pela ferramenta, sem a necessidade de recompilação de código. A responsabilidade

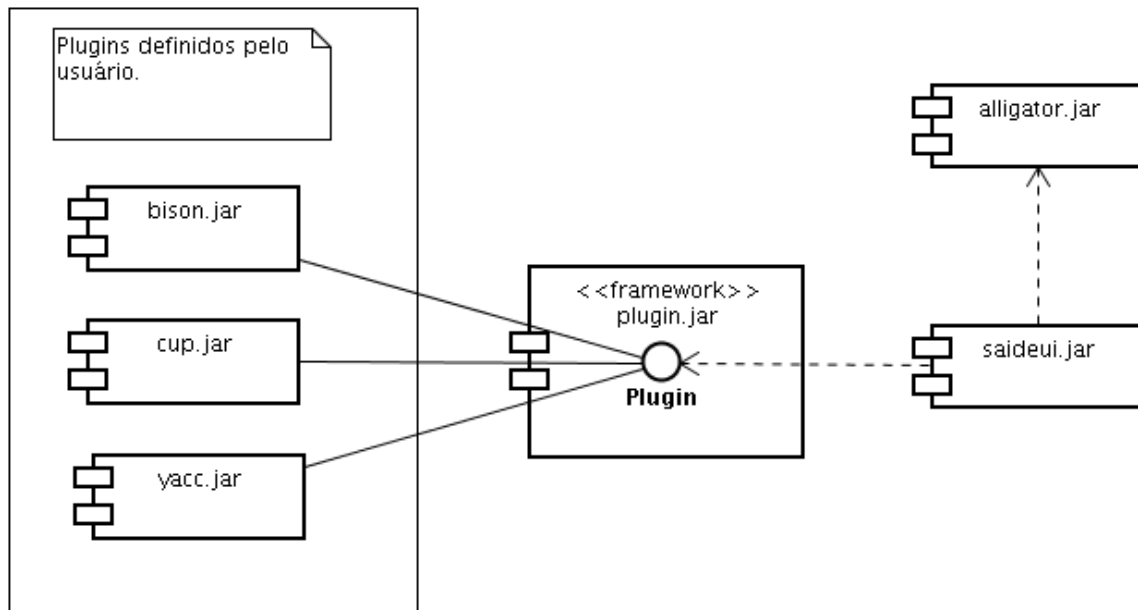


Figura 5.10: Arquitetura da ferramenta SAIDE.

dessas classes é realizar a análise léxica e sintática de especificações escritas em uma linguagem específica a um dado gerador. Todo *plugin* fornece uma classe que é um subtipo de `Plugin`, uma interface contida em `plugin.jar`. Esta interface define a forma de comunicação entre os *plugins* e a ferramenta. Na arquitetura apresentada, são mostrados três possíveis *plugins*: `bison.jar`, `cup.jar` e `yacc.jar`.

Os componentes implementados atualmente são `saideui.jar`, `plugin.jar`, `cup.jar` e `alligator.jar`, totalizando 20.115 linhas de código na linguagem Java.

## 5.4 Geração do Analisador Sintático

Ao ser solicitado a geração do analisador sintático, SAIDE compacta a tabela sintática segundo dois níveis possíveis: médio e alto. No nível alto tem-se um aumento de execução do analisador gerado, mas com a tentativa de se maximizar a compressão. O algoritmo de compactação utilizado neste nível é o BCS. No nível médio preserva-se a complexidade temporal do analisador, embora a compactação seja inferior à obtida pelo nível alto. O algoritmo utilizado é formado pela composição de GCS o LES o RMS na compactação da tabela *Action* e e GCS o LES para *Goto*.

Em seguida, SAIDE repassa ao *plugin* carregado em memória a tabela compactada e os objetos que correspondem aos dados na especificação do usuário, para que se grave o arquivo com o código do analisador sintático. O *plugin* utilizado possui a responsabilidade de produzir o analisador sintático com a mesma interface que os originalmente



criados pelo gerado ao qual o *plugin* está relacionado.

## 5.5 Algoritmos

Nesta seção são descritos os algoritmos implementados de forma a permitir a aplicação da metodologia proposta na Seção 5.1, juntamente com a explanação de como a ferramenta gera analisadores LALR( $k$ ).

**Resolução automática de conflitos:** anterior à remoção manual, SAIDE tenta automatizar parte do processo de remoção pela eliminação automática de conflitos resultantes da falta de contexto à direita. Para isto, SAIDE constrói analisadores LALR( $k$ ), com  $k$  variável, conforme explicado no Capítulo 3.

Os algoritmos utilizados na ferramenta SAIDE são baseados nos propostos por Charles [Charles, 1991], mas com a diferença de que o término de execução é garantido independentemente de qualquer característica da gramática e/ou da presença de ciclos nos grafos das relações *reads* e *includes*. Isto permite aplicar a tentativa de remoção a todos os possíveis conflitos, tratando-os uniformemente.

Na simulação dos passos do autômato LR(0) para descobrir os *lookaheads* de tamanho menor ou igual a  $k_{max}$ , utiliza-se uma estrutura de árvore para armazenar as pilhas obtidas a cada passo de execução [Kristensen e Madsen, 1981]. Cada nodo nessa árvore guarda sempre um estado. Dado um nodo  $n$ , as operações do TAD *Pilha* são feitas da seguinte forma:

- topo da pilha: corresponde ao estado armazenado em  $n$ ;
- conjunto de estados da pilha: é formado pelos estados armazenados nos nodos do caminho formado de  $n$  até a raiz da árvore (todo nodo mantém uma referência ao seu nodo pai);
- empilhamento de um estado  $p$ : cria-se um novo nodo cujo valor armazenado é ou contém  $p$ . Esse nodo é adicionado à lista de filhos de  $n$  pela chamada à função *ADD-CHILD*;
- desempilhamento de  $k$  estados: é feito pela obtenção do  $k$ -ésimo nodo ancestral de  $n$ . A função *UP* realiza esta tarefa;
- tamanho da pilha: o tamanho da pilha é obtido pela distância de  $n$  ao nodo raiz. Este valor é retornado pela função *HEIGHT*.

Todo nodo possui um número que o identifica univocamente. Este valor é obtido pela chamada a *ID*.

São utilizados seis procedimentos/funções principais: *SWEEP*, *FOLLOW-SOURCES<sub>A</sub>*, *FOLLOW-SOURCES<sub>B</sub>*, *NEXT-LOOKAHEADS<sub>A</sub>*, *NEXT-LOOKAHEADS<sub>B</sub>* e *RESOLVE-CONFLICTS*. Nesses algoritmos, é freqüente a utilização de pares para armazenamento de dados. Para auxiliar em sua manipulação, tem-se as funções *FST* e *SND*, que retornam respectivamente o primeiro e o segundo elemento de um par.

O procedimento *SWEEP* é idêntico ao procedimento *LOOKAHEAD* apresentado no Capítulo 3, exceto que *sources* armazena pares da forma  $(stack, w)$ , onde *w* é um *string* de *lookahead*. Este procedimento é mostrado na Figura 5.11.

```

SWEEP(p)
1  for a ∈ Σ
2  do if |Action[p, a] > 0 ∧ a ≠ $
3      then sources ← ∅
4      for act ∈ Action[p, a]
5      do if act é uma ação de empilhamento
6          then sources[act] ← {[p], a}
7          else ASSERT(act = redução pela produção  $A \rightarrow \alpha$ )
8              for p0 ∈ PRED(p,  $\alpha$ )
9                  do FOLLOW-SOURCESA(sources[act], [p0], A, a,  $\lambda$ )
10         RESOLVE-CONFLICTS(p, a, sources, 2)

```

Figura 5.11: Procedimento para remoção automática de conflitos pelo aumento do contexto à direita.

O procedimento *FOLLOW-SOURCES<sub>A</sub>* é uma fachada ao procedimento *FOLLOW-SOURCES<sub>B</sub>*. Esses procedimentos são mostrados respectivamente nas Figuras 5.12 e 5.13. O objetivo de *FOLLOW-SOURCES<sub>A</sub>* é transformar a pilha recebida (*stack*) em uma árvore enraizada em *root*, que corresponde à base de *stack*. Cada nodo nessa árvore é criado pela função *NODE*, que recebe o valor a ser armazenado. Os nodos da árvore criada armazenam pares da forma (*estado*, *string de lookahead*); esses valores podem ser posteriormente recuperados pela chamada a *VALUE*. A função *NODE* realiza o controle de atribuição de um identificador único a cada nodo criado.

O procedimento *FOLLOW-SOURCES<sub>B</sub>* visa obter a partir de uma pilha inicial a pilha sintática que ocasiona na leitura do terminal *a* que estende *w*. Quando *a* é encontrado, o procedimento armazena em *sources* o par  $(stack, wa)$ , onde *stack* é obtido pelos estados no caminho de *root* a *node*, ambos recebidos como parâmetro. A variável *sources* é definida na assinatura de *FOLLOW-SOURCES<sub>B</sub>*. A garantia do término de execução deste procedimento é dado pelas linhas 2–8 e 11. Nas linhas 2–8, o controle de *loop* infinito é feito armazenando-se no início de cada ativação do

procedimento a pilha de estados e a transição a ser simulada a partir dela. Toda vez que o procedimento é chamado para uma pilha e transição já utilizados em uma chamada anterior, o procedimento retorna. Ao invés de armazenar todos os estados da pilha, a utilização do identificador do nodo corrente é suficiente para identificá-la. Com isto, a cada chamada a  $FOLLOW-SOURCES_B$  cria-se um par da forma  $(id-nodo, transição)$ , acrescentando-o a um conjunto de pares já visitados. Por questões de eficiência, esse conjunto é dividido em dois:  $visited$  e  $roots$ . A utilização de  $roots$  ocorre somente no caso de chamadas cujas pilhas recebidas possuem tamanho um. Neste caso, o armazenamento do identificador do nodo é desnecessário, já que o estado de origem da transição corresponde à base/topo da pilha. Para os demais casos,  $visited$  é utilizado. No laço da linha 10, verifica-se todas as transições do estado  $q$ . Para cada símbolo  $Y$  que rotula uma transição, se  $Y \xrightarrow{*} \lambda$ , pela simulação da máquina o estado  $q$  deve ser empilhado. Para prosseguir com a simulação, faz-se uma chamada à função  $GET-FROM$  (linha 11). Esta função, dado um nodo  $node$  e um valor  $v$ , verifica inclusivamente a partir de  $node$  a existência de um nodo  $n'$  no caminho até a raiz, tal que  $n'$  armazena um valor igual a  $v$ . Se essa função retornar um valor diferente de nulo, o empilhamento de  $q$  implica em uma aresta de retorno de  $node$  a um nodo já inserido na árvore, o que caracteriza um ciclo:  $([p_1 p_2 \dots p_n q], w) \vdash^* ([p_1 p_2 \dots p_n q q_1 q_2 \dots q_n q], w)$ . O não empilhamento de  $q$  neste caso evita que o programa entre em *loop*.

```

FOLLOW-SOURCESA(sources, stack, X, a, w)
1  ASSERT(stack = [p1...pn])
2  root ← node ← NODE((p1, λ))
3  for 2 ≤ i ≤ n
4  do node2 ← NODE((pi, λ))
5     ADD-CHILD(node, node2)
6     node ← node2
7  SND(VALUE(node)) ← w
8  FOLLOW-SOURCESB(sources, (pn, X, GOTO0(pn, X)), a, w, root, node, ∅, ∅)

```

Figura 5.12: Procedimento fachada FOLLOW-SOURCES<sub>A</sub>.

Análogo a  $FOLLOW-SOURCES_A$ ,  $NEXT-LOOKAHEADS_A$  é um procedimento fachada para  $NEXT-LOOKAHEADS_B$ . Sua função é converter a pilha  $stack$  em uma árvore cujos nodos armazenam estados em  $M_0$ . No procedimento  $NEXT-LOOKAHEADS_B$  *loops* são controlados pelo uso do conjunto  $visited$ . Da mesma forma que em  $FOLLOW-SOURCES_B$ ,  $visited$  armazena pares da forma  $(id-nodo, transição)$ . Como nenhum nodo é criado durante a ativação de qualquer chamada a  $NEXT-LOOKAHEADS_B$ , a raiz da árvore é sempre a mesma. Conseqüentemente, o uso de  $roots$  é desnecessário. Os procedimentos  $NEXT-LOOKAHEADS$  são mostrados nas Figuras 5.14 e 5.15.

```

FOLLOW-SOURCESB(sources, transition, a, w, root, node, visited, roots)
1  stackSize ← HEIGHT(node, root)
2  if stackSize = 1
3    then if transition ∈ roots
4          then return
5          else roots ← roots ∪ {transition}
6    else if (ID(node), transition) ∈ visited
7          then return
8          else visited ← visited ∪ {(ID(node), transition)}
9  ASSERT(transition = (ts, X, q))
10 for Y ∈ V | GOTO0(q, Y) ≠ Ω
11 do if Y  $\xrightarrow{*}$  λ ∧ GET-FROM(node, (q, w)) = nil
12   then node2 ← NODE((q, w))
13       ADD-CHILD(node, node2)
14       FOLLOW-SOURCESB
15       (sources, (q, Y, GOTO0(q, Y)), a, w, root, node2, visited, roots)
16   else if Y = a
17       then node2 ← node
18           list ← [FST(VALUE(node2))]
19           while (node2 ← PARENT(node2)) ≠ nil
20             do list ← list + [FST(VALUE(node2))]
21
22           ASSERT(list = [pn...p1])
23           stack ← [p1...pnq]
24           sources ← sources ∪ {(stack, wa)}
25 bottom ← FST(VALUE(root))
26 for C → γ• ∈ ts | C ≠ S
27 do if |γ| + 1 < stackSize
28   then node2 ← UP(node, |γ|)
29       SND(VALUE(node2)) ← w
30       ts2 ← FST(VALUE(node2))
31       FOLLOW-SOURCESB
32       (sources, (ts2, C, GOTO0(ts2, C)), a, w, root, node2, visited, roots)
33   else ASSERT(γ = γ1γ2), onde |γ2| = stackSize - 1
34       for p0 ∈ PRED(bottom, γ1)
35         do root2 ← NODE((p0, w))
36           FOLLOW-SOURCES
37           (sources, (p0, C, GOTO0(p0, C)), a, w, root2, root2, visited, roots)

```

Figura 5.13: Procedimento FOLLOW-SOURCES<sub>B</sub>.

O procedimento *RESOLVE-CONFLICTS*, utilizado na resolução de conflitos, é exibido na Figura 5.16. O código desse procedimento é análogo ao apresentada por Charles [Charles, 1991] (vide Seção 3.4.2.1), exceto que os valores em *sources[act]* são

```

NEXT-LOOKAHEADSA(stack, X)
1  la ← ∅
2  ASSERT(stack = [p1...pn])
3  root ← node ← NODE(p1)
4  for 2 ≤ i ≤ n
5  do node2 ← NODE(pi)
6     ADD-CHILD(node, node2)
7     node ← node2
8  NEXT-LOOKAHEADSB(la, (pn, X, GOTO0(pn, X)), root, node, ∅)
9  return la

```

Figura 5.14: Função NEXT-LOOKAHEADS<sub>A</sub>.

```

NEXT-LOOKAHEADSB(la, transition, root, node, visited)
1  ASSERT(transition = (ts, X, q))
2  bottom ← FST(VALUE(root))
3  if (ID(node), transition) ∈ visited
4  then return
5  la ← la ∪ READ1(ts, X)
6  stackSize ← HEIGHT(node, root)
7  nStacks ← ∅
8  for C → γ • Xδ | δ  $\xrightarrow{*}$  λ ∧ C ≠ S
9  do if |γ| + 1 < stackSize
10     then node2 ← UP(node, |γ|)
11           nStacks ← nStacks ∪ {(node2, C)}
12
13     else ASSERT(γ = γ1γ2), onde |γ2| = stackSize - 1
14           for p0 ∈ PRED(bottom, γ1)
15             do la ← la ∪ FOLLOW1(p0, C)
16 for (n, C) ∈ nStacks
17 do ts ← FST(VALUE(n))
18     NEXT-LOOKAHEADSB(la, (ts, X, GOTO0(ts, X)), root, n, visited)

```

Figura 5.15: Procedimento NEXT-LOOKAHEADS<sub>B</sub>.

pares da forma (*stack*, *w*).

**Listagem dos conflitos:** após a execução de *RESOLVE-CONFLICTS*, o próximo passo executado por SAIDE é a identificação dos conflitos não resolvidos, listando-os de acordo com a prioridade em que devem ser eliminados. Quando  $k_{max} \geq 2$ , a identificação dos conflitos pela mera inspeção das linhas na tabela *Action* pode resultar em um número maior de entradas com conflito em relação às obtidas quando somente um *lookahead* é utilizado. Isto não é desejável, pois leva à conclusão errônea de que

```

RESOLVE-CONFLICTS( $q, t, sources, n$ )
1  if  $t = \$ \vee n > k_{max}$ 
2    then return
3  alocar uma nova linha p na tabela Action
4  for  $a \in \Sigma$ 
5  do  $Action[p, a] \leftarrow \emptyset$ 
6   $Action[q, t] \leftarrow \{la-shift\ p\}$ 
7  for act que indexa sources
8  do for  $src \in sources[act]$ 
9    do ASSERT( $src = (stack, w)$ )
10    $la \leftarrow NEXT-LOOKAHEADS_A(stack, t)$ 
11   for  $a \in la$ 
12     do  $Action[p, a] \leftarrow Action[p, a] \cup \{act\}$ 
13 for  $a \in (\Sigma - \{\$\}) \mid |Action[p, a]| > 1$ 
14 do for  $act \in Action[p, a]$ 
15   do  $nSources \leftarrow \emptyset$ 
16     for  $src \in sources[act]$ 
17     do ASSERT( $src = (stack, w)$ )
18       FOLLOW-SOURCES $_A(nSources[act], stack, t, a, w)$ 
19
20   RESOLVE-CONFLICTS( $p, a, nSources, n + 1$ )

```

Figura 5.16: Procedimento RESOLVE-CONFLICTS.

o número de conflitos aumentou após a a tentativa de remoção automática. Isto é exemplificado pela seguinte gramática:

- (1)  $A \rightarrow \mathbf{b} B$  ;
- (2)  $A \rightarrow \mathbf{a}$  ;
- (3)  $B \rightarrow \mathbf{c} C$  ;
- (4)  $B \rightarrow \mathbf{c} C D \mathbf{f}$  ;
- (5)  $C \rightarrow \mathbf{d} A$  ;
- (6)  $C \rightarrow \lambda$  ;
- (7)  $D \rightarrow \mathbf{e}$  ;
- (8)  $D \rightarrow \lambda$  ;

Ao computar a tabela *Action* correspondente, tem-se as seguintes entradas para o estado 8 do autômato  $LALRA_1$ , estado em que ocorrem todos os os conflitos:

	<b>a</b>	<b>b</b>	<b>c</b>	<b>d</b>	<b>e</b>	<b>f</b>	<b>\$</b>
8					S11, R8	R3, R8	R3
...							

Ao fazer  $k \leq k_{max} = 2$ , dois novos estados são criados: 13 e 14. As entradas  $Action[8, e]$  e  $Action[8, f]$  possuem agora ações de *lookahead-shift* para os estados 13 e 14.

	a	b	c	d	e	f	\$
8					L13	L14	R3
.							
13						S11, R3	
14					R3, R8	R3, R8	R3, R8

Nesta nova tabela existem 4 conflitos ao invés dos 2 anteriormente obtidos.

Para identificar o número correto de conflitos não removidos, varrem-se as linhas que correspondem somente aos estados em  $M_0$ , desconsiderando portanto, qualquer estado criado em função do cômputo de ADFLs. O procedimento *SET-CONFLICTS*, mostrado na Figura 5.17, é utilizado para identificação desse conjunto mínimo de conflitos. Nesse procedimento, a cada entrada  $(p, a)$  analisada, duas possibilidades indicam

*SET-CONFLICTS*()

```

1  conflicts ← ∅
2  lookaheads ← ∅
3  cftStart ← ∅
4  cftEnd ← ∅
5  for  $(p, a) \in \{p_1, p_2, \dots, p_n\} \times \Sigma \mid p_1 < p_2 < \dots < p_n \wedge p_i \in M_0$ 
6  do  $cftStart[p] \leftarrow |conflicts| + 1$ 
7     if  $|Action[p, a]| \geq 2$ 
8     then CFTS-1( $p, a$ )
9     else if  $|Action[p, a]| = \{la-shift\ q\}$ 
10     then CFTS-K( $p, a, q$ )
11   $cftEnd[p] = |conflicts|$ 

```

Figura 5.17: Procedimento *SET-CONFLICTS*.

a presença de um ou mais conflitos. A primeira ocorre quando  $Action[p, a] \geq 2$ . Na varredura das linhas correspondentes aos estados em  $M_0$ , isto só possível se  $k = 1$ . Nestas situações, utiliza-se o procedimento *CFTS-1* para identificação dos conflitos. O número de conflitos em uma entrada  $Action[p, a]$  é dado pela fórmula:

$$ncfts = (|shift| \times |reds|) + (\lambda x. \text{if } x \geq 2 \text{ then } 1 \text{ else } 0)|reds|$$

onde

$$\begin{aligned} shift &= \{s \mid s \in Action(p, a) \wedge s \text{ é uma ação de empilhamento}\} \\ reds &= \{r \mid r \in Action(p, a) \wedge r \text{ é uma ação de redução}\} \end{aligned}$$

Note que a maior cardinalidade possível para o conjunto *shift* é um.

CFTS-1(*p*, *a*)

```

1  reds ← ∅
2  shift ← nil
3  for act ∈ Action[p, a]
4  do if act é uma ação de empilhamento
5      then shift ← act
6      else reds ← reds ∪ {act}
7  if shift ≠ nil
8      then shiftItems ← ∅
9          for B → β • aγ ∈ p
10         do shiftItems ← shiftItems ∪ {B → β • aγ}
11         for red ∈ reds
12         do ASSERT(red = redução pela produção A → α)
13             conflicts ← conflicts + [(p, shiftItems ∪ {A → α•}, a)]
14             lookaheads ← lookaheads + [{a}]
15  if |reds| ≥ 2
16      then redItems ← ∅
17          for red ∈ reds
18          do ASSERT(red = redução pela produção A → α)
19              redItems ← redItems ∪ {A → α•}
20              conflicts ← conflicts + [(p, redItems, a)]
21              lookaheads ← lookaheads + [{a}]

```

Figura 5.18: Procedimento CFTS-1.

A segunda possibilidade acontece quando utiliza-se mais de um *lookahead*. Assim, se uma entrada *lookahead-shift* *q* for encontrada, *q* é um estado de um AFDL  $M_L$ . Os conflitos, se existirem, estarão contidos em entradas em *Action* que correspondem a estados sem transições em  $M_L$  – estados folha. Conforme estabelecido na linha 10 do procedimento *RESOLVE-CONFLICTS*, todo e qualquer conflito existente em um estado folha consiste em um subconjunto do conjunto original de ações sintáticas na tabela LALR(1), o que permite recuperar o conjunto original de ações. Isto é feito no procedimento *CFTS-K*, mostrado na Figura 5.19. Este procedimento realiza, para



todo AFDL encontrado, uma busca em profundidade de forma a encontrar as entradas com cardinalidade maior que um.

Os procedimentos *CFTS-1* e *CFTS-K*, toda vez que acrescentam um novo conflito à lista *conflicts*, também adicionam um conjunto de *strings* de *lookaheads* à lista *lookaheads*. Com isto, dado um conflito na *i*-ésima posição em *conflicts*, os *lookaheads* para os quais o conflito permanece são obtidos em *lookaheads*[*i*].

CFTS-K(*p*, *a*, *q*)

```

1  lookaheadsSR ← ∅
2  lookaheadsRR ← ∅
3  TRAVERSE(s, lookaheadsSR, lookaheadsRR, a)
4  if |lookaheadsSR| > 0
5      then shiftItems ← ∅
6          for  $B \rightarrow \beta \bullet a\gamma \in p$ 
7              do shiftItems ← shiftItems ∪ { $B \rightarrow \beta \bullet a\gamma$ }
8              for (shift, reduce) que indexa lookaheadsSR
9                  do ASSERT(reduce = redução pela produção  $A \rightarrow \alpha$ )
10                     conflicts ← conflicts + [(p, shiftItems ∪ { $A \rightarrow \alpha \bullet$ }, a)]
11                     lookaheads ← lookaheads + [lookaheadsSR[(shift, reduce)]]
12 if |lookaheadsRR| ≥ 2
13     then redItems ← ∅
14         for  $A \rightarrow \alpha \bullet \in p \mid a \in LA_1(p, A \rightarrow \alpha)$ 
15             do redItems ← redItems ∪ { $A \rightarrow \alpha \bullet$ }
16             conflicts ← conflicts + [(p, redItems, a)]
17             lookaheads ← lookaheads + [lookaheadsRR]
```

Figura 5.19: Procedimento CFTS-K.

O intervalo de índices que contém os conflitos de um dado estado são salvos em dois vetores globais, indexados pelos números dos estados em *LALRA*<sub>1</sub>: *cftStart* e *cftEnd*. Isto é feito nas linhas 6 e 11 do procedimento SET-CONFLICTS.

Para os conflitos que não puderam ser eliminados automaticamente pelo aumento de *k*, realizam-se as quatro fases da metodologia para remoção manual: entendimento, classificação, alteração da especificação e testes, aplicadas a cada conflito listado segundo a prioridade em que devem ser removidos.

A priorização dos conflitos é feita pela construção de um grafo dirigido denominado *grafo de conflito* – *G<sub>c</sub>*. Nesse grafo, os vértices são estados do autômato *LALRA*<sub>1</sub> que possuem pelo menos um conflito. Esses estados são identificados a partir da inspeção da lista *conflicts*. Uma aresta em *G<sub>c</sub>* conecta dois vértices *p* e *q* se existe um caminho de *p* a *q* em *LALRA*<sub>1</sub>. De *G<sub>c</sub>*, constrói-se um novo grafo dirigido acíclico *G<sub>CFC</sub>* de modo que um vértice *σ* neste grafo representa um conjunto de vértices *p*<sub>1</sub>, *p*<sub>2</sub>, ..., *p*<sub>*n*</sub> que formam um componente fortemente conectado em *G<sub>c</sub>*. Existe uma aresta entre *σ<sub>a</sub>* e *σ<sub>b</sub>*

```

TRAVERSE( $p, lookaheadsSR, lookaheadsRR, w$ )
1  for  $a \in \Sigma$ 
2  do if  $Action[p, a] = \{la-shift\ q\}$ 
3      then TRAVERSE( $q, lookaheadsSR, lookaheadsRR, wa$ )
4      else if  $|Action[p, a]| > 1$ 
5          then  $shift \leftarrow nil$ 
6               $reds \leftarrow \emptyset$ 
7              for  $act \in Action[p, a]$ 
8                  do if  $act$  é uma ação de empilhamento
9                      then  $shift \leftarrow act$ 
10                     else  $reds \leftarrow reds \cup \{act\}$ 
11             if  $shift \neq nil$ 
12                 then for  $red \in reds$ 
13                     do  $lookaheadsSR[(shift, red)] \leftarrow$ 
14                          $lookaheadsSR[(shift, red)] \cup \{wa\}$ 
15             if  $|reds| \geq 2$ 
16                 then  $lookaheadsRR \leftarrow lookaheadsRR \cup \{wa\}$ 

```

Figura 5.20: Procedimento TRAVERSE.

em  $G_{CFC}$  se e somente se  $G_c$  contém uma aresta de  $p_i$  a  $p_j$  e  $p_i$  e  $p_j$  são dois vértices quaisquer respectivamente no conjunto de vértices agregados a  $\sigma_a$  e  $\sigma_b$ . Pelo raciocínio de construção dos estados em  $LALRA_1$  a partir de  $LRA_1$ , explicado na Seção 3.3, uma aresta  $(p_i, p_j)$  em  $G_c$  estabelece que  $p_j$  possui itens que recebem *lookaheads* de  $p_i$ .

O grafo  $G_{CFC}$  é então ordenado topologicamente, o que resulta em uma seqüência de vértices. Para cada vértice  $\sigma_a$  dessa seqüência, que é percorrida da esquerda para a direita, listam-se os conflitos nos estados  $p_1, p_2, \dots, p_n$  em  $LALRA_1$  que constituem o CFC em  $G_c$  correspondente. A listagem dos conflitos em  $p_i$  é feita percorrendo-se a lista *conflicts* no intervalo  $[cftStart[p_i] \dots cftEnd[p_i]]$ .

A listagem dos conflitos segundo a seqüência obtida é uma heurística para guiar a ordem em que os conflitos devem ser resolvidos. Espera-se, com isto, que conflitos decorrentes da existência de outros possam ser automaticamente eliminados quando estes também o forem.

Seguindo a ordem em que os conflitos são reportados, o usuário aplica as quatro fases da metodologia. A implementação das fases de entendimento e classificação são discutidas a seguir.

**Depuração de conflitos:** DeRemer e Pennello [DeRemer e Pennello, 1982], como produto da pesquisa do método de cálculo de *lookaheads* para  $k = 1$ , obtiveram um conjunto de equações a partir das quais obtêm-se as árvores de derivação para explanação de um conflito.

Para um conflito *shift/reduce*, é criada uma árvore para cada item de empilhamento e uma única árvore para o item de redução. Analogamente, uma árvore é construída para cada ação de redução em um conflito *reduce/reduce*.

O formato geral de uma árvore de derivação para um item de redução é exibido na Figura 5.21. Observe que cada nível corresponde a um passo de derivação e o lado direito de uma produção é posicionado logo abaixo do não terminal que o produz. O item de redução é identificado pela derivação  $A_{s-1} \Rightarrow \alpha_s$ .

A parte (a) representa a derivação do símbolo de partida  $S'$  a  $\alpha B\beta_1$  que contribui com o símbolo  $t$  de conflito, isto é,  $t \in FIRST_1(\beta_1)$ ; (b) mostra como  $t$  é derivado a partir de  $\beta_1$  e finalmente (c) é o caminho da derivação  $B \xRightarrow{*} \alpha_1\alpha_2\dots\alpha_{s-1}A_{s-1}\gamma_{s-1}\dots\gamma_2\gamma_1 \Rightarrow \alpha_1\alpha_2\dots\alpha_{s-1}\alpha_s\gamma_{s-1}\dots\gamma_2\gamma_1$ , onde  $A_{s-1} \rightarrow \alpha_s\bullet$  é o item de redução envolvido no conflito.

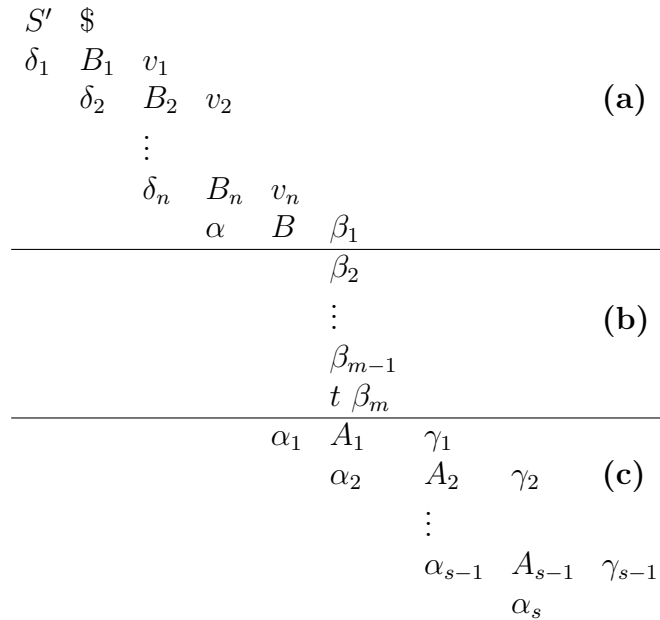


Figura 5.21: Formato de uma árvore de derivação.

A construção de uma árvore de derivação de um item de redução em um estado inconsistente  $q$  é feita em ordem inversa de suas partes:

- c) dado o item  $A_{s-1} \rightarrow \alpha_s\bullet$  em  $q$ , inicia-se uma busca a partir das transições  $(q', A_{s-1})$ , tal que  $q' \in PRED(q, \alpha_s)$ . As transições  $(q', A_{s-1})$  consistem em nodos em  $G_i$ , o grafo da relação *includes*. Em seguida, percorre-se algumas arestas em  $G_i$  até que se alcance um nodo referente a uma transição  $(p, B)$ , tal que  $t \in READ_1(p, B)$ . Todo item  $B_n \rightarrow \alpha \bullet B\beta_1$  em  $p$ , tal que  $t \in FIRST_1(\beta_1)$ , é considerado um *item de contribuição*. Um conjunto de árvores de derivação deve ser construído para cada um desses itens. Uma produção que induz uma

aresta de inclusão de  $(p, A)$  para  $(p', B)$  é redescoberta seguindo-se as transições do estado  $p'$  sob os lados direitos das produções de  $B$ .

- b) o cálculo deste componente obtém a subárvore correspondente à derivação  $\beta_1 \xrightarrow{*} t\beta_m$ . Para isto, faz-se o cálculo

$$\begin{aligned} E = & \{B_n \rightarrow \alpha B \bullet \beta_1\} \\ & \cup \{A \rightarrow \delta X \bullet \eta \mid A \rightarrow \delta \bullet X\eta \in E \wedge X \xrightarrow{*} \lambda\} \\ & \cup \{C \rightarrow \bullet \alpha \mid A \rightarrow \delta \bullet C\eta \in E \wedge C \rightarrow \alpha \in P\} \end{aligned} \quad (5.1)$$

até que  $t$  apareça como primeiro *token*. Cada adição a  $E$  deve ser mapeada no elemento que a gerou. Seguindo o encadeamento produzido por este mapeamento tem-se a seqüência inversa dos passos de derivação  $\alpha B \beta_1 \xrightarrow{*} \alpha B t \beta_m$ .

- a) o último componente da árvore é a derivação de  $S'$  ao item de contribuição. Para isto, calcula-se o menor caminho do estado inicial ao estado de contribuição no qual o item  $B_n \rightarrow \alpha \bullet B \beta_1$  está contido. A forma sentencial  $\xi$  é a seqüência dos símbolos de transição neste caminho. O cômputo de  $E'$  é dado por

$$\begin{aligned} E' = & \{(S' \rightarrow \bullet S \$, 1)\} \\ & \cup \{(C \rightarrow \bullet \alpha, j) \mid (A \rightarrow \delta \bullet C\eta, j) \in E' \wedge C \rightarrow \alpha \in P\} \\ & \cup \{(A \rightarrow \delta X \bullet \eta, j+1) \mid (A \rightarrow \delta \bullet X\eta, j) \in E' \wedge X = \xi_j \wedge j \leq |\xi|\} \end{aligned} \quad (5.2)$$

$E'$  é calculado em uma busca em amplitude mapeando-se as adições realizadas aos pares que as geraram. No momento em que o par  $(B_n \rightarrow \alpha \bullet B \beta_1, |\xi|)$  aparece em  $E'$ , o cômputo pára. A derivação consiste do encadeamento formado pelo mapeamento das adições em  $E'$ .

O suporte à fase de entendimento é obtido pela definição de funções que implementam as equações apresentadas.

O primeiro passo no suporte a esta fase é a geração das árvores de derivação para cada item de redução envolvido no conflito. Os algoritmos para cômputo das partes (a), (b) e (c) das árvores de derivação são executados partindo-se do pressuposto de que dada uma gramática  $G$ , tem-se a disposição o autômato  $LALRA_1 = (M_0, V, P, IS, GOTO_0, RED_1)$  e o grafo da relação *includes*, dado por  $G_i = (V, E)$ , correspondentes. Assume-se ainda a existência das funções  $READ_1$  e  $FIRST_1$ . A seguir, esses algoritmos são apresentados.

**Cálculo de (c):** é realizado pela função *CPS* (*C Paths*), cujo algoritmo é exibido na Figura 5.22. Esta função realiza uma busca em amplitude de forma a obter as menores

```

CPS( $ri, t, q$ )
1  ASSERT( $ri = A_{s-1} \rightarrow \alpha_s \bullet$ )
2   $queue \leftarrow \emptyset$ 
3   $child\_pair \leftarrow ((ri, nil), q)$ 
4  for  $p \in \text{PRED}(\alpha_s, q)$ 
5  do for  $A_{s-2} \rightarrow \alpha_{s-1} \bullet A_{s-1}\gamma_{s-1} \in p$ 
6      do  $queue \leftarrow queue +$ 
7           $[((A_{s-2} \rightarrow \alpha_{s-1} \bullet A_{s-1}\gamma_{s-1}, child\_pair), p)]$ 
8
9   $v \leftarrow \emptyset$ 
10  $csf \leftarrow false$ 
11 while  $\neg csf$ 
12 do  $child\_pair \leftarrow \text{DEQUEUE}(queue)$ 
13   ASSERT( $child\_pair = ((B_n \rightarrow \alpha \bullet B\beta_1, pair), p)$ )
14   if  $t \in \text{READ}_1(p, B)$ 
15       then  $csf \leftarrow true$ 
16       else if  $\beta_1 \stackrel{*}{\Rightarrow} \lambda$ 
17           then for  $(p', B_n) \in V(G_i) \mid ((p, B), (p', B_n)) \in E(G_i)$ 
18               do for  $C \rightarrow \eta_1 \bullet B_n\eta_2 \in p' \mid (C \rightarrow \eta_1 \bullet B_n\eta_2, p') \notin v$ 
19                   do  $queue \leftarrow queue + [((C \rightarrow \eta_1 \bullet B_n\eta_2, child\_pair), p')]$ 
20                        $v \leftarrow v \cup (C \rightarrow \eta_1 \bullet B_n\eta_2, p')$ 
21
22  $cps \leftarrow \emptyset$ 
23 for  $B_n \rightarrow \alpha \bullet B\beta_1 \in p \mid t \in \text{FIRST}_1(\beta_1)$ 
24 do  $cps \leftarrow cps \cup ((B_n \rightarrow \alpha \bullet B\beta_1, child\_pair), p)$ 
25 return  $cps$ 

```

Figura 5.22: Função CPS.

derivações  $B_n \stackrel{*}{\Rightarrow} \alpha B\beta_1 \stackrel{*}{\Rightarrow} \alpha\alpha_1\dots\alpha_{s-1}A_{s-1}\gamma_{s-1}\dots\gamma_1 \Rightarrow \alpha\alpha_1\dots\alpha_{s-1}\alpha_s\gamma_{s-1}\dots\gamma_1$  a partir dos itens  $B_n \rightarrow \alpha \bullet B\beta_1$  em um estado  $p$ , tal que  $t$ , o símbolo de conflito, pertence a  $\text{FIRST}_1(\beta_1)$ .

As derivações são dadas por pares pais ligados a pares filhos, onde um par é da forma  $((i, child), p)$ . O componente  $i$  armazena o item que aparece no estado  $p$  ao qual o par se refere. O componente  $child$  armazena a referência ao par filho. A Figura 5.23 esquematiza essa ligação.

CPS define três parâmetros formais:  $ri$  – o item de redução,  $t$  – o símbolo de conflito e  $q$  – o estado que contém  $ri$ .

O primeiro passo da função é a criação do par folha  $child\_pair$  dado por  $((ri, nil), q)$ . Como  $ri$  é um item de redução, seja  $A_{s-1} \rightarrow \alpha_s \bullet$  sua forma. Para todo item  $i$  em um estado  $p$  predecessor a  $q$  sobre  $\alpha_s$ , tal que o símbolo gramatical sob o marcador de posição em  $i$  é igual a  $A_{s-1}$ , acrescenta-se à fila  $queue$  o par  $((i, child\_pair), p)$ . Ao

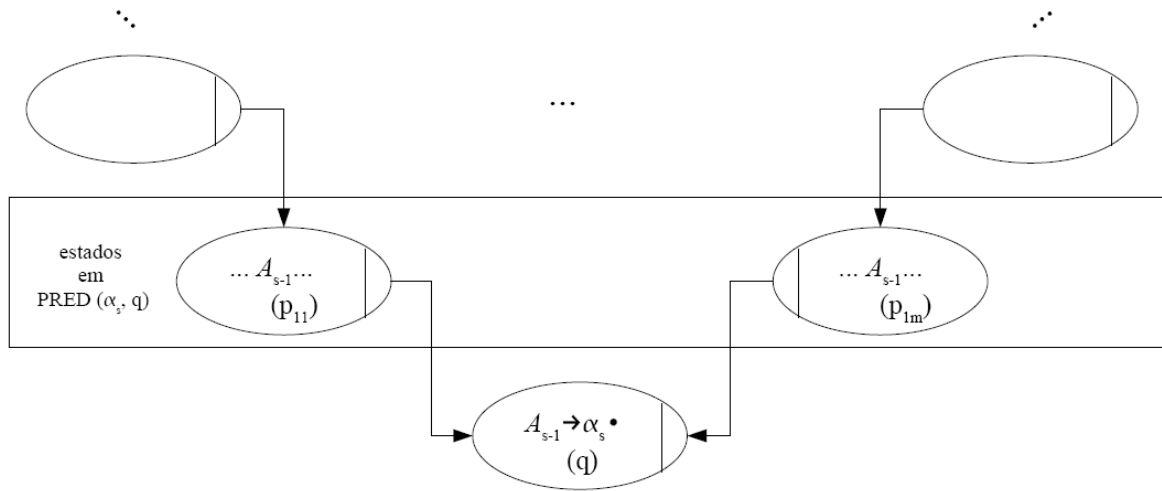


Figura 5.23: Esquema de ligação dos pares na função CPS.

final desse laço, tem-se todos os pares pai do par *child\_pair*.

No próximo comando, na linha 9, o conjunto vazio  $v$  é declarado. Esse conjunto armazena pares da forma  $(i, p)$  que indicam que o item  $i$  no estado  $p$  já foi visitado em algum momento na busca em amplitude realizada pelo laço nas linha 11 a 20. A consulta a esse conjunto permite descartar derivações cíclicas e portanto podar a árvore de busca. A viabilidade do algoritmo é garantida por essa consulta; embora do ponto de vista teórico a busca em amplitude eventualmente encontra a derivação desejada, na prática isto nem sempre ocorre, já que a árvore de possibilidades pode se expandir de tal forma a consumir toda a memória principal de um computador. Originalmente, este detalhe não foi discutido por DeRemer e Pennello.

A busca inicia pelo desenfileamento do primeiro par em *queue*, atribuindo-o a *child\_pair*. Esse par é da forma  $((B_n \rightarrow \alpha \bullet B\beta_1, pair), p)$ . Se  $READ_1(p, B)$  contiver  $t$ , a variável *csf* (**C**onflict **S**tate **F**ound) recebe o valor *true* e a busca é interrompida. Do contrário, se  $\beta_1 \xrightarrow{*} \lambda$ , para todos os itens  $C \rightarrow \eta_1 \bullet B_n \eta_2$  no estado  $p'$  não visitados, isto é, não contidos em  $v$ , tal que existe uma aresta de  $(p, B)$  para  $(p', B_n)$  em  $G_i$ , faz-se  $C \rightarrow \eta_1 \bullet B_n \eta_2$  pai de *child\_pair*. Este novo par é então adicionado a *queue*.

O último passo no cálculo de (c) é ligar todos os itens  $B_n \rightarrow \alpha \bullet B\beta_1$  em  $p$ , tal que  $FIRST_1(\beta_1)$  contém  $t$ , à subárvore dada por *child\_pair*. Para cada item  $i$  em  $p$  que satisfaça essa condição, adiciona-se  $((i, child\_pair), p)$  à lista *cps*.

Dado um item de contribuição em  $p$ , o seu caminho de derivação em *cps* é sempre o menor possível. Esta propriedade é garantida pela busca em amplitude.

**Cálculo de (b):** é computado pela função *BP* (**B** **P**ath), cujo algoritmo é apresentado na Figura 5.24. A função *BP* calcula o caminho de derivação de  $\beta_1 \xrightarrow{*} t\beta_1$ ,

dado o item de contribuição  $B_n \rightarrow \alpha \bullet B\beta$ . Essa derivação é obtida por uma busca em amplitude de forma a garantir que seja a menor possível. Durante a busca, os pares são ligados dos filhos para os pais, conforme apresentado na Figura 5.25. São passados dois argumentos à função: o item que contribui para o conflito –  $ci$  e  $t$ , o símbolo de conflito. A execução começa pelo avançado do marcador de posição do item recebido, o que resulta em  $ci'$ , inserido em *queue*.

No próximo passo entra-se em um laço que é executado até o ponto em que *csf* (*Conflict Symbol Found*) se torna verdadeiro, o que indica que  $t$  apareceu como primeiro símbolo na forma sentencial obtida pela derivação em *pair*. Do contrário, se o item corrente é da forma  $A \rightarrow \delta \bullet X\eta$ , com  $X \in N$ , bifurca-se a busca de modo a realizar duas verificações:

- i) se  $X$  é anulável, pode ocorrer que  $\eta \xrightarrow{*} t\beta_m$ . A inserção do par  $(A \rightarrow \delta X \bullet \eta, \textit{parent\_pair})$  em *queue* expande o espaço de busca de forma a cobrir esta possibilidade;
- ii) como  $X$  é não terminal, os *tokens* gerados por suas produções também podem gerar formas sentenciais  $t\beta_m$ . A inserção de  $(X \rightarrow \bullet \alpha, \textit{pair})$  em *queue* permite essa averiguação.

A execução da função termina pela inversão da ligação dos pares, de forma a permitir um percorrimento *top down* a partir de *pair*, da mesma forma que um par em *CPS*.

**Cálculo do componente (a):** é computado pela função *AP* (*A Path*), cujo algoritmo é:

```

AP( $ci, p, \xi$ )
1  return PSI( $\xi, ci, p$ )

```

Os parâmetros formais de *AP* são: o item de contribuição –  $ci$ , o estado  $p$  que contém  $ci$  e a forma sentencial  $\xi$ , referente à concatenação dos símbolos de entrada dos estados no menor caminho de *IS* a  $p$ .

A função *AP* é equivalente à função *PSI*( $\xi, ci, p$ ). Esta função (*Path from the Start Item*), fornece o par referente à derivação  $S\$ \xrightarrow{*} \delta_1\delta_2\dots\delta_n\alpha B\beta_1v_n\dots v_2v_1\$$

Antes de apresentar o algoritmo para cômputo de *PSI*, considere a função  $XI_1$  que calcula a forma  $\xi$  mencionada, cujo algoritmo é mostrado na Figura 5.26. A execução

```

BP( $ci, t$ )
1  ASSERT( $ci = B_n \rightarrow \alpha \bullet B\beta_1$ )
2   $ci' \leftarrow (B_n \rightarrow \alpha B \bullet \beta_1)$ 
3   $queue \leftarrow (ci', nil)$ 
4   $csf \leftarrow false$ 
5  while  $\neg csf$ 
6  do  $pair \leftarrow DEQUEUE(queue)$ 
7     ASSERT( $pair = (item, parent\_pair)$ )
8     if  $item$  é da forma  $A \rightarrow \delta \bullet X\eta$ 
9     then if  $X = t$ 
10    then  $csf \leftarrow true$ 
11    else if  $X \in N$ 
12    then if  $X \stackrel{*}{\Rightarrow} \lambda$ 
13    then  $queue \leftarrow queue + [(A \rightarrow \delta X \bullet \eta, parent\_pair)]$ 
14    for  $X \rightarrow \alpha \in P$ 
15    do  $queue \leftarrow queue + [(X \rightarrow \bullet \alpha, pair)]$ 
16  REVERSE( $pair$ )
17  return  $pair$ 

```

Figura 5.24: Função BP.

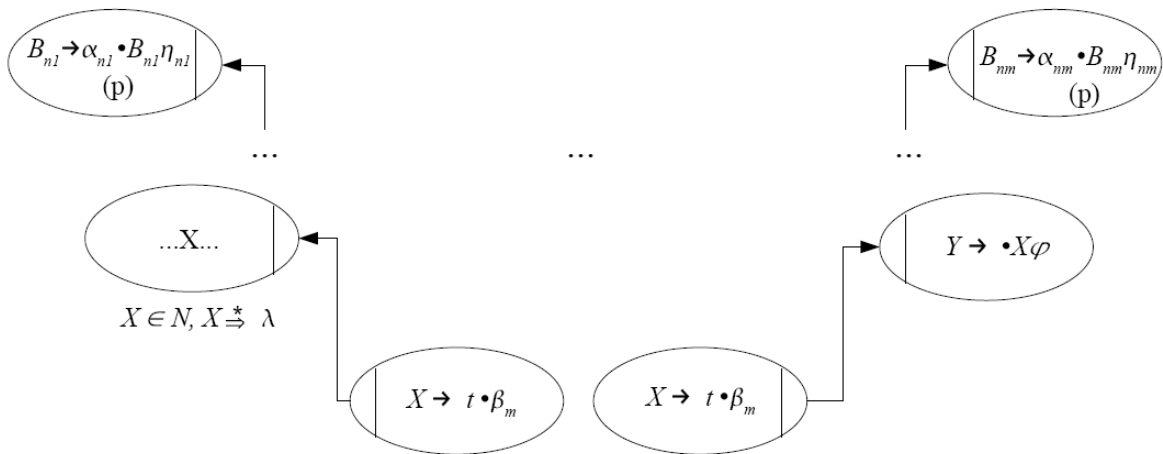


Figura 5.25: Esquema de ligação dos pares em BP.

dessa função começa pela atribuição de  $p$  à variável  $p'$  e a inicialização de uma lista responsável por armazenar o símbolo de entrada de cada estado. Em seguida, executa-se o laço externo de forma que a cada passo da iteração o símbolo de entrada do estado predecessor mais próximo ao estado inicial em  $LALRA_1$  é acrescentado ao início da lista. O predecessor  $p$  de  $p'$  mais próximo a  $IS$  é o estado que possui o menor número. Isto é garantido pelo fato de SAIDE calcular os estados em  $M_0$  aos moldes de uma busca em amplitude. A variável  $list$ , que contém os símbolos referentes a  $\xi$ , é então



retornada.

```

XI1(p)
1  p' ← p
2  list ← ∅
3  while p' ≠ IS
4  do X ← ENTRY-SYMBOL(p')
5     list ← [X] + list
6     lowest_state ← p'
7     for p ∈ M0 | GOTO0(p, X) = p'
8     do if p < lowest_state
9         then lowest_state ← p
10    p' ← lowest_state
11  return list

```

Figura 5.26: Função XI<sub>1</sub>.

A função *PSI* é definida conforme o algoritmo na Figura 5.27. Esta função retorna a derivação obtida a partir do item inicial  $S' \rightarrow \bullet S\$$  ao item *leaf\_item* recebido como parâmetro e contido no estado  $p$ . Na obtenção da derivação, a função se guia pelos símbolos de transição contidos em  $\xi$ , primeiro argumento recebido. A execução de *PSI* inicia pela adição do par  $((S' \rightarrow \bullet S\$, 1), nil)$  à lista  $E'$ . Em seguida declara o par *leaf\_pair* utilizado como critério de parada do laço nas linhas 3 – 14. Note que

```

PSI( $\xi$ , leaf_item, p)
1  E'[1] ← ((S' → •S$, 1), nil)
2  leaf_pair ← (leaf_item, | $\xi$ |)
3  for i ← 1 to |E'|
4  do pair ← E'[i]
5     ASSERT((item, j) = FST(pair))
6     if (item, j) ≠ leaf_pair
7         then if item é da forma A → δ • Xη
8             then if j ≤ | $\xi$ | ∧ X = ξi
9                 then if ((A → δX • η, j + 1), SND(pair)) ∉ E'
10                    then E'[|E'|] ← ((A → δX • η, j + 1), SND(pair))
11            if X ∈ N
12                then for X → α ∈ P ∧ ((X → •α, j), pair) ∉ E'
13                    do E' ← E' + [((X → •α, j), pair)]
14        else break
15  REVERSE(pair)
16  return pair

```

Figura 5.27: Função PSI.

$E'$  é uma lista indexada de modo a permitir uma busca em amplitude em que os pares são ligados dos filhos para os pais. A busca começa na linha 4, atribuindo-se a  $pair$  o primeiro elemento não visitado em  $E'$ . Se o primeiro componente em  $pair$  for diferente de  $leaf\_pair$ , continua-se a execução do laço. Neste ponto, a busca divide-se em dois caminhos:

1. o item armazenado em  $FST(FST(pair))$  é da forma  $A \rightarrow \delta \bullet X \eta$ . Se  $j$  for menor ou igual a  $|\xi|$ , pode-se garantidamente verificar se  $X$  é igual a  $\xi_j$ , o  $j$ -ésimo símbolo gramatical em  $\xi$ . Se esta igualdade for verdadeira, o marcador de posição avança uma posição e o par  $((A \rightarrow \delta X \bullet \eta, j + 1), SND(pair))$ , onde  $SND(pair)$  contém a referência ao pai de  $pair$ , é adicionado a  $E'$ , se este não o contiver;
2.  $X$  é um símbolo não terminal. Para todas as produções cujo lado esquerdo é igual  $X$ , adicionam-se a  $E'$  os pares  $((X \rightarrow \bullet \alpha, j), pair)$ , desde que não tenham sido adicionados em um momento anterior.

No momento em que a verificação na linha 6 falha, a derivação do item inicial a  $leaf\_item$  é encontrada. O último passo da função é a inversão da seqüência de ligações, de modo que se possa realizar um percorrimto *top down* a partir de  $pair$ .

O segundo passo para dar suporte à fase de entendimento consiste na construção de um conjunto de árvores para cada item envolvido no conflito.

Inicialmente, considere o caso de um conflito *reduce/reduce* dado pela tupla  $(r_1, r_2, \dots, r_n)$ , onde  $r_i$  é um item de redução. Quando este conflito ocorre, é necessário construir uma árvore para cada item envolvido no conflito. Essas árvores são computadas pela função *RR-DEBUG-TRACES*, cujo algoritmo é exibido na Figura 5.28.

Inicialmente, *RR-DEBUG-TRACES* escolhe um item  $r_i$  dentre os itens  $r_1, r_2, \dots, r_n$  referentes ao conflito *reduce/reduce*. No caso, o algoritmo escolhe sempre o primeiro item. Em seguida, realiza a chamada a *RTRACES* (Figura 5.29), de modo a obter  $m$  possíveis árvores de derivação de  $r_1$ ; uma para cada item de contribuição encontrado. Armazena-se na primeira posição do vetor  $rs$  a árvore  $rt_1$  do item de redução  $r_1$ . Este vetor serve ao propósito de reportar conflitos específicos a LALR, conforme será mostrado posteriormente.

O dicionário *lcs* (**L**eft **C**ontexts), indexado por formas sentenciais, armazena as árvores que possuem um dado contexto à esquerda  $\xi$ , juntamente com o item de redução a partir das quais foram calculadas. O conjunto  $\xi_s$ , por sua vez, armazena todos os contextos à esquerda processados nesta função. Dada uma árvore de derivação, o contexto à esquerda é obtido pela função  $XI_2$ . Para o formato de árvore mostrado na Figura 5.21, o contexto à esquerda retornado por  $XI_2$  é igual a  $\xi = \delta_1 \delta_2 \dots \delta_n \alpha \alpha_1 \alpha_2 \dots \alpha_s$ . O laço nas linhas 6 – 9 em *RR-DEBUG-TRACES* realiza essas duas tarefas para as

```

RR-DEBUG-TRACES(cft, t, q)
1  ASSERT( $(r_1, r_2, \dots, r_n) = cft$ )
2   $rs[1..n] \leftarrow nil$ 
3   $(rt_1, rt_2, \dots, rt_m) \leftarrow RTRACES(r_1, t, q)$ 
4   $rs[1] \leftarrow rt_1$ 
5   $lcs \leftarrow \xi_s \leftarrow \emptyset$ 
6  for  $rt_i \in (rt_1, rt_2, \dots, rt_m)$ 
7  do  $\xi \leftarrow XI_2(rt_i)$ 
8      $lcs[\xi] \leftarrow lcs[\xi] \cup \{(rt_i, r_1)\}$ 
9      $\xi_s \leftarrow \xi_s \cup \{\xi\}$ 
10
11 for  $r_i \in (r_2, \dots, r_n)$ 
12 do  $(rt_1, rt_2, \dots, rt_m) \leftarrow RTRACES(r_i, t, q)$ 
13     $rs[i] \leftarrow rt_1$ 
14    for  $rt_i \in (rt_1, rt_2, \dots, rt_m)$ 
15    do  $\xi \leftarrow XI_2(rt_i)$ 
16        $lcs[\xi] \leftarrow lcs[\xi] \cup \{(rt_i, r)\}$ 
17        $\xi_s \leftarrow \xi_s \cup \{\xi\}$ 
18  $traces \leftarrow \emptyset$ 
19  $lr\_conflict \leftarrow false$ 
20 for  $\xi \in \xi_s$ 
21 do  $trace \leftarrow rr' \leftarrow \emptyset$ 
22    for  $(rt, r) \in lcs[\xi] \mid r \notin rr'$ 
23    do  $trace \leftarrow trace \cup \{rt\}$ 
24        $rr' \leftarrow rr' \cup \{r\}$ 
25    if  $|rr'| = \text{número de itens em } cft$ 
26    then  $lr\_conflict \leftarrow true$ 
27            $traces \leftarrow traces \cup \{(trace, false)\}$ 
28 if  $\neg lr\_conflict$ 
29    then  $traces \leftarrow traces \cup \{(\{rs[1] \dots rs[n]\}, true)\}$ 
30 return  $traces$ 

```

Figura 5.28: Função para cálculo das árvores de redução de um conflito *reduce/reduce*.

árvores  $rt_1, rt_2, \dots, rt_m$  referentes a  $r_1$ . Nas linhas 11 – 17, faz-se o mesmo para cada árvore retornada por *RTRACES*, dado o item  $r_i \in (r_2, \dots, r_n)$ .

No último laço da função, nas linhas 20 – 27, para cada contexto à esquerda  $\xi$  em  $\xi_s$ , se cada árvore em  $lcs[\xi]$  corresponder a um item de redução, tem-se um conjunto de árvores para explanação do conflito. Como todas possuem o mesmo contexto à esquerda, o conflito em questão não é específico a LALR. A variável *lr\_conflict* recebe o valor *true* para indicar isto. Ao final do laço, se *lr\_conflict* for falso, para todos os contextos à esquerda analisados, não foi encontrado nenhum tal que o conjunto de árvores em  $lcs[\xi]$  cobrisse todos os itens de redução. Portanto, o conflito é necessari-

```

RTRACES( $ri, t, q$ )
1   $cps \leftarrow CPS(ri, t, q)$ 
2   $traces \leftarrow \emptyset$ 
3  for  $cp \in cps$ 
4  do  $ci \leftarrow FST(FST(cp))$ 
5       $p \leftarrow SND(cp)$ 
6       $bp \leftarrow BP(ci, t)$ 
7       $ap \leftarrow AP(ci, p, XI_1(p))$ 
8       $traces \leftarrow traces \cup \{(ap, bp, cp)\}$ 
9  return  $traces$ 

```

Figura 5.29: Algoritmo para cômputo das árvores de derivação dado um item de redução. Para item de contribuição, uma árvore é construída.

```

 $XI_2(trace)$ 
1  ASSERT $((ap, bp, cp) = trace)$ 
2   $list \leftarrow \emptyset$ 
3   $pair \leftarrow ap$ 
4  while  $pair \neq nil$ 
5  do ASSERT $(pair = ((item, j), child\_pair))$ , onde  $item$  é da forma  $B_1 \rightarrow \delta_2 \bullet B_2\theta_2$ 
6       $list \leftarrow list + [\delta_2]$ 
7       $pair \leftarrow child\_pair$ 
8   $pair \leftarrow SND(FST(cp))$ 
9  while  $pair \neq nil$ 
10 do ASSERT $(pair = ((item, child\_pair), p))$ , onde  $item$  é da forma  $A_1 \rightarrow \alpha_2 \bullet A_2\gamma_2$ 
11      $list \leftarrow list + [\alpha_2]$ 
12      $pair \leftarrow child\_pair$ 
13 return  $list$ 

```

Figura 5.30: Função para cômputo do contexto à esquerda de uma árvore de derivação.

amente específico a LALR. Neste caso, retorna-se o conjunto de árvores de derivação em  $rs$ .

Para conflitos *shift/reduce*, utiliza-se a função *SH-DEBUG-TRACES*. Esta função, apresentada na Figura 5.31, constrói as árvores de derivação para os itens de empilhamento pela invocação de *PSI*, que recebe o contexto à esquerda  $\xi$  de cada árvore referente a um item de redução.

Por último, a função *DEBUG-CONFLICT* é a fachada de todas as construções apresentadas. O seu pseudocódigo é definido na Figura 5.32.

```

SH-DEBUG-TRACES(cft, t, q)
1  ASSERT( $(s_1, s_2, \dots, s_n, r) = cft$ )
2  traces  $\leftarrow \emptyset$ 
3  for rtrace  $\in$  RTRACES(r, t, q)
4  do  $\xi \leftarrow \text{XI}_2(\textit{rtrace})$ 
5     strace[1..n]  $\leftarrow nil$ 
6     for  $s_i \in (s_1, s_2, \dots, s_n)$ 
7     do strace[i]  $\leftarrow \text{PSI}(\xi, s_i, q)$ 
8     traces  $\leftarrow traces \cup \{(strace[1], \dots, strace[n], rtrace)\}$ 
9  return traces

```

Figura 5.31: Função SH-DEBUG-TRACES.

```

DEBUG-CONFLICT(cft, t, q)
1  if cft é um conflito shift/reduce
2     then for trace  $\in$  SH-DEBUG-TRACE(cft, t, q)
3         do PRINT-SHIFT-TRACE(trace)
4     else for (trace, lr_conflict)  $\in$  RR-DEBUG-TRACES
5         do if  $\neg lr\_conflict$ 
6             then anunciar conflito LALR
7             else anunciar conflito LR/LALR
8         PRINT-RR-TRACE(trace)

```

Figura 5.32: Procedimento DEBUG-CONFLICT.

## 5.6 Conclusão

Este capítulo apresentou a ferramenta SAIDE, um ambiente de desenvolvimento de analisadores LALR( $k$ ).

Quando comparada com as ferramentas apresentadas no Capítulo 2 e aos geradores de analisadores LALR, a ferramenta supre a demanda por recursos que facilitam a remoção de conflitos, além de prover um único ambiente no qual é permitido a utilização de mais de uma linguagem de especificação, conforme a arquitetura descrita. Confrontada com Visual Parse++, ferramenta que mais se assemelha à proposta neste trabalho, SAIDE, além da independência de linguagem de especificação, dá ao projetista a flexibilidade de determinar o nível de compactação a ser aplicado às tabelas do analisador LALR( $k$ ) produzido, conforme o perfil da aplicação que irá executá-lo. Isto é um recurso também não oferecida por geradores LALR atualmente disponíveis [Johnson, 1979, CUP, 2007, Bison, 2007, SableCC, 2007]. No quesito de remoção automática, Visual Parse++ e SAIDE são equivalentes, pois permitem a remoção dos conflitos ocasionados pela falta de contexto à direita, resolvendo-os, sempre que possível,

pelo aumento do valor de  $k$ . Entretanto, na remoção manual de conflitos, SAIDE se destaca pelo suporte a uma metodologia elaborada como parte desta dissertação, que sistematiza e reduz o tempo gasto pelo usuário no processo de remoção<sup>3</sup>. Outro ponto a ser destacado é que na aplicação das fases da metodologia durante a remoção manual, SAIDE ordena os conflitos segundo uma prioridade de remoção. Esta característica, não encontrada em nenhuma outra ferramenta e/ou gerador, faz com que conflitos reportados em decorrência da existência de outros sejam automaticamente eliminados quando estes são removidos.

Os algoritmos implementados na ferramenta SAIDE possuem alguns diferenciais dos originais nos quais se baseiam. Na parte de construção das árvores de derivação, mostrou-se que a viabilidade do algoritmo proposto por DeRemer e Pennello, descrito em [DeRemer e Pennello, 1982] é dependente de um sistema de controle de derivações cíclicas não considerado pelos seus criadores. No cômputo de *lookaheads* de tamanho variável, os algoritmos apresentados se diferem dos mostrados por Charles [Charles, 1991] por garantir o término de execução independentemente de derivações cíclicas na gramática e/ou de ciclos nas relações *reads* e *includes*. Isto torna a metodologia de remoção de conflitos aplicável a todos os conflitos.

---

<sup>3</sup>Testes comparativos com a ferramenta Visual Parse++ não foram realizados porque a versão de teste disponível na página do fabricante <http://www.sand-stone.com/> depende de um código enviado pelo SandStone, empresa que distribui o programa. Apesar de ser possível baixar a ferramenta, SandStone suspendeu a distribuição desses códigos e o suporte à ferramenta. Maiores informações podem ser encontradas na *thread* de discussão em <http://compilers.iecc.com/comparch/article/07-02-038>

# Capítulo 6

## Avaliação

De forma a avaliar a corretude dos algoritmos propostos anteriormente, este capítulo apresenta oito pequenas especificações escritas utilizando-se a ferramenta SAIDE. Essas especificações possibilitam a realização de testes caixa preta cujos resultados são facilmente verificáveis. Em seguida, relata-se o resultado de um experimento com as linguagens Algol-60, Scheme, Oberon-2 e Notus para aferir o número de conflitos que podem ser removidos automaticamente pelo aumento de  $k$ , juntamente com a medição do tempo de execução desse processo. Para testar a metodologia e o ambiente proposto, reporta-se a experiência de criação do analisador sintático para a linguagem Machina[Bigonha et al., 2006].

### 6.1 Testes Caixa Preta

Esta seção apresenta os resultados de experimentos realizados com oito gramáticas de teste, cujas características são mostradas na Tabela 6.1. Os arquivos de especificação foram escritos na linguagem aceita pelo gerador CUP com a utilização do *plugin cup.jar*, implementado exclusivamente para fins de validação. A máquina LALR(0) gerada por SAIDE foi verificada com a produzida pelo CUP <sup>1</sup>. Quando aplicável, esta verificação também foi estendida para o número de conflitos. A corretude da eliminação dos conflitos e dos *lookaheads* foi avaliada manualmente, o que é facilitado pelo tamanho reduzido das gramáticas.

As especificações de 1 a 6 são especificações cujos conflitos não podem ser resolvidos pelo aumento de  $k$ . O propósito delas é a conferência do número de conflitos identificados por *SET-CONFLICTS*, que neste caso deve ser sempre o mesmo, inde-

---

<sup>1</sup>Como a numeração dos estados nem sempre é a mesma, utilizou-se um mapeamento de estados do autômato gerado por SAIDE nos correspondentes do autômato construído pelo CUP.

Gramática	$ \Sigma $	$ N $	$ P $	Possui solução (aumento de $k$ )
test1.cup	5	4	5	não
test2.cup	7	3	8	não
test3.cup	5	6	9	não
test4.cup	2	5	6	não
test5.cup	3	4	6	não
test6.cup	1	3	4	não
test7.cup	7	7	11	sim
test8.cup	14	18	32	sim

Tabela 6.1: Pequenas gramáticas utilizadas no processo de validação.

Teste	$k_{max} = 1$		$k_{max} = 2$		$k_{max} = 3$	
	Conflitos	Estados	Conflitos	Estados	Conflitos	Estados
test1.cup	1	10	1	11	1	12
test2.cup	16	13	16	29	16	61
test3.cup	1	13	1	14	1	16
test4.cup	2	7	2	9	2	9
test5.cup	4	7	4	11	4	15
test6.cup	1	3	1	3	1	3
test7.cup	1	18	0	19	0	19
test8.cup	16	41	1	47	0	48

Tabela 6.2: Experimentos realizados com as gramáticas de teste.

```

terminal IF, THEN, ELSE, TRUE ;

nonterminal stmt, if_stmt, exp ;

stmt ::= if_stmt ;

exp ::= TRUE ;

if_stmt ::= IF exp THEN stmt ELSE stmt |
          IF exp THEN stmt ;

```

Figura 6.1: Especificação test1.cup.

pendentemente do valor de  $k$ <sup>2</sup>. Exceto pela sexta especificação, a utilização de AFDLs para  $k \leq k_{max} = 2$  para essas gramáticas implica que o número de estados *Action* é igual ao número de entradas com conflito nessa tabela mais o número de linhas em *Action* quando  $k = 1$ . Note que na especificação test4.cup, o número de estados

<sup>2</sup>Isto também é válido para os procedimentos *CFTS-1*, *CFTS-K* e *TRAVERSE*.



```

terminal PLUS, MINUS, MULT, PER, NUM, ID ;

nonterminal exp, term ;

exp ::= exp PLUS exp |
      exp MINUS exp |
      exp MULT exp |
      exp PER exp |
      term ;

term ::= NUM |
       ID ;

```

Figura 6.2: Especificação `test2.cup`.

em  $k_{max} = 3$  não é diferente do obtido para  $k_{max} = 2$ . Para esta especificação, SAIDE reporta os seguintes conflitos:

**Error: Shift/Reduce conflict found in state 0**

involving items:

B -> .

A -> . a

under symbol a

**Error: Shift/Reduce conflict found in state 6**

involving items:

B -> .

A -> . a

under symbol a

Pela gramática da especificação, vê-se que o não terminal  $A$  produz somente **a**. Este símbolo só pode ser estendido pelo marcador de fim de arquivo. Como  $\$$  não é seguido por qualquer símbolo, o maior *lookahead* possível tem tamanho 2. Isto implica que o aumento de  $k$  não resolve o conflito. Além disto, a utilização de três símbolos de *lookahead* não aumenta o número de estados em *Action*. Esta situação também ocorre com a especificação `test6.cup`, cujo único conflito reportado tem  $\$$  como símbolo de *lookahead*. As especificações `test4.cup`, `test5.cup` e `test6.cup` são ainda utilizadas para aferir o término de execução. As Figuras 6.7 e 6.8 mostram um fragmento do

```
terminal
  ID,
  LPAR,
  RPAR,
  COMMA;

non terminal
  expression,
  array_access,
  function_call,
  expression_list,
  expression_list_opt;

start with expression ;

expression ::=
  array_access |
  function_call ;

array_access ::= ID LPAR expression_list RPAR ;

function_call ::= ID LPAR expression_list_opt RPAR ;

expression_list ::=
  expression |
  expression_list COMMA expression ;

expression_list_opt ::=
  /* lambda */ |
  expression_list ;
```

Figura 6.3: Especificação `test3.cup`.

autômato LALR(0) das gramáticas em `test4.cup` e `test5.cup`. A primeira figura mostra três estados que implicam na formação de um ciclo no grafo da relação *reads*, constituído pelos nodos que representam as transições  $(2, C)$ ,  $(5, D)$  e  $(6, B)$ . No segundo fragmento, o ciclo no grafo de inclusão é devido aos nodos que representam as transições  $(3, A)$  e  $(3, C)$ . Na especificação `test6.cup` tem-se o caso em que um não terminal gera ele mesmo em zero ou mais passos, sem o consumo de nenhum *token*. As três situações representadas por essas especificações acarretam na não terminação dos algoritmos originalmente propostos por Charles [Charles, 1991]. No entanto, as alterações mencionadas na Seção 5.5 garantem o fim da execução, o que é observado pelos testes realizados.

```
terminal a ;  
  
nonterminal A, B, C, D ;  
  
A ::= B C D A ;  
  
A ::= a ;  
  
B ::= /* lambda */ ;  
  
C ::= /* lambda */ ;  
  
D ::= /* lambda */ ;
```

Figura 6.4: Especificação `test4.cup`.

```
terminal a, e;  
  
non terminal A, C, D;  
  
start with A ;  
  
A ::= a | C ;  
  
C ::= D A ;  
  
D ::= /* lambda */ | e ;
```

Figura 6.5: Especificação `test5.cup`.

```
non terminal A, B ;  
  
A ::= B ;  
  
B ::=  
  A |  
  /* lambda */ ;
```

Figura 6.6: Especificação `test6.cup`.

As especificações `test7.cup` e `test8.cup` testam a remoção automática de conflitos. A primeira especificação consiste em uma gramática que define assinatura de funções aos moldes da linguagem C, exceto que, para um mesmo tipo, é permitido a definição

de um ou mais parâmetros formais, como por exemplo `int soma(int op1, op2)`. A linguagem expressa pela gramática é LALR(2), o que está de acordo com o resultado mostrado. Ao fazer  $k_{max} = 3$ , nenhum novo estado é criado em *Action*. Assim, apesar de se permitir o uso de até três símbolos de leitura, o analisador gerado consulta no máximo 2.

A gramática em `test8.cup` é um fragmento da gramática de Notus, cujos conflitos são resolvidos inspecionando-se até três símbolos à frente. A tabela *Action* construída com  $k_{max} = 3$  para essa especificação, juntamente com a tabela *Goto*, estão disponíveis no Apêndice B deste documento.

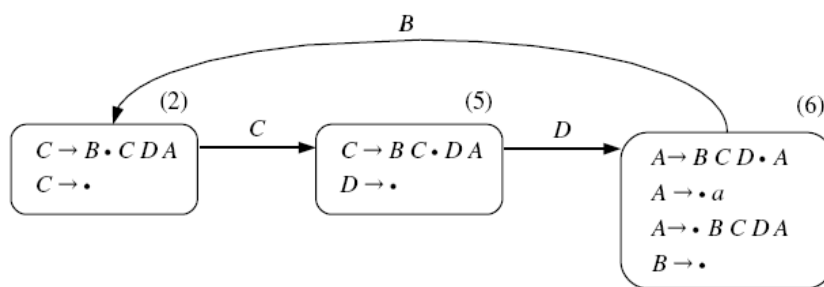


Figura 6.7: Fragmento do autômato LALR(0) da especificação `test4.cup`.

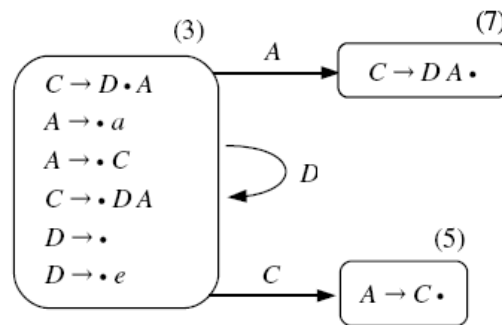


Figura 6.8: Fragmento do autômato LALR(0) da especificação `test5.cup`.

```
terminal
  COMMA,
  LPAR,
  RPAR,
  INT,
  FLOAT,
  ID;

non terminal
  func_header,
  formal_param_list_opt,
  formal_param_list,
  formal_param_section,
  simple_type,
  id_list;

start with func_header ;

func_header ::= simple_type ID LPAR formal_param_list_opt RPAR ;

formal_param_list_opt ::=
  /* lambda */ |
  formal_param_list ;

formal_param_list ::=
  formal_param_section |
  formal_param_list COMMA formal_param_section ;

formal_param_section ::= simple_type id_list ;

simple_type ::=
  FLOAT |
  INT ;

id_list ::=
  ID |
  id_list COMMA ID ;
```

Figura 6.9: Especificação test7.cup.

```
terminal
  ARROW,
  BAR,
  DOT,
  DOMAIN_ID,
  ID,
  LPAR,
  RPAR,
  PUBLIC,
  PRIVATE,
  LBRACE,
  RBRACE,
  COMMA,
  ASTERIX;

non terminal
  primary_domain_exp,
  domain_full_name,
  tuple_domain_exp,
  visibility,
  constructor,
  enum_domain_exp,
  enumerand_name_comma_list_opt,
  enumerand_name_comma_list,
  domain_exp_comma_list_opt,
  domain_exp_comma_list,
  domain_exp,
  union_domain_exp,
  function_domain_exp,
  inst_domain_exp,
  list_domain_exp,
  module_full_name,
  module_qualification;

primary_domain_exp ::=
  domain_full_name |
  enum_domain_exp |
  tuple_domain_exp ;

domain_full_name ::=
  module_full_name DOT DOMAIN_ID |
  DOMAIN_ID ;

module_full_name ::= module_qualification DOMAIN_ID ;

module_qualification ::=
  module_qualification DOMAIN_ID DOT |
  /* lambda */ ;
```

```
tuple_domain_exp ::=
  visibility constructor LPAR domain_exp_comma_list_opt RPAR ;

visibility ::=
  PUBLIC |
  PRIVATE |
  /* lambda */ ;

constructor ::=
  DOMAIN_ID |
  /* lambda */ ;

enum_domain_exp ::=
  visibility LBRACE enumerand_name_comma_list_opt RBRACE ;

enumerand_name_comma_list_opt ::=
  enumerand_name_comma_list |
  /* lambda */ ;

enumerand_name_comma_list ::=
  enumerand_name_comma_list COMMA ID |
  ID ;

domain_exp_comma_list_opt ::=
  domain_exp_comma_list |
  /* lambda */ ;

domain_exp_comma_list ::=
  domain_exp_comma_list COMMA domain_exp |
  domain_exp ;

domain_exp ::= union_domain_exp ;

union_domain_exp ::=
  union_domain_exp BAR function_domain_exp |
  function_domain_exp ;

function_domain_exp ::=
  inst_domain_exp ARROW function_domain_exp |
  inst_domain_exp ;

inst_domain_exp ::= list_domain_exp ;

list_domain_exp ::=
  list_domain_exp ASTERIX |
  primary_domain_exp ;
```

Figura 6.10: Especificação test8.cup.

Gramática	Produções	Terminais	Não terminais	Conflitos	Conflitos/ Produção
Algol-60	131	58	67	61	0.47
Scheme	175	42	83	78	0.45
Oberon-2	213	75	112	32	0.15
Notus	236	77	110	575	2.44

Tabela 6.3: Número de conflitos encontrados nas gramáticas de Algol-60, Scheme, Oberon-2 e Notus.

Gramática	$k_{max} = 1$		$k_{max} = 2$		$k_{max} = 3$		$k_{max} = 4$	
	Confs.	T. (ms)	Confs.	T. (ms)	Confs.	T. (ms)	Confs.	T. (ms)
Algol-60	61	670	61	2.480	61	4.837	61	18.017
Scheme	78	839	38	1.320	38	2.485	38	4.957
Oberon-2	32	898	1	1.464	1	1.567	1	1.493
Notus	575	1.068	541	34.828	539	38.921	539	67.467

Tabela 6.4: Avaliação do aumento de  $k$  na remoção automática de conflitos para 4 gramáticas de teste.

## 6.2 Remoção Automática de Conflitos

Para avaliar o percentual de conflitos que podem ser removidos automaticamente, fez-se um experimento com as gramáticas das linguagens Algol-60, Scheme, Oberon-2 e Notus, cujas características são mostradas na Tabela 6.3.

A Tabela 6.4 mostra o número de conflitos obtidos para  $k_{max} \leq 4$  e o tempo gasto na tentativa de removê-los. A unidade de tempo é milisegundos e a medição foi feita executando-se SAIDE em um PC AMD Athlon 3.0 Ghz (64 bits), com 1 Gb de memória RAM e SO Linux Ubuntu 7.04 (kernel 2.6.20-15).

Pelos resultados mostrados, vê-se que o aumento do número de *lookaheads* inspecionados é capaz de resolver parte dos conflitos, o que está em conformidade com a divisão representada pela Figura 3.1. Nas gramáticas de Scheme e Oberon-2, o número de conflitos para  $k_{max} = 2$  é reduzido em 51% e 97% em relação a  $k_{max} = 1$ . Para  $3 \leq k_{max} \leq 4$ , o número de conflitos permanece constante nessas gramáticas. Em Notus e Algol-60, os resultados foram bem inferiores. Nessas gramáticas, a única remoção obtida foi em Notus com  $k_{max} = 2$ , o que em termos percentuais corresponde a 6% de conflitos eliminados. A discrepância entre os resultados obtidos para Scheme e Oberon-2 em relação a Algol-60 e Notus é que as gramáticas destas linguagens estão escritas em um nível de abstração maior em relação às primeiras, mais apropriadas ao entendimento humano do que como entradas a geradores de analisadores sintáticos. Como consequência, o número de conflitos e a complexidade para removê-los automa-



ticamente é maior.

Como efeito colateral do aumento de  $k_{max}$ , tem-se um aumento porcentual considerável no tempo gasto na eliminação automática de conflitos, embora do ponto de vista prático o tempo de resposta tenha se mantido em níveis aceitáveis.

### 6.3 Estudo de Caso: Machĭna

Esta seção reporta a experiência de criação do analisador sintático para a linguagem Machĭna<sup>3</sup> no ambiente e metodologia da ferramenta SAIDE.

A escrita da especificação sintática de Machĭna foi dividida em 12 incrementos em que cada um resulta em uma especificação  $\mathcal{S}_i$ . Um incremento  $i$  consiste em um conjunto autocontido  $\mathcal{R}$  de regras gramaticais; cada regra em  $\mathcal{R}$  depende somente de não terminais definidos em  $\mathcal{R}$  ou em  $\mathcal{S}_{i-1}$ . A transição de um incremento  $i$  ao incremento  $i + 1$  é feita quando o número de conflitos em  $i$  é zero. Os conflitos removidos a cada incremento formam uma seqüência ordenada de passos.

Durante a criação do analisador sintático, utilizaram-se até 2 símbolos de *lookaheads* na tentativa de remoção automática de conflitos. A cada conflito não removido automaticamente, aplicaram-se as quatro fases definidas na metodologia proposta no Capítulo 5. A Figura 6.11 mostra o gráfico de pontos que representa o número de conflitos obtidos a cada passo. Para os 12 incrementos, foram utilizados 27 passos. Nesse gráfico, cada passo corresponde a uma compilação da especificação. O fim de um incremento é marcado por uma linha vertical pontilhada, ponto em que o número de conflitos é igual a zero. Para facilitar a identificação dos intervalos em que ocorre aumento de conflitos, os pontos do gráfico são ligados por linhas.

Com exceção dos passos 8 e 9 no incremento 4, a cada passo dentro de um incremento o número de conflitos foi reduzido. O aumento de conflitos ocorre somente entre a transição de dois incrementos consecutivos, ocasionado pelo acréscimo de produções. Para o experimento em questão, a metodologia comportou-se como esperado: desde que seus passos sejam corretamente aplicados pelo usuário, tem-se a constante diminuição de conflitos. É importante mencionar que na remoção manual de conflitos o autômato LALR não foi consultado; a interpretação do conflito na fase de entendimento baseou-se somente na visualização das árvores de derivação.

O aumento de conflitos entre os passos 8 e 9 é devido a uma interpretação incompleta dos fatores que o ocasionaram. O conflito em questão é mostrado na Figura 6.12. Para facilitar a exibição, somente parte das árvores de derivação obtidas pela depuração do

---

<sup>3</sup>Machĭna é uma linguagem de programação para definição de especificações formais baseadas no modelo ASM [Gurevich, ].

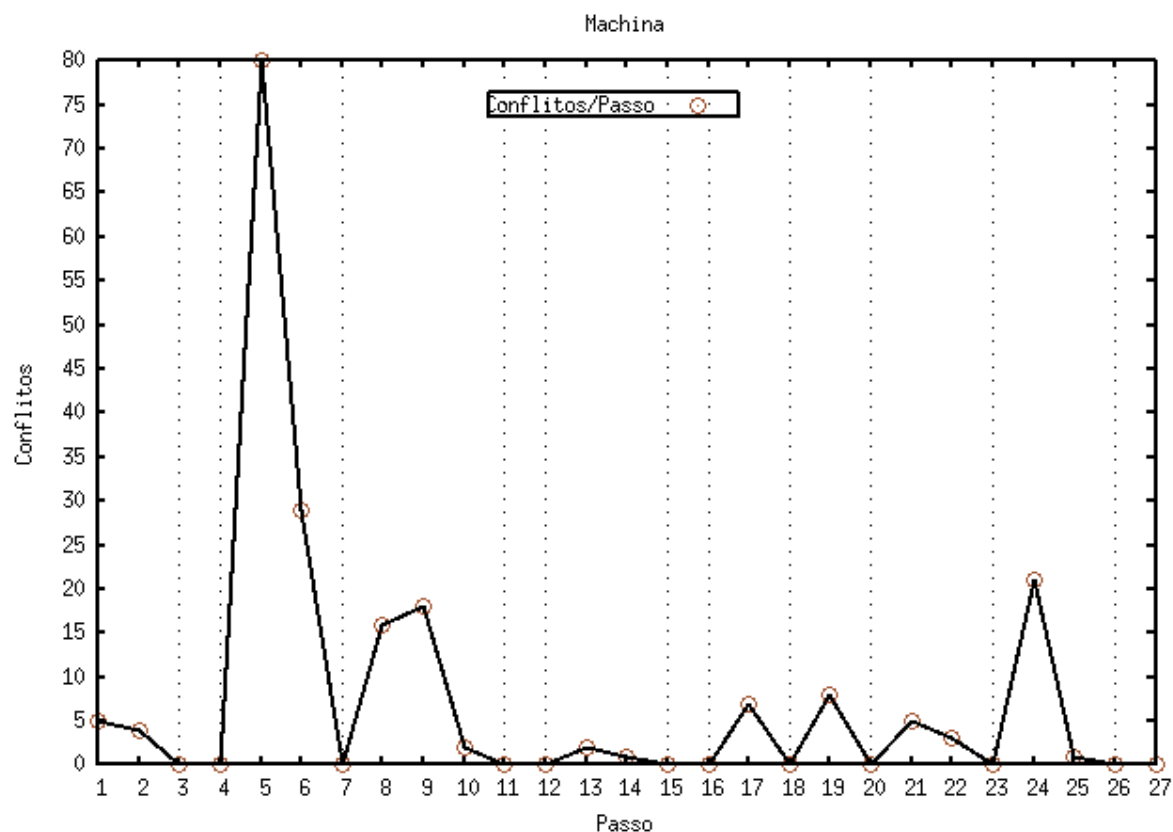


Figura 6.11: Gráfico de pontos dos conflitos obtidos a cada passo de construção do analisador sintático de Machina.

```

Error: Reduce/Reduce conflict found in state 102

involving items:

type_designator -> type_name .
formal_type -> type_name .
type_expression -> type_name .

under symbol PUBLIC

```

Figura 6.12: Conflito *reduce/reduce* reportado por SAIDE segundo a prioridade de remoção. Esse conflito é obtido no passo 8.

conflito é mostrada nas Figuras 6.13, 6.14 e 6.15, em que cada uma se refere a um item do conflito. O não terminal `type_name` é definido por:

```
type_name ::= ID ;
```

Pela interpretação das árvores de derivação mostradas e pela consulta aos padrões disponíveis na fase de classificação, a solução utilizada na remoção do conflito foi a junção das regras `formal_type ::= type_name` e `type_expression ::= type_name`

quando utilizadas no lado direito de `type_expression`. Essa estratégia é a descrita por SAIDE para conflitos ocasionados por *alias* entre não terminais. Desta forma, `type_expression` é redefinido por:

```
type_expression ::=
    formal_type_or_type_name | ... | user_defined_type | ... ;
```

onde `formal_type_or_type_name` é dado por

```
formal_type_or_type_name ::=
    ID ;
```

Essa alteração, no entanto, gera um novo conflito, conforme mostrado na Figura 6.16.

```
algebra_part_opt
algebra_part
ALGEBRA COLON algebra_section_list_opt
                algebra_section_list
                algebra_section_list algebra_section
                                type_section
                                PUBLIC type_declaration
                algebra_section_list algebra_section
                                external_section
                                external_declaration
                                EXTERNAL external_function COLON type_expression
                                                                user_defined_type
                                                                type_designator
                                                                type_name
```

Figura 6.13: Fragmento da primeira árvore de derivação referente ao item `type_designator -> type_name` . do conflito mostrado na Figura 6.12.

```
algebra_part_opt
algebra_part
ALGEBRA COLON algebra_section_list_opt
                algebra_section_list
                algebra_section_list algebra_section
                                type_section
                                PUBLIC type_declaration
                algebra_section_list algebra_section
                                external_section
                                external_declaration
                                EXTERNAL external_function COLON type_expression
                                                                formal_type
                                                                type_name
```

Figura 6.14: Fragmento da segunda árvore de derivação referente ao item `formal_type -> type_name` . do conflito mostrado na Figura 6.12.

```

algebra_part_opt
algebra_part
ALGEBRA COLON algebra_section_list_opt
                algebra_section_list
                algebra_section_list algebra_section
                                type_section
                                PUBLIC type_declaration
                algebra_section_list algebra_section
                                external_section
                                external_declaration
                                EXTERNAL external_function COLON type_expression
                                                                type_name

```

Figura 6.15: Fragmento da terceira árvore de derivação referente ao item `type_expression -> type_name .` do conflito mostrado na Figura 6.12.

```

Error: Reduce/Reduce conflict found in state 122
involving items:
formal_type_or_type_name -> ID .
type_name -> ID .

under symbol PUBLIC

```

Figura 6.16: Conflito *reduce/reduce* reportado por SAIDE segundo a prioridade de remoção. Esse conflito é obtido no passo 9.

```

algebra_part_opt
algebra_part
ALGEBRA COLON algebra_section_list_opt
                algebra_section_list
                algebra_section_list algebra_section
                                type_section
                                PUBLIC type_declaration
                algebra_section_list algebra_section
                                external_section
                                external_declaration
                                EXTERNAL external_function COLON type_expression
                                                                formal_type_or_type_name
                                                                ID

```

Figura 6.17: Fragmento da primeira árvore de derivação referente ao item `formal_type_or_type_name -> ID .` do conflito mostrado na Figura 6.16.

Esse conflito se dá em decorrência da incompletude de interpretação do primeiro conflito. Vê-se que a estratégia de remoção inicial desconsiderou o fato de que o não terminal `user_defined_type` também produz `type_name`. Isto é comprovado pelas árvores de depuração mostradas nas Figuras 6.13, 6.17 e 6.18.

A estratégia de remoção, no entanto, permanece a mesma: combinar as produções

```

algebra_part_opt
algebra_part
ALGEBRA COLON algebra_section_list_opt
                algebra_section_list
                algebra_section_list algebra_section
                                type_section
                                PUBLIC type_declaration
                algebra_section_list algebra_section
                                external_section
                                external_declaration
                                EXTERNAL external_function COLON type_expression
                                                                user_defined_type
                                                                type_designator
                                                                type_name
                                                                ID

```

Figura 6.18: Fragmento da segunda árvore de derivação referente ao item `type_name`  $\rightarrow$  `ID` . do conflito mostrado na Figura 6.16.

`formal_type_or_type_name`  $\rightarrow$  `ID` e `type_name`  $\rightarrow$  `ID`. Desta forma, `type_expression` é alterado para:

```

type_expression ::=
    ...
    type_name_or_user_defined_type |
    ... ;

```

onde

```

type_name_or_user_defined_type ::=
    type_designator LPAR type_arguments RPAR |
    type_designator_or_type_name ;

```

```

type_designator_or_type_name ::=
    module_name DOT type_name |
    type_name ;

```

Isto reduz o número de conflitos a dois, conforme indicado no gráfico da Figura 6.11.

A gramática final de Machĩna obtida no experimento contém 105 terminais, 241 não terminais e 450 produções. As características das tabelas *Action* e *Goto* do analisador LALR com  $k_{max} = 2$  produzido são mostradas na Figura 6.5. Os resultados da compactação das tabelas do analisador gerado segundo os dois níveis disponíveis em SAIDE são apresentados nas Tabelas 6.6 e 6.7. Note que os resultados são semelhantes aos obtidos no Capítulo 4. Como feito anteriormente, o número de *bytes* considerado para qualquer entrada, exceto as da matriz *Sigma*, é de 2 bytes.

<i>Action</i>		
Linhas	Colunas	Total (bytes)
703	105	147.630
<i>Goto</i>		
701	241	337.882

Tabela 6.5: Tabelas *Action* e *Goto* não compactadas do analisador LALR com  $k_{max} = 2$  de Machina.

$ value $	Total (bytes)
4.439	8.894
<b>Compactação (%)</b>	98

Tabela 6.6: Compactação das tabelas do analisador de Machina pelo método BCS.

<i>Action</i>								
Linhas	Colunas	$ dr $	$ dc $	$ r $	$ v $	<i>Sigmap</i> (bytes)	<i>RMS</i> (bytes)	Total
83	51	608	105	608	105	10.246	1.406	22.970
<b>Compactação (%):</b>								84,4
<i>Goto</i>								
63	24	701	241	701	241	-	-	6.792
<b>Compactação (%):</b>								98

Tabela 6.7: Compactação das tabelas do analisador de Machina pelas composições GCS ◦ LES ◦ RMS, para (*Action*), e GCS ◦ LES, para *Goto*.

## 6.4 Conclusão

Este capítulo apresentou três partes no processo de avaliação da ferramenta e metodologia propostas nesta dissertação.

A primeira delas consistiu na realização de testes caixa preta de pequenas especificações cujos resultados são facilmente verificáveis e puderam ser conferidos manualmente.

Na segunda parte, foram realizados testes com as gramáticas Algol-60, Scheme, Oberon-2 e Notus para capturar o porcentual de conflitos que podem ser removidos pelo aumento de  $k_{max}$ . Embora as gramáticas de Scheme e Oberon-2 tenham tido 51% e 97% dos conflitos eliminados, Notus apresentou apenas 6% de redução e Algol-60 não teve nenhum conflito removido automaticamente. Disto, conclui-se que a remoção automática utilizada por SAIDE é um recurso que resolve parte dos conflitos e é dependente da gramática em questão. Nos testes realizados, essa dependência está ligada

ao nível de abstração da gramática; quanto maior o nível de abstração maior a inadequação para remoção automática de conflitos. Este foi o caso das gramáticas das linguagens Notus e Algol-60, escritas originalmente para leitura humana, ao contrário das gramáticas de Oberon-2 e Scheme, mais próximas à confecção de analisadores sintáticos.

Na avaliação da metodologia e do ambiente proposto reportou-se a experiência de criação do analisador sintático de Machina na ferramenta SAIDE, utilizando-se até dois *tokens* de *lookaheads*. No experimento, cada passo da aplicação da metodologia contribuiu para a constante redução de conflitos, o que sugere sua corretude para o teste em questão. Mostrou-se ainda, a suficiência da utilização das árvores de derivação como fonte de entendimento do conflito, uma vez que em nenhum momento o autômato LALR foi consultado. Na geração do analisador sintático, a tabela correspondente foi compactada segundo os níveis de compactação alto e médio, obtidos respectivamente pela utilização dos métodos de compressão BCS e pela combinação de GCS, LES e RMS. Isto resultou em tabelas  $\approx 98\%$  e  $\approx 94\%$  menores.





# Capítulo 7

## Conclusão

Este trabalho apresentou a ferramenta SAIDE, um protótipo de um ambiente integrado de desenvolvimento que permite a aplicação de uma metodologia na remoção de conflitos LALR.

Mostrou-se que o problema de remoção de conflitos é extremamente recorrente durante a construção de analisadores sintáticos LALR e que tanto os geradores quanto as ferramentas visuais atualmente disponíveis dão pouco suporte a esta questão.

O primeiro passo na elaboração da metodologia partiu do entendimento do funcionamento dos analisadores LALR e na enumeração das possíveis situações que acarretam em conflitos, que são divididos em dois grandes conjuntos: os de ambigüidade e os decorrentes de ausência de contexto à direita. Da definição da fronteira entre conflitos LR e LALR, provou-se que conflitos específicos a LALR não implicam em ambigüidade, o que é uma importante informação no processo de classificação.

A metodologia proposta sistematiza o processo de remoção, definindo um conjunto de fases e operações de auxílio na remoção de conflitos. No suporte à remoção automática de conflitos, SAIDE elimina parte dos conflitos oriundos da falta de contexto à direita. Os algoritmos utilizados nesta tarefa são independentes de qualquer característica da gramática e detêm a garantia de término, duas propriedades importantes, mas originalmente não atendidas pelos algoritmos de Charles, utilizados como base. Para os testes realizados, a aplicação correta das fases de remoção manual guiaram o usuário na constante redução do número de conflitos.

A arquitetura da ferramenta garante sua extensibilidade pelo uso de *plugins*, que permitem a abstração em relação às linguagens de especificação. Com isto, SAIDE pode ser utilizado com qualquer linguagem de especificação, desde que exista um *plugin* correspondente implementado. Atualmente, tem-se somente o *plugin* para a linguagem de especificação aceita pelo gerador CUP. Esse *plugin* foi utilizado para permitir a avaliação da ferramenta.

Na parte de produção de analisadores sintáticos, SAIDE tem a vantagem de gerar analisadores sintáticos cujas tabelas são compactadas segundo dois níveis: alto e médio. A compactação alta é obtida pelo uso da estratégia BCS, que reduz o tamanho da tabela a menos de 5% em relação ao tamanho original, embora altere o tempo de acesso teórico a uma entrada e com possível atraso na detecção de erros. O nível médio utiliza a combinação dos métodos RMS, GCS e LES para compactação da tabela *Action* e GCS e LES para a tabela *Goto*. A taxa de compactação desse método é próxima a 90%, com a preservação do acesso teórico  $O(1)$  e do instante de detecção de erros. Esses dois métodos foram escolhidos a partir da implementação e realização de testes com oito métodos de compactação.

As seguintes contribuições são resultado desta dissertação de mestrado:

- a metodologia proposta sistematiza a remoção de conflitos, que é um processo laborioso e consome grande parte do tempo gasto na construção de analisadores LALR;
- criação da ferramenta SAIDE, de forma a dar suporte à metodologia proposta;
- alteração nos algoritmos de cômputo de *lookaheads* de tamanho  $k$  propostos por Charles [Charles, 1991], de forma a garantir o término de execução independentemente de certas características da gramática. Isto uniformiza o processo de remoção de conflitos;
- alteração dos algoritmos de DeRemer e Pennello [DeRemer e Pennello, 1982] para controle de derivações cíclicas. Embora os algoritmos originais estejam corretos, sua utilização sem as alterações mencionadas é inviável, pois rapidamente consome-se toda a memória do computador;
- criação de uma heurística para listar os conflitos segundo a ordem em que devem ser removidos. Esta ordenação captura as situações em que conflitos são ocasionados em decorrência de outros; com isto a remoção destes implicam na remoção dos primeiros;
- avaliação de oito métodos de compactação das tabelas sintáticas dos analisadores LALR(1) das linguagens C, C#, HTML, Java, VB .NET. Este estudo permitiu a identificação de dois importantes métodos de compactação: BCS e o obtido pela combinação dos métodos RMS, LES e GCS.
- adaptação do método BCS e o formado pela combinação RMS, LES e GCS para compactação de tabelas LALR( $k$ ), com  $k$  variável, geradas pela ferramenta SAIDE;

- flexibilização de escolha do nível de compactação do analisador sintático de acordo com o perfil da aplicação que irá executá-lo;
- definição de uma arquitetura cuja extensibilidade é garantida pela utilização de *plugins*, que permitem que SAIDE seja independente de linguagem de especificação, ao contrário do que ocorre nas ferramentas visuais estudadas.

Como parte dos trabalhos futuros, destacam-se as seguintes atividades:

- reescrita do componente referente à interface gráfica da ferramenta SAIDE de forma a integrá-la aos ambientes padrão de desenvolvimento atualmente disponíveis, como Eclipse e Netbeans;
- criação de *plugins* para suporte a linguagens de especificação de geradores LALR além do CUP, como Bison, SableCC, Happy, etc;
- mecanismo automático ou semi-automático para classificação de conflitos;
- identificação de novos padrões de conflitos de ambigüidade em linguagens de programação;
- disponibilização de todo o código fonte para a comunidade munido da documentação adequada para que outras pessoas possam colaborar com o projeto.



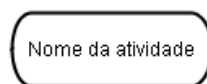
# Apêndice A

## Diagrama de Atividades

Um diagrama de atividades documenta um fluxo que vai de um estado inicial a um estado final, passando por atividades, decisões, bifurcações, barreiras de sincronização, etc. [Booch et al., 2005].

A execução do fluxo documentado no diagrama de atividade parte do estado inicial para o primeiro elemento a ele conectado por um fluxo de controle. O estado inicial de um diagrama de atividade é representado por um círculo preenchido com a cor preta. Fluxos de controle são representados por retas com uma seta aberta em uma das extremidades. Visam indicar a passagem de execução de um ponto antecessor a um ponto sucessor.

Um ponto de execução em um diagrama de atividades é denominado *atividade*. Uma atividade é um conjunto de ações atômicas que realizam alguma computação. Atividades são representadas em UML da seguinte forma



O estado final é opcional em um diagrama de atividades. A sua não existência modela um processo infinito. Estados finais são representados como



Como exemplo, considere o diagrama de atividades da Figura A.1.

Atividades podem receber e criar objetos. Por exemplo, no diagrama da Figura A.1, pode-se considerar como insumo para a atividade *Escrever carta* dois objetos: papel,

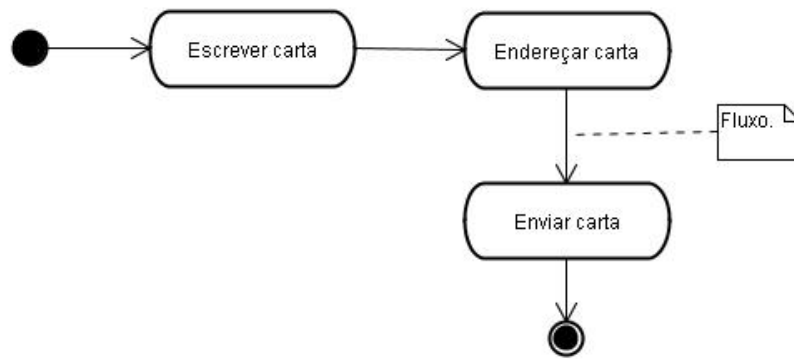


Figura A.1: Exemplo de um diagrama de atividades.

lápiz e borracha. Com isto, obtém-se o diagrama da Figura A.2. Nesse diagrama, os objetos, representados por caixas retangulares, são conectados à atividade por meio de fluxos de objetos. Esses fluxos são retas tracejadas com uma seta aberta em uma das extremidades.

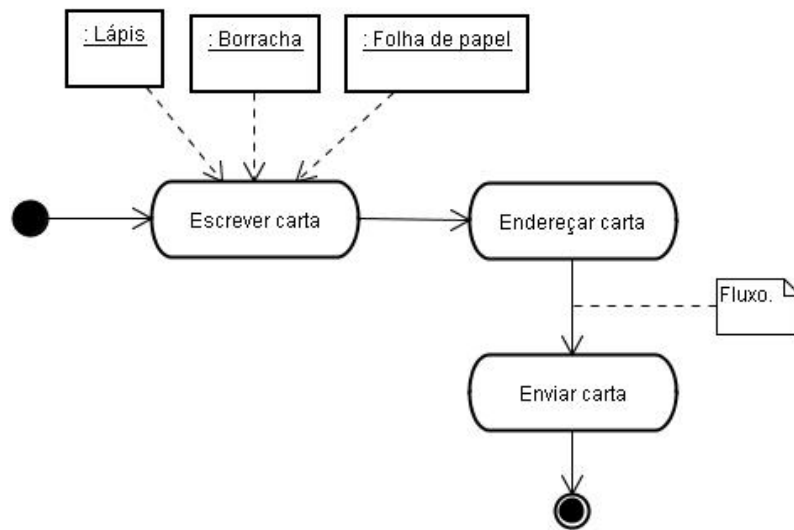


Figura A.2: Exemplo de um diagrama de atividades com a utilização de objetos como entrada para uma atividade.

Como resultado da atividade *Escrever carta*, pode-se considerar a folha preenchida. Assim, tem-se o mesmo objeto fornecido na entrada, mas com um estado diferente. O estado é escrito entre colchetes e colocado abaixo de seu nome. Disto, resulta o diagrama da Figura A.3. Nele, tem-se que a atividade *Escrever carta* altera o estado do objeto *carta*, que na entrada consiste de uma folha em branco, mas na saída encontra-se preenchida com algum texto. Este mesmo objeto, por sua vez, é utilizado como insumo para a atividade *Endereçar carta*, juntamente com o objeto *envelope*. O objeto

*envelope*, que ao ser utilizado como entrada para a atividade *Endereçar carta* possui o estado de não endereçado, ao final, tem o seu estado alterado para endereçado.

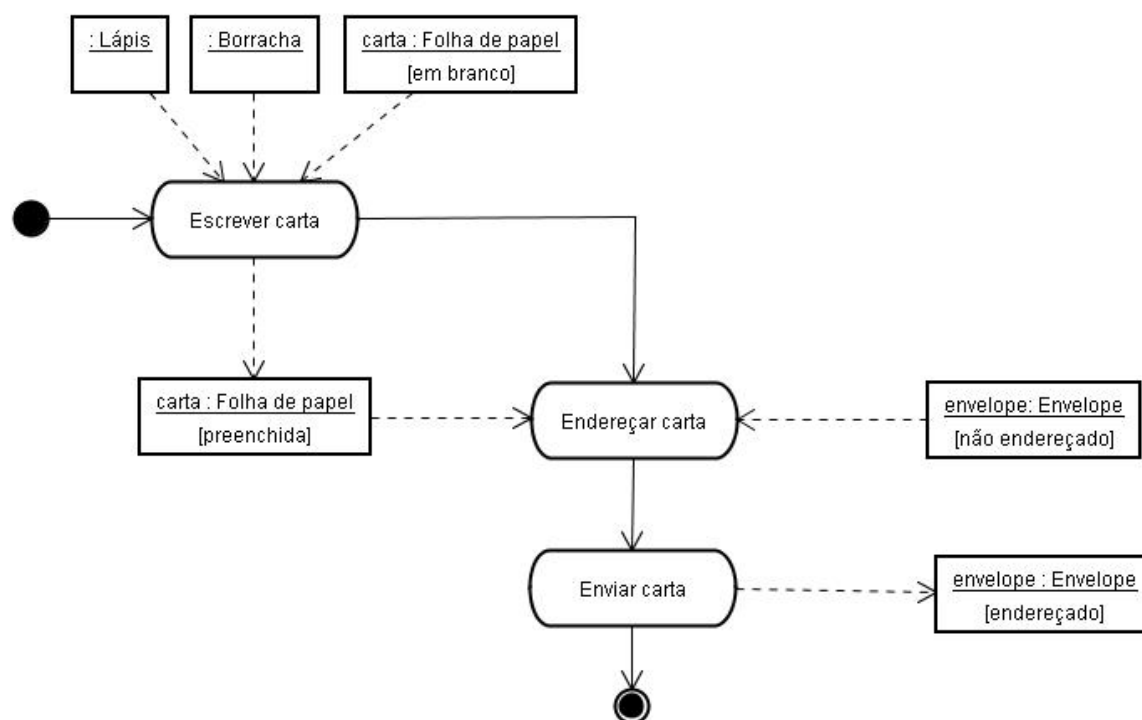


Figura A.3: Exemplo de um diagrama de atividades com a utilização de estados de objetos.

Outro recurso disponível em diagramas de atividades é a ramificação. Ramificações permitem que em um dado ponto do fluxo de execução, decida-se qual caminho seguir a partir da avaliação de um conjunto de guardas, que consistem em expressões booleanas contidas entre colchetes. O caminho a ser seguido refere-se à primeira guarda avaliada como verdadeiro. As ramificações são representadas como losangos com um fluxo de entrada e dois ou mais fluxos de saídas. Os fluxos de cada ramificação podem ser combinados <sup>1</sup> posteriormente em um único fluxo comum. Isto é obtido novamente com a utilização de um losango, mas com uma sintaxe diferente da anterior. Losangos de combinação tem dois ou mais fluxos de entrada e uma única transição de saída. Um exemplo simples de ramificação é apresentado na Figura A.4. A figura apresenta a expansão da atividade *Escrever carta*. Entende-se que a escrita de uma carta é escrever uma palavra por vez até que se tenha a carta por completo <sup>2</sup>.

<sup>1</sup>Tradução do termo *merged*.

<sup>2</sup>Detalhes como pontuação foram omitidos por questões de simplicidade.

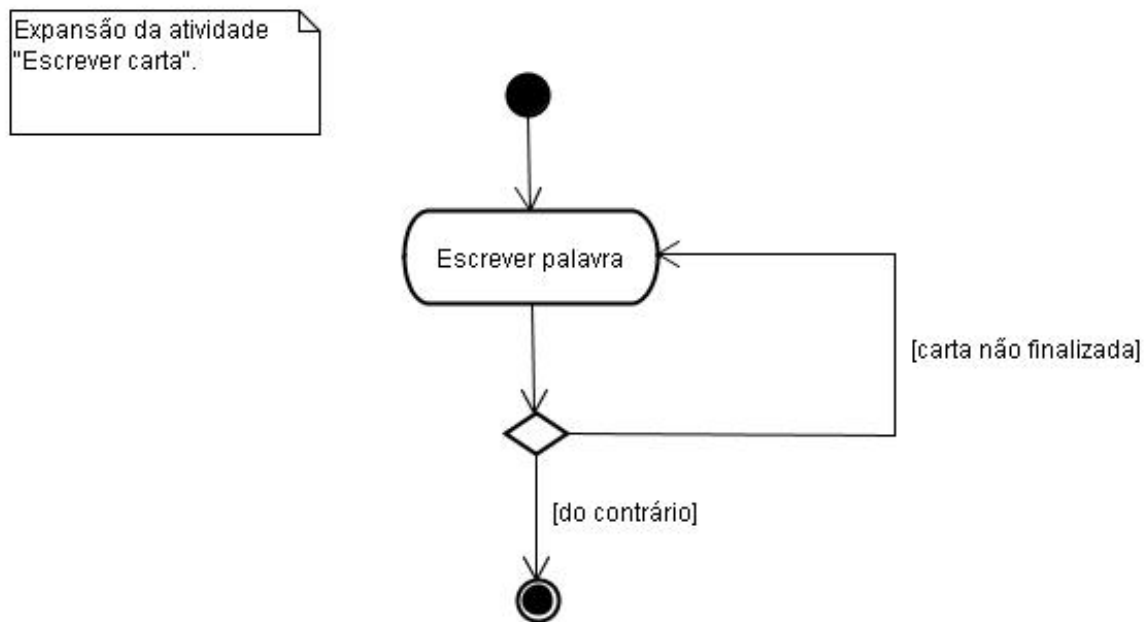


Figura A.4: Exemplo de um diagrama de estados com a utilização de ramificação.



# Apêndice B

## Tabelas *Action* e *Goto* da Gramática 6.10 ( $k_{max} = 3$ )

Writing parser with no compression

Codes (terminals):

0 = DOMAIN\_ID  
1 = RBRACE  
2 = PRIVATE  
3 = \$  
4 = LBRACE  
5 = ID  
6 = PUBLIC  
7 = COMMA  
8 = error  
9 = DOT  
10 = ARROW  
11 = ASTERIX  
12 = RPAR  
13 = BAR  
14 = LPAR

Codes (nonterminals):

0 = module\_full\_name  
1 = enum\_domain\_exp  
2 = inst\_domain\_exp  
3 = tuple\_domain\_exp

4 = enumerand\_name\_comma\_list  
 5 = enumerand\_name\_comma\_list\_opt  
 6 = function\_domain\_exp  
 7 = union\_domain\_exp  
 8 = \_\_\_start  
 9 = primary\_domain\_exp  
 10 = domain\_exp  
 11 = domain\_exp\_comma\_list  
 12 = visibility  
 13 = constructor  
 14 = domain\_full\_name  
 15 = list\_domain\_exp  
 16 = domain\_exp\_comma\_list\_opt  
 17 = module\_qualification

Codes (productions)

0 = module\_full\_name -> module\_qualification DOMAIN\_ID  
 1 = enum\_domain\_exp -> visibility LBRACE enumerand\_name\_comma\_list\_opt RBRACE  
 2 = inst\_domain\_exp -> list\_domain\_exp  
 3 = tuple\_domain\_exp -> visibility constructor LPAR domain\_exp\_comma\_list\_opt RPAR  
 4 = enumerand\_name\_comma\_list -> enumerand\_name\_comma\_list COMMA ID  
 5 = enumerand\_name\_comma\_list -> ID  
 6 = enumerand\_name\_comma\_list\_opt -> enumerand\_name\_comma\_list  
 7 = enumerand\_name\_comma\_list\_opt ->  
 8 = function\_domain\_exp -> inst\_domain\_exp ARROW function\_domain\_exp  
 9 = function\_domain\_exp -> inst\_domain\_exp  
 10 = union\_domain\_exp -> union\_domain\_exp BAR function\_domain\_exp  
 11 = union\_domain\_exp -> function\_domain\_exp  
 12 = \_\_\_start -> primary\_domain\_exp \$  
 13 = primary\_domain\_exp -> domain\_full\_name  
 14 = primary\_domain\_exp -> enum\_domain\_exp  
 15 = primary\_domain\_exp -> tuple\_domain\_exp  
 16 = domain\_exp -> union\_domain\_exp  
 17 = domain\_exp\_comma\_list -> domain\_exp\_comma\_list COMMA domain\_exp  
 18 = domain\_exp\_comma\_list -> domain\_exp  
 19 = visibility -> PUBLIC  
 20 = visibility -> PRIVATE  
 21 = visibility ->

---

```
22 = constructor -> DOMAIN_ID
23 = constructor ->
24 = domain_full_name -> module_full_name DOT DOMAIN_ID
25 = domain_full_name -> DOMAIN_ID
26 = list_domain_exp -> list_domain_exp ASTERIX
27 = list_domain_exp -> primary_domain_exp
28 = domain_exp_comma_list_opt -> domain_exp_comma_list
29 = domain_exp_comma_list_opt ->
30 = module_qualification -> module_qualification DOMAIN_ID DOT
31 = module_qualification ->
```

Action table:

```
action[0,0] = L42
action[0,1] = ERR
action[0,2] = S5
action[0,3] = ERR
action[0,4] = R21
action[0,5] = ERR
action[0,6] = S6
action[0,7] = ERR
action[0,8] = ERR
action[0,9] = ERR
action[0,10] = ERR
action[0,11] = ERR
action[0,12] = ERR
action[0,13] = ERR
action[0,14] = R21
action[1,0] = ERR
action[1,1] = ERR
action[1,2] = ERR
action[1,3] = ERR
action[1,4] = ERR
action[1,5] = ERR
action[1,6] = ERR
action[1,7] = ERR
action[1,8] = ERR
action[1,9] = S11
```

action[1,10] = ERR  
action[1,11] = ERR  
action[1,12] = ERR  
action[1,13] = ERR  
action[1,14] = ERR  
action[2,0] = ERR  
action[2,1] = ERR  
action[2,2] = ERR  
action[2,3] = R25  
action[2,4] = ERR  
action[2,5] = ERR  
action[2,6] = ERR  
action[2,7] = R25  
action[2,8] = ERR  
action[2,9] = ERR  
action[2,10] = R25  
action[2,11] = R25  
action[2,12] = R25  
action[2,13] = R25  
action[2,14] = ERR  
action[3,0] = ERR  
action[3,1] = ERR  
action[3,2] = ERR  
action[3,3] = R14  
action[3,4] = ERR  
action[3,5] = ERR  
action[3,6] = ERR  
action[3,7] = R14  
action[3,8] = ERR  
action[3,9] = ERR  
action[3,10] = R14  
action[3,11] = R14  
action[3,12] = R14  
action[3,13] = R14  
action[3,14] = ERR  
action[4,0] = ERR  
action[4,1] = ERR  
action[4,2] = ERR

---

action[4,3] = R15  
action[4,4] = ERR  
action[4,5] = ERR  
action[4,6] = ERR  
action[4,7] = R15  
action[4,8] = ERR  
action[4,9] = ERR  
action[4,10] = R15  
action[4,11] = R15  
action[4,12] = R15  
action[4,13] = R15  
action[4,14] = ERR  
action[5,0] = R20  
action[5,1] = ERR  
action[5,2] = ERR  
action[5,3] = ERR  
action[5,4] = R20  
action[5,5] = ERR  
action[5,6] = ERR  
action[5,7] = ERR  
action[5,8] = ERR  
action[5,9] = ERR  
action[5,10] = ERR  
action[5,11] = ERR  
action[5,12] = ERR  
action[5,13] = ERR  
action[5,14] = R20  
action[6,0] = R19  
action[6,1] = ERR  
action[6,2] = ERR  
action[6,3] = ERR  
action[6,4] = R19  
action[6,5] = ERR  
action[6,6] = ERR  
action[6,7] = ERR  
action[6,8] = ERR  
action[6,9] = ERR  
action[6,10] = ERR

action[6,11] = ERR  
action[6,12] = ERR  
action[6,13] = ERR  
action[6,14] = R19  
action[7,0] = ERR  
action[7,1] = ERR  
action[7,2] = ERR  
action[7,3] = S12  
action[7,4] = ERR  
action[7,5] = ERR  
action[7,6] = ERR  
action[7,7] = ERR  
action[7,8] = ERR  
action[7,9] = ERR  
action[7,10] = ERR  
action[7,11] = ERR  
action[7,12] = ERR  
action[7,13] = ERR  
action[7,14] = ERR  
action[8,0] = S13  
action[8,1] = ERR  
action[8,2] = ERR  
action[8,3] = ERR  
action[8,4] = S14  
action[8,5] = ERR  
action[8,6] = ERR  
action[8,7] = ERR  
action[8,8] = ERR  
action[8,9] = ERR  
action[8,10] = ERR  
action[8,11] = ERR  
action[8,12] = ERR  
action[8,13] = ERR  
action[8,14] = R23  
action[9,0] = ERR  
action[9,1] = ERR  
action[9,2] = ERR  
action[9,3] = R13

---

action[9,4] = ERR  
action[9,5] = ERR  
action[9,6] = ERR  
action[9,7] = R13  
action[9,8] = ERR  
action[9,9] = ERR  
action[9,10] = R13  
action[9,11] = R13  
action[9,12] = R13  
action[9,13] = R13  
action[9,14] = ERR  
action[10,0] = S16  
action[10,1] = ERR  
action[10,2] = ERR  
action[10,3] = ERR  
action[10,4] = ERR  
action[10,5] = ERR  
action[10,6] = ERR  
action[10,7] = ERR  
action[10,8] = ERR  
action[10,9] = ERR  
action[10,10] = ERR  
action[10,11] = ERR  
action[10,12] = ERR  
action[10,13] = ERR  
action[10,14] = ERR  
action[11,0] = S17  
action[11,1] = ERR  
action[11,2] = ERR  
action[11,3] = ERR  
action[11,4] = ERR  
action[11,5] = ERR  
action[11,6] = ERR  
action[11,7] = ERR  
action[11,8] = ERR  
action[11,9] = ERR  
action[11,10] = ERR  
action[11,11] = ERR

action[11,12] = ERR  
action[11,13] = ERR  
action[11,14] = ERR  
action[12,0] = ERR  
action[12,1] = ERR  
action[12,2] = ERR  
action[12,3] = ERR  
action[12,4] = ERR  
action[12,5] = ERR  
action[12,6] = ERR  
action[12,7] = ERR  
action[12,8] = ERR  
action[12,9] = ERR  
action[12,10] = ERR  
action[12,11] = ERR  
action[12,12] = ERR  
action[12,13] = ERR  
action[12,14] = ERR  
action[13,0] = ERR  
action[13,1] = ERR  
action[13,2] = ERR  
action[13,3] = ERR  
action[13,4] = ERR  
action[13,5] = ERR  
action[13,6] = ERR  
action[13,7] = ERR  
action[13,8] = ERR  
action[13,9] = ERR  
action[13,10] = ERR  
action[13,11] = ERR  
action[13,12] = ERR  
action[13,13] = ERR  
action[13,14] = R22  
action[14,0] = ERR  
action[14,1] = R7  
action[14,2] = ERR  
action[14,3] = ERR  
action[14,4] = ERR



---

action[14,5] = S20  
action[14,6] = ERR  
action[14,7] = ERR  
action[14,8] = ERR  
action[14,9] = ERR  
action[14,10] = ERR  
action[14,11] = ERR  
action[14,12] = ERR  
action[14,13] = ERR  
action[14,14] = ERR  
action[15,0] = ERR  
action[15,1] = ERR  
action[15,2] = ERR  
action[15,3] = ERR  
action[15,4] = ERR  
action[15,5] = ERR  
action[15,6] = ERR  
action[15,7] = ERR  
action[15,8] = ERR  
action[15,9] = ERR  
action[15,10] = ERR  
action[15,11] = ERR  
action[15,12] = ERR  
action[15,13] = ERR  
action[15,14] = S21  
action[16,0] = ERR  
action[16,1] = ERR  
action[16,2] = ERR  
action[16,3] = ERR  
action[16,4] = ERR  
action[16,5] = ERR  
action[16,6] = ERR  
action[16,7] = ERR  
action[16,8] = ERR  
action[16,9] = L43  
action[16,10] = ERR  
action[16,11] = ERR  
action[16,12] = ERR

action[16,13] = ERR  
action[16,14] = ERR  
action[17,0] = ERR  
action[17,1] = ERR  
action[17,2] = ERR  
action[17,3] = R24  
action[17,4] = ERR  
action[17,5] = ERR  
action[17,6] = ERR  
action[17,7] = R24  
action[17,8] = ERR  
action[17,9] = ERR  
action[17,10] = R24  
action[17,11] = R24  
action[17,12] = R24  
action[17,13] = R24  
action[17,14] = ERR  
action[18,0] = ERR  
action[18,1] = R6  
action[18,2] = ERR  
action[18,3] = ERR  
action[18,4] = ERR  
action[18,5] = ERR  
action[18,6] = ERR  
action[18,7] = S23  
action[18,8] = ERR  
action[18,9] = ERR  
action[18,10] = ERR  
action[18,11] = ERR  
action[18,12] = ERR  
action[18,13] = ERR  
action[18,14] = ERR  
action[19,0] = ERR  
action[19,1] = S24  
action[19,2] = ERR  
action[19,3] = ERR  
action[19,4] = ERR  
action[19,5] = ERR

---

action[19,6] = ERR  
action[19,7] = ERR  
action[19,8] = ERR  
action[19,9] = ERR  
action[19,10] = ERR  
action[19,11] = ERR  
action[19,12] = ERR  
action[19,13] = ERR  
action[19,14] = ERR  
action[20,0] = ERR  
action[20,1] = R5  
action[20,2] = ERR  
action[20,3] = ERR  
action[20,4] = ERR  
action[20,5] = ERR  
action[20,6] = ERR  
action[20,7] = R5  
action[20,8] = ERR  
action[20,9] = ERR  
action[20,10] = ERR  
action[20,11] = ERR  
action[20,12] = ERR  
action[20,13] = ERR  
action[20,14] = ERR  
action[21,0] = L45  
action[21,1] = ERR  
action[21,2] = S5  
action[21,3] = ERR  
action[21,4] = R21  
action[21,5] = ERR  
action[21,6] = S6  
action[21,7] = ERR  
action[21,8] = ERR  
action[21,9] = ERR  
action[21,10] = ERR  
action[21,11] = ERR  
action[21,12] = R29  
action[21,13] = ERR

action[21,14] = R21  
action[22,0] = R30  
action[22,1] = ERR  
action[22,2] = ERR  
action[22,3] = ERR  
action[22,4] = ERR  
action[22,5] = ERR  
action[22,6] = ERR  
action[22,7] = ERR  
action[22,8] = ERR  
action[22,9] = ERR  
action[22,10] = ERR  
action[22,11] = ERR  
action[22,12] = ERR  
action[22,13] = ERR  
action[22,14] = ERR  
action[23,0] = ERR  
action[23,1] = ERR  
action[23,2] = ERR  
action[23,3] = ERR  
action[23,4] = ERR  
action[23,5] = S33  
action[23,6] = ERR  
action[23,7] = ERR  
action[23,8] = ERR  
action[23,9] = ERR  
action[23,10] = ERR  
action[23,11] = ERR  
action[23,12] = ERR  
action[23,13] = ERR  
action[23,14] = ERR  
action[24,0] = ERR  
action[24,1] = ERR  
action[24,2] = ERR  
action[24,3] = R1  
action[24,4] = ERR  
action[24,5] = ERR  
action[24,6] = ERR

---

action[24,7] = R1  
action[24,8] = ERR  
action[24,9] = ERR  
action[24,10] = R1  
action[24,11] = R1  
action[24,12] = R1  
action[24,13] = R1  
action[24,14] = ERR  
action[25,0] = ERR  
action[25,1] = ERR  
action[25,2] = ERR  
action[25,3] = ERR  
action[25,4] = ERR  
action[25,5] = ERR  
action[25,6] = ERR  
action[25,7] = R9  
action[25,8] = ERR  
action[25,9] = ERR  
action[25,10] = S34  
action[25,11] = ERR  
action[25,12] = R9  
action[25,13] = R9  
action[25,14] = ERR  
action[26,0] = ERR  
action[26,1] = ERR  
action[26,2] = ERR  
action[26,3] = ERR  
action[26,4] = ERR  
action[26,5] = ERR  
action[26,6] = ERR  
action[26,7] = R11  
action[26,8] = ERR  
action[26,9] = ERR  
action[26,10] = ERR  
action[26,11] = ERR  
action[26,12] = R11  
action[26,13] = R11  
action[26,14] = ERR

action[27,0] = ERR  
action[27,1] = ERR  
action[27,2] = ERR  
action[27,3] = ERR  
action[27,4] = ERR  
action[27,5] = ERR  
action[27,6] = ERR  
action[27,7] = R16  
action[27,8] = ERR  
action[27,9] = ERR  
action[27,10] = ERR  
action[27,11] = ERR  
action[27,12] = R16  
action[27,13] = S35  
action[27,14] = ERR  
action[28,0] = ERR  
action[28,1] = ERR  
action[28,2] = ERR  
action[28,3] = ERR  
action[28,4] = ERR  
action[28,5] = ERR  
action[28,6] = ERR  
action[28,7] = R27  
action[28,8] = ERR  
action[28,9] = ERR  
action[28,10] = R27  
action[28,11] = R27  
action[28,12] = R27  
action[28,13] = R27  
action[28,14] = ERR  
action[29,0] = ERR  
action[29,1] = ERR  
action[29,2] = ERR  
action[29,3] = ERR  
action[29,4] = ERR  
action[29,5] = ERR  
action[29,6] = ERR  
action[29,7] = R18

---

action[29,8] = ERR  
action[29,9] = ERR  
action[29,10] = ERR  
action[29,11] = ERR  
action[29,12] = R18  
action[29,13] = ERR  
action[29,14] = ERR  
action[30,0] = ERR  
action[30,1] = ERR  
action[30,2] = ERR  
action[30,3] = ERR  
action[30,4] = ERR  
action[30,5] = ERR  
action[30,6] = ERR  
action[30,7] = S36  
action[30,8] = ERR  
action[30,9] = ERR  
action[30,10] = ERR  
action[30,11] = ERR  
action[30,12] = R28  
action[30,13] = ERR  
action[30,14] = ERR  
action[31,0] = ERR  
action[31,1] = ERR  
action[31,2] = ERR  
action[31,3] = ERR  
action[31,4] = ERR  
action[31,5] = ERR  
action[31,6] = ERR  
action[31,7] = R2  
action[31,8] = ERR  
action[31,9] = ERR  
action[31,10] = R2  
action[31,11] = S37  
action[31,12] = R2  
action[31,13] = R2  
action[31,14] = ERR  
action[32,0] = ERR

action[32,1] = ERR  
action[32,2] = ERR  
action[32,3] = ERR  
action[32,4] = ERR  
action[32,5] = ERR  
action[32,6] = ERR  
action[32,7] = ERR  
action[32,8] = ERR  
action[32,9] = ERR  
action[32,10] = ERR  
action[32,11] = ERR  
action[32,12] = S38  
action[32,13] = ERR  
action[32,14] = ERR  
action[33,0] = ERR  
action[33,1] = R4  
action[33,2] = ERR  
action[33,3] = ERR  
action[33,4] = ERR  
action[33,5] = ERR  
action[33,6] = ERR  
action[33,7] = R4  
action[33,8] = ERR  
action[33,9] = ERR  
action[33,10] = ERR  
action[33,11] = ERR  
action[33,12] = ERR  
action[33,13] = ERR  
action[33,14] = ERR  
action[34,0] = L46  
action[34,1] = ERR  
action[34,2] = S5  
action[34,3] = ERR  
action[34,4] = R21  
action[34,5] = ERR  
action[34,6] = S6  
action[34,7] = ERR  
action[34,8] = ERR



---

action[34,9] = ERR  
action[34,10] = ERR  
action[34,11] = ERR  
action[34,12] = ERR  
action[34,13] = ERR  
action[34,14] = R21  
action[35,0] = L47  
action[35,1] = ERR  
action[35,2] = S5  
action[35,3] = ERR  
action[35,4] = R21  
action[35,5] = ERR  
action[35,6] = S6  
action[35,7] = ERR  
action[35,8] = ERR  
action[35,9] = ERR  
action[35,10] = ERR  
action[35,11] = ERR  
action[35,12] = ERR  
action[35,13] = ERR  
action[35,14] = R21  
action[36,0] = L48  
action[36,1] = ERR  
action[36,2] = S5  
action[36,3] = ERR  
action[36,4] = R21  
action[36,5] = ERR  
action[36,6] = S6  
action[36,7] = ERR  
action[36,8] = ERR  
action[36,9] = ERR  
action[36,10] = ERR  
action[36,11] = ERR  
action[36,12] = ERR  
action[36,13] = ERR  
action[36,14] = R21  
action[37,0] = ERR  
action[37,1] = ERR

action[37,2] = ERR  
action[37,3] = ERR  
action[37,4] = ERR  
action[37,5] = ERR  
action[37,6] = ERR  
action[37,7] = R26  
action[37,8] = ERR  
action[37,9] = ERR  
action[37,10] = R26  
action[37,11] = R26  
action[37,12] = R26  
action[37,13] = R26  
action[37,14] = ERR  
action[38,0] = ERR  
action[38,1] = ERR  
action[38,2] = ERR  
action[38,3] = R3  
action[38,4] = ERR  
action[38,5] = ERR  
action[38,6] = ERR  
action[38,7] = R3  
action[38,8] = ERR  
action[38,9] = ERR  
action[38,10] = R3  
action[38,11] = R3  
action[38,12] = R3  
action[38,13] = R3  
action[38,14] = ERR  
action[39,0] = ERR  
action[39,1] = ERR  
action[39,2] = ERR  
action[39,3] = ERR  
action[39,4] = ERR  
action[39,5] = ERR  
action[39,6] = ERR  
action[39,7] = R8  
action[39,8] = ERR  
action[39,9] = ERR

---

action[39,10] = ERR  
action[39,11] = ERR  
action[39,12] = R8  
action[39,13] = R8  
action[39,14] = ERR  
action[40,0] = ERR  
action[40,1] = ERR  
action[40,2] = ERR  
action[40,3] = ERR  
action[40,4] = ERR  
action[40,5] = ERR  
action[40,6] = ERR  
action[40,7] = R10  
action[40,8] = ERR  
action[40,9] = ERR  
action[40,10] = ERR  
action[40,11] = ERR  
action[40,12] = R10  
action[40,13] = R10  
action[40,14] = ERR  
action[41,0] = ERR  
action[41,1] = ERR  
action[41,2] = ERR  
action[41,3] = ERR  
action[41,4] = ERR  
action[41,5] = ERR  
action[41,6] = ERR  
action[41,7] = R17  
action[41,8] = ERR  
action[41,9] = ERR  
action[41,10] = ERR  
action[41,11] = ERR  
action[41,12] = R17  
action[41,13] = ERR  
action[41,14] = ERR  
action[42,0] = ERR  
action[42,1] = ERR  
action[42,2] = ERR

action[42,3] = S2  
action[42,4] = ERR  
action[42,5] = ERR  
action[42,6] = ERR  
action[42,7] = ERR  
action[42,8] = ERR  
action[42,9] = R31  
action[42,10] = ERR  
action[42,11] = ERR  
action[42,12] = ERR  
action[42,13] = ERR  
action[42,14] = R21  
action[43,0] = L44  
action[43,1] = ERR  
action[43,2] = ERR  
action[43,3] = ERR  
action[43,4] = ERR  
action[43,5] = ERR  
action[43,6] = ERR  
action[43,7] = ERR  
action[43,8] = ERR  
action[43,9] = ERR  
action[43,10] = ERR  
action[43,11] = ERR  
action[43,12] = ERR  
action[43,13] = ERR  
action[43,14] = ERR  
action[44,0] = ERR  
action[44,1] = ERR  
action[44,2] = ERR  
action[44,3] = R0  
action[44,4] = ERR  
action[44,5] = ERR  
action[44,6] = ERR  
action[44,7] = R0  
action[44,8] = ERR  
action[44,9] = S22  
action[44,10] = R0

---

action[44,11] = R0  
action[44,12] = R0  
action[44,13] = R0  
action[44,14] = ERR  
action[45,0] = ERR  
action[45,1] = ERR  
action[45,2] = ERR  
action[45,3] = ERR  
action[45,4] = ERR  
action[45,5] = ERR  
action[45,6] = ERR  
action[45,7] = S2  
action[45,8] = ERR  
action[45,9] = R31  
action[45,10] = S2  
action[45,11] = S2  
action[45,12] = S2  
action[45,13] = S2  
action[45,14] = R21  
action[46,0] = ERR  
action[46,1] = ERR  
action[46,2] = ERR  
action[46,3] = ERR  
action[46,4] = ERR  
action[46,5] = ERR  
action[46,6] = ERR  
action[46,7] = S2  
action[46,8] = ERR  
action[46,9] = R31  
action[46,10] = S2  
action[46,11] = S2  
action[46,12] = S2  
action[46,13] = S2  
action[46,14] = R21  
action[47,0] = ERR  
action[47,1] = ERR  
action[47,2] = ERR  
action[47,3] = ERR

action[47,4] = ERR  
action[47,5] = ERR  
action[47,6] = ERR  
action[47,7] = S2  
action[47,8] = ERR  
action[47,9] = R31  
action[47,10] = S2  
action[47,11] = S2  
action[47,12] = S2  
action[47,13] = S2  
action[47,14] = R21  
action[48,0] = ERR  
action[48,1] = ERR  
action[48,2] = ERR  
action[48,3] = ERR  
action[48,4] = ERR  
action[48,5] = ERR  
action[48,6] = ERR  
action[48,7] = S2  
action[48,8] = ERR  
action[48,9] = R31  
action[48,10] = S2  
action[48,11] = S2  
action[48,12] = S2  
action[48,13] = S2  
action[48,14] = R21

Goto table:

goto[0,0] = 1  
goto[0,1] = 3  
goto[0,2] = NIL  
goto[0,3] = 4  
goto[0,4] = NIL  
goto[0,5] = NIL  
goto[0,6] = NIL  
goto[0,7] = NIL  
goto[0,8] = NIL

---

```
goto[0,9] = 7
goto[0,10] = NIL
goto[0,11] = NIL
goto[0,12] = 8
goto[0,13] = NIL
goto[0,14] = 9
goto[0,15] = NIL
goto[0,16] = NIL
goto[0,17] = 10
goto[1,0] = NIL
goto[1,1] = NIL
goto[1,2] = NIL
goto[1,3] = NIL
goto[1,4] = NIL
goto[1,5] = NIL
goto[1,6] = NIL
goto[1,7] = NIL
goto[1,8] = NIL
goto[1,9] = NIL
goto[1,10] = NIL
goto[1,11] = NIL
goto[1,12] = NIL
goto[1,13] = NIL
goto[1,14] = NIL
goto[1,15] = NIL
goto[1,16] = NIL
goto[1,17] = NIL
goto[2,0] = NIL
goto[2,1] = NIL
goto[2,2] = NIL
goto[2,3] = NIL
goto[2,4] = NIL
goto[2,5] = NIL
goto[2,6] = NIL
goto[2,7] = NIL
goto[2,8] = NIL
goto[2,9] = NIL
goto[2,10] = NIL
```

goto[2,11] = NIL  
goto[2,12] = NIL  
goto[2,13] = NIL  
goto[2,14] = NIL  
goto[2,15] = NIL  
goto[2,16] = NIL  
goto[2,17] = NIL  
goto[3,0] = NIL  
goto[3,1] = NIL  
goto[3,2] = NIL  
goto[3,3] = NIL  
goto[3,4] = NIL  
goto[3,5] = NIL  
goto[3,6] = NIL  
goto[3,7] = NIL  
goto[3,8] = NIL  
goto[3,9] = NIL  
goto[3,10] = NIL  
goto[3,11] = NIL  
goto[3,12] = NIL  
goto[3,13] = NIL  
goto[3,14] = NIL  
goto[3,15] = NIL  
goto[3,16] = NIL  
goto[3,17] = NIL  
goto[4,0] = NIL  
goto[4,1] = NIL  
goto[4,2] = NIL  
goto[4,3] = NIL  
goto[4,4] = NIL  
goto[4,5] = NIL  
goto[4,6] = NIL  
goto[4,7] = NIL  
goto[4,8] = NIL  
goto[4,9] = NIL  
goto[4,10] = NIL  
goto[4,11] = NIL  
goto[4,12] = NIL



---

goto[4,13] = NIL  
goto[4,14] = NIL  
goto[4,15] = NIL  
goto[4,16] = NIL  
goto[4,17] = NIL  
goto[5,0] = NIL  
goto[5,1] = NIL  
goto[5,2] = NIL  
goto[5,3] = NIL  
goto[5,4] = NIL  
goto[5,5] = NIL  
goto[5,6] = NIL  
goto[5,7] = NIL  
goto[5,8] = NIL  
goto[5,9] = NIL  
goto[5,10] = NIL  
goto[5,11] = NIL  
goto[5,12] = NIL  
goto[5,13] = NIL  
goto[5,14] = NIL  
goto[5,15] = NIL  
goto[5,16] = NIL  
goto[5,17] = NIL  
goto[6,0] = NIL  
goto[6,1] = NIL  
goto[6,2] = NIL  
goto[6,3] = NIL  
goto[6,4] = NIL  
goto[6,5] = NIL  
goto[6,6] = NIL  
goto[6,7] = NIL  
goto[6,8] = NIL  
goto[6,9] = NIL  
goto[6,10] = NIL  
goto[6,11] = NIL  
goto[6,12] = NIL  
goto[6,13] = NIL  
goto[6,14] = NIL

goto[6,15] = NIL  
goto[6,16] = NIL  
goto[6,17] = NIL  
goto[7,0] = NIL  
goto[7,1] = NIL  
goto[7,2] = NIL  
goto[7,3] = NIL  
goto[7,4] = NIL  
goto[7,5] = NIL  
goto[7,6] = NIL  
goto[7,7] = NIL  
goto[7,8] = NIL  
goto[7,9] = NIL  
goto[7,10] = NIL  
goto[7,11] = NIL  
goto[7,12] = NIL  
goto[7,13] = NIL  
goto[7,14] = NIL  
goto[7,15] = NIL  
goto[7,16] = NIL  
goto[7,17] = NIL  
goto[8,0] = NIL  
goto[8,1] = NIL  
goto[8,2] = NIL  
goto[8,3] = NIL  
goto[8,4] = NIL  
goto[8,5] = NIL  
goto[8,6] = NIL  
goto[8,7] = NIL  
goto[8,8] = NIL  
goto[8,9] = NIL  
goto[8,10] = NIL  
goto[8,11] = NIL  
goto[8,12] = NIL  
goto[8,13] = 15  
goto[8,14] = NIL  
goto[8,15] = NIL  
goto[8,16] = NIL

---

```
goto[8,17] = NIL
goto[9,0] = NIL
goto[9,1] = NIL
goto[9,2] = NIL
goto[9,3] = NIL
goto[9,4] = NIL
goto[9,5] = NIL
goto[9,6] = NIL
goto[9,7] = NIL
goto[9,8] = NIL
goto[9,9] = NIL
goto[9,10] = NIL
goto[9,11] = NIL
goto[9,12] = NIL
goto[9,13] = NIL
goto[9,14] = NIL
goto[9,15] = NIL
goto[9,16] = NIL
goto[9,17] = NIL
goto[10,0] = NIL
goto[10,1] = NIL
goto[10,2] = NIL
goto[10,3] = NIL
goto[10,4] = NIL
goto[10,5] = NIL
goto[10,6] = NIL
goto[10,7] = NIL
goto[10,8] = NIL
goto[10,9] = NIL
goto[10,10] = NIL
goto[10,11] = NIL
goto[10,12] = NIL
goto[10,13] = NIL
goto[10,14] = NIL
goto[10,15] = NIL
goto[10,16] = NIL
goto[10,17] = NIL
goto[11,0] = NIL
```

goto[11,1] = NIL  
goto[11,2] = NIL  
goto[11,3] = NIL  
goto[11,4] = NIL  
goto[11,5] = NIL  
goto[11,6] = NIL  
goto[11,7] = NIL  
goto[11,8] = NIL  
goto[11,9] = NIL  
goto[11,10] = NIL  
goto[11,11] = NIL  
goto[11,12] = NIL  
goto[11,13] = NIL  
goto[11,14] = NIL  
goto[11,15] = NIL  
goto[11,16] = NIL  
goto[11,17] = NIL  
goto[12,0] = NIL  
goto[12,1] = NIL  
goto[12,2] = NIL  
goto[12,3] = NIL  
goto[12,4] = NIL  
goto[12,5] = NIL  
goto[12,6] = NIL  
goto[12,7] = NIL  
goto[12,8] = NIL  
goto[12,9] = NIL  
goto[12,10] = NIL  
goto[12,11] = NIL  
goto[12,12] = NIL  
goto[12,13] = NIL  
goto[12,14] = NIL  
goto[12,15] = NIL  
goto[12,16] = NIL  
goto[12,17] = NIL  
goto[13,0] = NIL  
goto[13,1] = NIL  
goto[13,2] = NIL

---

```
goto[13,3] = NIL
goto[13,4] = NIL
goto[13,5] = NIL
goto[13,6] = NIL
goto[13,7] = NIL
goto[13,8] = NIL
goto[13,9] = NIL
goto[13,10] = NIL
goto[13,11] = NIL
goto[13,12] = NIL
goto[13,13] = NIL
goto[13,14] = NIL
goto[13,15] = NIL
goto[13,16] = NIL
goto[13,17] = NIL
goto[14,0] = NIL
goto[14,1] = NIL
goto[14,2] = NIL
goto[14,3] = NIL
goto[14,4] = 18
goto[14,5] = 19
goto[14,6] = NIL
goto[14,7] = NIL
goto[14,8] = NIL
goto[14,9] = NIL
goto[14,10] = NIL
goto[14,11] = NIL
goto[14,12] = NIL
goto[14,13] = NIL
goto[14,14] = NIL
goto[14,15] = NIL
goto[14,16] = NIL
goto[14,17] = NIL
goto[15,0] = NIL
goto[15,1] = NIL
goto[15,2] = NIL
goto[15,3] = NIL
goto[15,4] = NIL
```

goto[15,5] = NIL  
goto[15,6] = NIL  
goto[15,7] = NIL  
goto[15,8] = NIL  
goto[15,9] = NIL  
goto[15,10] = NIL  
goto[15,11] = NIL  
goto[15,12] = NIL  
goto[15,13] = NIL  
goto[15,14] = NIL  
goto[15,15] = NIL  
goto[15,16] = NIL  
goto[15,17] = NIL  
goto[16,0] = NIL  
goto[16,1] = NIL  
goto[16,2] = NIL  
goto[16,3] = NIL  
goto[16,4] = NIL  
goto[16,5] = NIL  
goto[16,6] = NIL  
goto[16,7] = NIL  
goto[16,8] = NIL  
goto[16,9] = NIL  
goto[16,10] = NIL  
goto[16,11] = NIL  
goto[16,12] = NIL  
goto[16,13] = NIL  
goto[16,14] = NIL  
goto[16,15] = NIL  
goto[16,16] = NIL  
goto[16,17] = NIL  
goto[17,0] = NIL  
goto[17,1] = NIL  
goto[17,2] = NIL  
goto[17,3] = NIL  
goto[17,4] = NIL  
goto[17,5] = NIL  
goto[17,6] = NIL

---

```
goto[17,7] = NIL
goto[17,8] = NIL
goto[17,9] = NIL
goto[17,10] = NIL
goto[17,11] = NIL
goto[17,12] = NIL
goto[17,13] = NIL
goto[17,14] = NIL
goto[17,15] = NIL
goto[17,16] = NIL
goto[17,17] = NIL
goto[18,0] = NIL
goto[18,1] = NIL
goto[18,2] = NIL
goto[18,3] = NIL
goto[18,4] = NIL
goto[18,5] = NIL
goto[18,6] = NIL
goto[18,7] = NIL
goto[18,8] = NIL
goto[18,9] = NIL
goto[18,10] = NIL
goto[18,11] = NIL
goto[18,12] = NIL
goto[18,13] = NIL
goto[18,14] = NIL
goto[18,15] = NIL
goto[18,16] = NIL
goto[18,17] = NIL
goto[19,0] = NIL
goto[19,1] = NIL
goto[19,2] = NIL
goto[19,3] = NIL
goto[19,4] = NIL
goto[19,5] = NIL
goto[19,6] = NIL
goto[19,7] = NIL
goto[19,8] = NIL
```

goto[19,9] = NIL  
goto[19,10] = NIL  
goto[19,11] = NIL  
goto[19,12] = NIL  
goto[19,13] = NIL  
goto[19,14] = NIL  
goto[19,15] = NIL  
goto[19,16] = NIL  
goto[19,17] = NIL  
goto[20,0] = NIL  
goto[20,1] = NIL  
goto[20,2] = NIL  
goto[20,3] = NIL  
goto[20,4] = NIL  
goto[20,5] = NIL  
goto[20,6] = NIL  
goto[20,7] = NIL  
goto[20,8] = NIL  
goto[20,9] = NIL  
goto[20,10] = NIL  
goto[20,11] = NIL  
goto[20,12] = NIL  
goto[20,13] = NIL  
goto[20,14] = NIL  
goto[20,15] = NIL  
goto[20,16] = NIL  
goto[20,17] = NIL  
goto[21,0] = 1  
goto[21,1] = 3  
goto[21,2] = 25  
goto[21,3] = 4  
goto[21,4] = NIL  
goto[21,5] = NIL  
goto[21,6] = 26  
goto[21,7] = 27  
goto[21,8] = NIL  
goto[21,9] = 28  
goto[21,10] = 29



---

```
goto[21,11] = 30
goto[21,12] = 8
goto[21,13] = NIL
goto[21,14] = 9
goto[21,15] = 31
goto[21,16] = 32
goto[21,17] = 10
goto[22,0] = NIL
goto[22,1] = NIL
goto[22,2] = NIL
goto[22,3] = NIL
goto[22,4] = NIL
goto[22,5] = NIL
goto[22,6] = NIL
goto[22,7] = NIL
goto[22,8] = NIL
goto[22,9] = NIL
goto[22,10] = NIL
goto[22,11] = NIL
goto[22,12] = NIL
goto[22,13] = NIL
goto[22,14] = NIL
goto[22,15] = NIL
goto[22,16] = NIL
goto[22,17] = NIL
goto[23,0] = NIL
goto[23,1] = NIL
goto[23,2] = NIL
goto[23,3] = NIL
goto[23,4] = NIL
goto[23,5] = NIL
goto[23,6] = NIL
goto[23,7] = NIL
goto[23,8] = NIL
goto[23,9] = NIL
goto[23,10] = NIL
goto[23,11] = NIL
goto[23,12] = NIL
```

goto[23,13] = NIL  
goto[23,14] = NIL  
goto[23,15] = NIL  
goto[23,16] = NIL  
goto[23,17] = NIL  
goto[24,0] = NIL  
goto[24,1] = NIL  
goto[24,2] = NIL  
goto[24,3] = NIL  
goto[24,4] = NIL  
goto[24,5] = NIL  
goto[24,6] = NIL  
goto[24,7] = NIL  
goto[24,8] = NIL  
goto[24,9] = NIL  
goto[24,10] = NIL  
goto[24,11] = NIL  
goto[24,12] = NIL  
goto[24,13] = NIL  
goto[24,14] = NIL  
goto[24,15] = NIL  
goto[24,16] = NIL  
goto[24,17] = NIL  
goto[25,0] = NIL  
goto[25,1] = NIL  
goto[25,2] = NIL  
goto[25,3] = NIL  
goto[25,4] = NIL  
goto[25,5] = NIL  
goto[25,6] = NIL  
goto[25,7] = NIL  
goto[25,8] = NIL  
goto[25,9] = NIL  
goto[25,10] = NIL  
goto[25,11] = NIL  
goto[25,12] = NIL  
goto[25,13] = NIL  
goto[25,14] = NIL

---

```
goto [25,15] = NIL
goto [25,16] = NIL
goto [25,17] = NIL
goto [26,0] = NIL
goto [26,1] = NIL
goto [26,2] = NIL
goto [26,3] = NIL
goto [26,4] = NIL
goto [26,5] = NIL
goto [26,6] = NIL
goto [26,7] = NIL
goto [26,8] = NIL
goto [26,9] = NIL
goto [26,10] = NIL
goto [26,11] = NIL
goto [26,12] = NIL
goto [26,13] = NIL
goto [26,14] = NIL
goto [26,15] = NIL
goto [26,16] = NIL
goto [26,17] = NIL
goto [27,0] = NIL
goto [27,1] = NIL
goto [27,2] = NIL
goto [27,3] = NIL
goto [27,4] = NIL
goto [27,5] = NIL
goto [27,6] = NIL
goto [27,7] = NIL
goto [27,8] = NIL
goto [27,9] = NIL
goto [27,10] = NIL
goto [27,11] = NIL
goto [27,12] = NIL
goto [27,13] = NIL
goto [27,14] = NIL
goto [27,15] = NIL
goto [27,16] = NIL
```

goto[27,17] = NIL  
goto[28,0] = NIL  
goto[28,1] = NIL  
goto[28,2] = NIL  
goto[28,3] = NIL  
goto[28,4] = NIL  
goto[28,5] = NIL  
goto[28,6] = NIL  
goto[28,7] = NIL  
goto[28,8] = NIL  
goto[28,9] = NIL  
goto[28,10] = NIL  
goto[28,11] = NIL  
goto[28,12] = NIL  
goto[28,13] = NIL  
goto[28,14] = NIL  
goto[28,15] = NIL  
goto[28,16] = NIL  
goto[28,17] = NIL  
goto[29,0] = NIL  
goto[29,1] = NIL  
goto[29,2] = NIL  
goto[29,3] = NIL  
goto[29,4] = NIL  
goto[29,5] = NIL  
goto[29,6] = NIL  
goto[29,7] = NIL  
goto[29,8] = NIL  
goto[29,9] = NIL  
goto[29,10] = NIL  
goto[29,11] = NIL  
goto[29,12] = NIL  
goto[29,13] = NIL  
goto[29,14] = NIL  
goto[29,15] = NIL  
goto[29,16] = NIL  
goto[29,17] = NIL  
goto[30,0] = NIL

---

```
goto [30,1] = NIL
goto [30,2] = NIL
goto [30,3] = NIL
goto [30,4] = NIL
goto [30,5] = NIL
goto [30,6] = NIL
goto [30,7] = NIL
goto [30,8] = NIL
goto [30,9] = NIL
goto [30,10] = NIL
goto [30,11] = NIL
goto [30,12] = NIL
goto [30,13] = NIL
goto [30,14] = NIL
goto [30,15] = NIL
goto [30,16] = NIL
goto [30,17] = NIL
goto [31,0] = NIL
goto [31,1] = NIL
goto [31,2] = NIL
goto [31,3] = NIL
goto [31,4] = NIL
goto [31,5] = NIL
goto [31,6] = NIL
goto [31,7] = NIL
goto [31,8] = NIL
goto [31,9] = NIL
goto [31,10] = NIL
goto [31,11] = NIL
goto [31,12] = NIL
goto [31,13] = NIL
goto [31,14] = NIL
goto [31,15] = NIL
goto [31,16] = NIL
goto [31,17] = NIL
goto [32,0] = NIL
goto [32,1] = NIL
goto [32,2] = NIL
```

goto[32,3] = NIL  
goto[32,4] = NIL  
goto[32,5] = NIL  
goto[32,6] = NIL  
goto[32,7] = NIL  
goto[32,8] = NIL  
goto[32,9] = NIL  
goto[32,10] = NIL  
goto[32,11] = NIL  
goto[32,12] = NIL  
goto[32,13] = NIL  
goto[32,14] = NIL  
goto[32,15] = NIL  
goto[32,16] = NIL  
goto[32,17] = NIL  
goto[33,0] = NIL  
goto[33,1] = NIL  
goto[33,2] = NIL  
goto[33,3] = NIL  
goto[33,4] = NIL  
goto[33,5] = NIL  
goto[33,6] = NIL  
goto[33,7] = NIL  
goto[33,8] = NIL  
goto[33,9] = NIL  
goto[33,10] = NIL  
goto[33,11] = NIL  
goto[33,12] = NIL  
goto[33,13] = NIL  
goto[33,14] = NIL  
goto[33,15] = NIL  
goto[33,16] = NIL  
goto[33,17] = NIL  
goto[34,0] = 1  
goto[34,1] = 3  
goto[34,2] = 25  
goto[34,3] = 4  
goto[34,4] = NIL

---

```
goto [34,5] = NIL
goto [34,6] = 39
goto [34,7] = NIL
goto [34,8] = NIL
goto [34,9] = 28
goto [34,10] = NIL
goto [34,11] = NIL
goto [34,12] = 8
goto [34,13] = NIL
goto [34,14] = 9
goto [34,15] = 31
goto [34,16] = NIL
goto [34,17] = 10
goto [35,0] = 1
goto [35,1] = 3
goto [35,2] = 25
goto [35,3] = 4
goto [35,4] = NIL
goto [35,5] = NIL
goto [35,6] = 40
goto [35,7] = NIL
goto [35,8] = NIL
goto [35,9] = 28
goto [35,10] = NIL
goto [35,11] = NIL
goto [35,12] = 8
goto [35,13] = NIL
goto [35,14] = 9
goto [35,15] = 31
goto [35,16] = NIL
goto [35,17] = 10
goto [36,0] = 1
goto [36,1] = 3
goto [36,2] = 25
goto [36,3] = 4
goto [36,4] = NIL
goto [36,5] = NIL
goto [36,6] = 26
```

goto[36,7] = 27  
goto[36,8] = NIL  
goto[36,9] = 28  
goto[36,10] = 41  
goto[36,11] = NIL  
goto[36,12] = 8  
goto[36,13] = NIL  
goto[36,14] = 9  
goto[36,15] = 31  
goto[36,16] = NIL  
goto[36,17] = 10  
goto[37,0] = NIL  
goto[37,1] = NIL  
goto[37,2] = NIL  
goto[37,3] = NIL  
goto[37,4] = NIL  
goto[37,5] = NIL  
goto[37,6] = NIL  
goto[37,7] = NIL  
goto[37,8] = NIL  
goto[37,9] = NIL  
goto[37,10] = NIL  
goto[37,11] = NIL  
goto[37,12] = NIL  
goto[37,13] = NIL  
goto[37,14] = NIL  
goto[37,15] = NIL  
goto[37,16] = NIL  
goto[37,17] = NIL  
goto[38,0] = NIL  
goto[38,1] = NIL  
goto[38,2] = NIL  
goto[38,3] = NIL  
goto[38,4] = NIL  
goto[38,5] = NIL  
goto[38,6] = NIL  
goto[38,7] = NIL  
goto[38,8] = NIL



---

```
goto [38,9] = NIL
goto [38,10] = NIL
goto [38,11] = NIL
goto [38,12] = NIL
goto [38,13] = NIL
goto [38,14] = NIL
goto [38,15] = NIL
goto [38,16] = NIL
goto [38,17] = NIL
goto [39,0] = NIL
goto [39,1] = NIL
goto [39,2] = NIL
goto [39,3] = NIL
goto [39,4] = NIL
goto [39,5] = NIL
goto [39,6] = NIL
goto [39,7] = NIL
goto [39,8] = NIL
goto [39,9] = NIL
goto [39,10] = NIL
goto [39,11] = NIL
goto [39,12] = NIL
goto [39,13] = NIL
goto [39,14] = NIL
goto [39,15] = NIL
goto [39,16] = NIL
goto [39,17] = NIL
goto [40,0] = NIL
goto [40,1] = NIL
goto [40,2] = NIL
goto [40,3] = NIL
goto [40,4] = NIL
goto [40,5] = NIL
goto [40,6] = NIL
goto [40,7] = NIL
goto [40,8] = NIL
goto [40,9] = NIL
goto [40,10] = NIL
```

goto[40,11] = NIL  
goto[40,12] = NIL  
goto[40,13] = NIL  
goto[40,14] = NIL  
goto[40,15] = NIL  
goto[40,16] = NIL  
goto[40,17] = NIL  
goto[41,0] = NIL  
goto[41,1] = NIL  
goto[41,2] = NIL  
goto[41,3] = NIL  
goto[41,4] = NIL  
goto[41,5] = NIL  
goto[41,6] = NIL  
goto[41,7] = NIL  
goto[41,8] = NIL  
goto[41,9] = NIL  
goto[41,10] = NIL  
goto[41,11] = NIL  
goto[41,12] = NIL  
goto[41,13] = NIL  
goto[41,14] = NIL  
goto[41,15] = NIL  
goto[41,16] = NIL  
goto[41,17] = NIL

# Apêndice C

## Gramática de Máquina ( $k_{max} = 2$ )

terminal

BOOLEAN\_LIT,  
CHAR\_LIT,  
INTEGER\_LIT,  
FLOAT\_LIT,  
STRING\_LIT,  
LIST\_LIT,  
AGENT\_LIST,  
ABSTRACTIONS,  
ACTION,  
AGENT1,  
AGENT2,  
ALGEBRA,  
AND,  
AS,  
BEGIN,  
BOOL,  
CASE,  
CHAR,  
CHOOSE,  
CREATE,  
DEFAULT,  
DERIVED,  
DESTROY,  
DISPATCH,  
DO,

DYNAMIC,  
ELSE,  
ELSEIF,  
END,  
ENUM,  
EXTERNAL,  
FILE,  
FORALL,  
IF,  
IMPORT,  
INCLUDE,  
IN,  
INITIAL,  
INOUT,  
INPUT,  
INT,  
INTERFACE,  
INVARIANT,  
IS,  
LET,  
LIST1,  
LIST2,  
LOOP,  
MACHINA,  
MODULE,  
NOT,  
OF,  
OR,  
OTHERWISE,  
OUT,  
OUTPUT,  
PUBLIC,  
REAL,  
RETURN,  
SATISFYING,  
SET1,

---

SET2,  
SHARED,  
STATE,  
STATIC,  
STEP,  
STOP,  
STRING,  
THEN,  
TUPLE,  
TYPE,  
WITH,  
XOR,  
LPAR,  
RPAR,  
DOT,  
COLON,  
COMMA,  
SEMI,  
EQUALS,  
NEQUALS,  
ASSIGN,  
MINUS,  
PLUS,  
TIMES,  
PER,  
PERC,  
LT,  
GT,  
LTE,  
GTE,  
CC,  
DD,  
LBRACE,  
RBRACE,  
LBRACK,  
RBRACK,  
QUESTION\_MARK,  
BAR,

SARROW,  
DARROW,  
TRANSITION,  
ID ;

nonterminal

program\_module,  
module\_body,  
foreign\_part\_list\_opt,  
foreign\_part,  
import\_part,  
importation\_list,  
importation,  
interface\_import\_opt,  
interface\_import,  
interface\_element\_comma\_list,  
interface\_element,  
included\_list\_opt,  
included\_list,  
included\_element\_comma\_list,  
included\_element,  
include\_part,  
inclusion\_list,  
module\_inclusion,  
literal,  
machina\_name,  
module\_name,  
interface\_name,  
agent\_name,  
rule\_name,  
type\_name,  
function\_name,  
variable\_name,  
parameter\_name,  
constant\_name,  
  
algebra\_part\_opt,  
algebra\_part,

---

algebra\_section\_list\_opt,  
algebra\_section\_list,  
algebra\_section,  
expression,  
type\_section,  
function\_section,  
external\_section,  
type\_declaration,  
type\_declarator,  
type\_denotation,  
type\_parameters,  
type\_parameter,  
type\_expression,  
basic\_type,  
function\_declaration,  
class\_modifier\_opt,  
class\_modifier,  
declared\_element,  
declared\_element1,  
declared\_element2,  
function\_parameters,  
function\_parameter,  
external\_declaration,  
external\_function,  
external\_function1,  
external\_function2,  
external\_parameters,  
external\_parameter,  
  
simple\_expression,  
range\_expression,  
adding\_expression,  
multiplying\_expression,  
unary\_expression,  
in\_expression,  
constructing\_expression,  
primary,  
relational\_op,

adding\_op,  
multiplying\_op,  
unary\_op,  
function\_call\_or\_variable\_designator,  
set\_expression,  
node\_expression,  
expression\_comma\_list\_opt,  
expression\_comma\_list,  
node\_element\_list\_opt,  
node\_element\_list,  
node\_element,  
type\_designator,  
type\_identification,  
builtin\_type,  
enumeration\_type,  
constant\_name\_comma\_list,  
compound\_type,  
union\_type,  
tuple\_type,  
tuple\_elements\_opt,  
tuple\_elements,  
tuple\_element,  
list\_type,  
file\_type,  
agent\_type,  
functional\_type,  
set\_type,  
tree\_type,  
tree\_element\_list\_opt,  
tree\_element\_list,  
type\_arguments,  
type\_argument,  
abstraction\_type,  
parameter\_type\_comma\_list,  
parameter\_type,  
passing\_type\_opt,  
passing\_type,  
parameter\_name\_opt,



---

name,  
type\_name\_or\_user\_defined\_type,  
type\_designator\_or\_type\_name,  
call\_or\_variable\_or\_coersion\_or\_type,  
compound\_expression,  
if\_expression,  
else\_exp\_part,  
elseif\_exp\_part\_list\_opt,  
elseif\_exp\_part\_list,  
elseif\_exp\_part,  
case\_expression,  
case\_exp\_clause\_list\_opt,  
case\_exp\_clause\_list,  
case\_exp\_clause,  
default\_case\_exp\_opt,  
with\_expression,  
with\_exp\_clause\_list\_opt,  
with\_exp\_clause\_list,  
with\_exp\_clause,  
with\_exp\_pattern,  
default\_with\_opt,  
pattern,

node\_pattern,  
node\_pattern\_element\_list\_opt,  
node\_pattern\_element\_list,  
node\_pattern\_element,  
empty\_tuple\_or\_list\_pattern,  
tuple\_or\_list\_pattern,  
mixed,  
mix,  
type\_or\_function\_name,  
element,

abstraction\_part\_opt,  
abstraction\_part,  
abstraction\_section\_list\_opt,  
abstraction\_section\_list,

abstraction\_section,  
public\_opt,  
rule\_abstraction,  
action\_declaration,  
rule\_declaration,  
rule\_params\_opt,  
rule\_params,  
rule\_body,  
rule\_parameters,  
rule\_parameter,  
passing\_mode,  
one\_pass\_transition,  
repeating\_transition,  
local\_declarations,  
main\_transition,  
begin\_opt,  
local\_algebra\_section\_list\_opt,  
local\_algebra\_section\_list,  
local\_algebra\_section,  
local\_type,  
local\_function,  
local\_external,  
single\_transition,  
transition\_rule,  
basic\_rule,  
stepwise\_transition,  
step\_block,  
step\_number,  
dynamic\_function\_designator,  
abbreviation\_rule,  
location,  
empty\_rule,  
  
rule,  
conditional\_rule,  
if\_rule,  
case\_rule,  
else\_part,

---

elseif\_term\_list\_opt,  
elseif\_term\_list,  
elseif\_term,  
  
case\_rule\_atom\_list\_opt,  
case\_rule\_atom\_list,  
case\_rule\_atom,  
case\_rule\_otherwise,  
choose\_rule,  
choose\_elements,  
choose\_element,  
choose\_domain,  
universal\_rule,  
agent\_rule,  
with\_rule,  
with\_rule\_atom\_list\_opt,  
with\_rule\_otherwise,  
with\_rule\_atom\_list,  
with\_rule\_atom,  
with\_pattern,  
forall\_rule,  
for\_elements,  
for\_element,  
for\_domain,  
create\_rule,  
dispatch\_rule,  
stop\_rule,  
return\_rule,  
destroy\_rule,  
agent\_declaration\_list,  
agent\_declaration,  
single\_agent\_declaration,  
multiple\_agent\_declaration,  
number\_of\_agents,  
agent\_designator\_list,  
agt\_designator,  
call\_rule,  
call\_or\_update\_rule,

```
specification_unit,  
agent_interface,  
machina_definition,  
initial_state_part_opt,  
invariant_part_opt,  
initial_state_part,  
transition_part_opt,  
transition_part,  
module_transition,  
invariant_part,  
interface_body,  
interface_section,  
exported_type,  
exported_abstraction,  
formal_parameters_opt,  
formal_parameters,  
formal_parameter,  
transmission_mode_opt,  
transmission_mode,  
machina_body,  
machina_directive_list_opt,  
machina_directive_list,  
machina_directive,  
semi_opt  
;
```

```
start with specification_unit ;
```

```
specification_unit ::=  
  program_module |  
  agent_interface |  
  machina_definition ;
```

```
program_module ::=  
  MODULE module_name module_body END |  
  MODULE module_name module_body END module_name ;
```

---

```
module_body ::=
  foreign_part_list_opt algebra_part_opt abstraction_part_opt initial_state_part_opt

foreign_part_list_opt ::=
  foreign_part |
  /* lambda */ ;

foreign_part ::=
  import_part |
  include_part ;

import_part ::=
  IMPORT importation_list |
  IMPORT importation_list SEMI ;

importation_list ::=
  importation |
  importation_list COMMA importation ;

importation ::=
  interface_name interface_import_opt ;

interface_import_opt ::=
  interface_import |
  /* lambda */ ;

interface_import ::=
  LPAR interface_element_comma_list RPAR ;

interface_element_comma_list ::=
  interface_element |
  interface_element_comma_list COMMA interface_element ;

interface_element ::=
  ID ;

include_part ::=
```

```
INCLUDE inclusion_list |
INCLUDE inclusion_list SEMI ;

inclusion_list ::=
  module_inclusion |
  inclusion_list COMMA module_inclusion ;

module_inclusion ::=
  module_name included_list_opt ;

included_list_opt ::=
  included_list |
  /* lambda */ ;

included_list ::=
  LPAR included_element_comma_list RPAR ;

included_element_comma_list ::=
  included_element |
  included_element_comma_list COMMA included_element ;

included_element ::=
  ID ;

initial_state_part_opt ::=
  initial_state_part |
  /* lambda */ ;

initial_state_part ::=
  INITIAL STATE COLON main_transition ;

transition_part_opt ::=
  transition_part |
  /* lambda */ ;

transition_part ::=
  module_transition ;
```

---

```
module_transition ::=
  TRANSITION COLON main_transition ;
```

```
invariant_part_opt ::=
  invariant_part |
  /* lambda */ ;
```

```
invariant_part_opt ::=
  invariant_part |
  /* lambda */ ;
```

```
invariant_part ::=
  INVARIANT COLON expression ;
```

```
literal ::=
  BOOLEAN_LIT |
  CHAR_LIT |
  INTEGER_LIT |
  FLOAT_LIT |
  STRING_LIT |
  LIST_LIT |
  AGENT_LIST ;
```

```
machina_name ::=
  ID ;
```

```
module_name ::=
  ID ;
```

```
interface_name ::=
  ID ;
```

```
agent_name ::=
  ID ;
```

```
rule_name ::=
  ID ;
```

```
type_name ::=
  ID ;

function_name ::=
  ID ;

variable_name ::=
  ID ;

parameter_name ::=
  ID ;

constant_name ::=
  ID ;

algebra_part_opt ::=
  algebra_part |
  /* lambda */ ;

algebra_part ::=
  ALGEBRA COLON algebra_section_list_opt ;

algebra_section_list_opt ::=
  algebra_section_list |
  /* lambda */ ;

algebra_section_list ::=
  algebra_section |
  algebra_section_list algebra_section ;

algebra_section ::=
  type_section |
  function_section |
  external_section ;

type_section ::=
  PUBLIC type_declaration |
```



---

```
type_declaration ;

type_declaration ::=
    TYPE type_declarator EQUALS type_denotation |
    TYPE type_declarator ;

type_declarator ::=
    type_name LPAR type_parameters RPAR |
    type_name ;

type_parameters ::=
    type_parameter |
    type_parameters COMMA type_parameter ;

type_parameter ::=
    parameter_name ;

type_denotation ::=
    type_expression DEFAULT expression |
    type_expression ;

function_section ::=
    PUBLIC function_declaration |
    SHARED function_declaration |
    function_declaration ;

function_declaration ::=
    class_modifier_opt declared_element COLON type_expression ASSIGN expression |
    class_modifier_opt declared_element COLON type_expression ;

class_modifier_opt ::=
    class_modifier |
    /* lambda */ ;

class_modifier ::=
    STATIC |
    DERIVED |
    DYNAMIC ;
```

```
declared_element ::=
  declared_element1 |
  declared_element2 ;

declared_element1 ::=
  function_name |
  declared_element1 COMMA function_name ;

declared_element2 ::=
  function_name LPAR function_parameters RPAR ;

function_parameters ::=
  function_parameter |
  function_parameters COMMA function_parameter ;

function_parameter ::=
  parameter_name COLON type_expression ;

external_section ::=
  external_declaration ;

external_declaration ::=
  EXTERNAL external_function COLON type_expression ;

external_function ::=
  external_function1 |
  external_function2 ;

external_function1 ::=
  function_name |
  external_function1 COMMA function_name ;

external_function2 ::=
  function_name LPAR external_parameters RPAR ;

external_parameters ::=
  external_parameter |
```

---

```
external_parameters COMMA external_parameter ;

external_parameter ::=
  parameter_name COLON type_expression ;

abstraction_part_opt ::=
  abstraction_part |
  /* lambda */ ;

abstraction_part ::=
  ABSTRACTIONS COLON abstraction_section_list_opt ;

abstraction_section_list_opt ::=
  abstraction_section_list |
  /* lambda */ ;

abstraction_section_list ::=
  abstraction_section |
  abstraction_section_list abstraction_section ;

abstraction_section ::=
  public_opt rule_abstraction SEMI |
  public_opt rule_abstraction ;

public_opt ::=
  PUBLIC |
  /* lambda */ ;

rule_abstraction ::=
  action_declaration ;

action_declaration ::=
  ACTION rule_declaration ;

rule_declaration ::=
  rule_name rule_params_opt IS rule_body END |
  rule_name rule_params_opt IS rule_body END rule_name ;
```

```
rule_params_opt ::=
  rule_params |
  /* lambda */ ;

rule_params ::=
  LPAR rule_parameters RPAR ;

rule_parameters ::=
  rule_parameter |
  rule_parameters COMMA rule_parameter ;

rule_parameter ::=
  passing_mode parameter_name COLON type_expression |
  parameter_name COLON type_expression ;

rule_body ::=
  one_pass_transition |
  repeating_transition ;

one_pass_transition ::=
  local_declarations BEGIN COLON main_transition |
  begin_opt main_transition ;

begin_opt ::=
  BEGIN COLON |
  /* lambda */ ;

repeating_transition ::=
  local_declarations LOOP COLON main_transition |
  LOOP COLON main_transition ;

local_declarations ::=
  ALGEBRA COLON local_algebra_section_list_opt ;

local_algebra_section_list_opt ::=
  local_algebra_section_list |
  /* lambda */ ;
```

---

```
local_algebra_section_list ::=
  local_algebra_section |
  local_algebra_section_list local_algebra_section ;

local_algebra_section ::=
  local_type |
  local_function |
  local_external ;

local_type ::=
  type_declaration ;

local_function ::=
  function_declaration ;

local_external ::=
  external_declaration ;

passing_mode ::=
  IN |
  OUT |
  INOUT ;

main_transition ::=
  stepwise_transition |
  single_transition ;

stepwise_transition ::=
  step_block |
  stepwise_transition step_block ;

step_block ::=
  STEP step_number COLON transition_rule ;

step_number ::= INTEGER_LIT ;

single_transition ::=
  transition_rule ;
```

```
transition_rule ::=
  rule |
  transition_rule SEMI rule ;

rule ::=
  basic_rule |
  conditional_rule |
  universal_rule |
  agent_rule ;

basic_rule ::=
  call_or_update_rule |
  abbreviation_rule |
  empty_rule ;

call_or_update_rule ::=
  location ASSIGN expression |
  call_rule ;

call_rule ::=
  function_call_or_variable_designator ;

location ::=
  dynamic_function_designator ;

dynamic_function_designator ::=
  function_call_or_variable_designator ;

abbreviation_rule ::=
  LET variable_name EQUALS expression ;

empty_rule ::=
  /* lambda */ ;

conditional_rule ::=
  if_rule |
```

---

```
case_rule |
choose_rule |
with_rule ;

if_rule ::=
  IF expression THEN transition_rule else_part END ;

else_part ::=
  elseif_term_list_opt ELSE transition_rule |
  elseif_term_list_opt ;

elseif_term_list_opt ::=
  elseif_term_list |
  /* lambda */ ;

elseif_term_list ::=
  elseif_term |
  elseif_term_list elseif_term ;

elseif_term ::=
  ELSEIF expression THEN transition_rule ;

case_rule ::=
  CASE expression case_rule_atom_list_opt case_rule_otherwise END |
  CASE expression case_rule_atom_list_opt END ;

case_rule_atom_list_opt ::=
  case_rule_atom_list |
  /* lambda */ ;

case_rule_atom_list ::=
  case_rule_atom |
  case_rule_atom_list case_rule_atom ;

case_rule_atom ::=
  OF expression DARROW transition_rule ;

case_rule_otherwise ::=
```

```
OTHERWISE DARROW transition_rule ;

choose_rule ::=
  CHOOSE choose_elements SATISFYING expression DO transition_rule END |
  CHOOSE choose_elements DO transition_rule END ;

choose_elements ::=
  choose_element |
  choose_elements COMMA choose_element ;

choose_element ::=
  variable_name COLON choose_domain |
  choose_domain ;

choose_domain ::=
  expression ;

with_rule ::=
  WITH expression with_rule_atom_list_opt with_rule_otherwise END |
  WITH expression with_rule_atom_list_opt END ;

with_rule_atom_list_opt ::=
  with_rule_atom_list |
  /* lambda */ ;

with_rule_atom_list ::=
  with_rule_atom |
  with_rule_atom_list with_rule_atom ;

with_rule_atom ::=
  AS with_pattern DARROW transition_rule ;

with_pattern ::=
  function_name COLON pattern |
  pattern ;

with_rule_otherwise ::=
  OTHERWISE DARROW transition_rule ;
```



---

```
universal_rule ::=
  forall_rule ;

forall_rule ::=
  FORALL for_elements DO transition_rule END ;

for_elements ::=
  for_element |
  for_elements COMMA for_element ;

for_element ::=
  variable_name COLON for_domain ;

for_domain ::=
  expression ;

agent_rule ::=
  create_rule |
  dispatch_rule |
  stop_rule |
  return_rule |
  destroy_rule ;

create_rule ::=
  CREATE agent_declaration_list ;

agent_declaration_list ::=
  agent_declaration |
  agent_declaration_list COMMA agent_declaration ;

agent_declaration ::=
  single_agent_declaration |
  multiple_agent_declaration ;

single_agent_declaration ::=
  agent_name COLON agent_type ;
```

```
multiple_agent_declaration ::=
  agent_name COLON agent_type number_of_agents ;

agent_type ::=
  AGENT1 OF module_name ;

number_of_agents ::=
  LPAR expression RPAR ;

dispatch_rule ::=
  DISPATCH agent_designator_list ;

stop_rule ::=
  STOP ;

return_rule ::=
  RETURN ;

destroy_rule ::=
  DESTROY agent_designator_list ;

agent_designator_list ::=
  agt_designator |
  agent_designator_list COMMA agt_designator ;

agt_designator ::=
  agent_name |
  function_call_or_variable_designator DOT agent_name ;

basic_type ::=
  BOOL |
  CHAR |
  INT |
  REAL |
  STRING |
  LIST2 |
  SET2 |
  AGENT2 |
```

---

```
INPUT |
OUTPUT ;

type_identification ::=
  type_designator |
  builtin_type ;

builtin_type ::=
  basic_type ;

type_expression ::=
  enumeration_type |
  compound_type |
  type_name_or_user_defined_type |
  abstraction_type |
  LPAR type_expression RPAR ;

type_name_or_user_defined_type ::=
  type_designator LPAR type_arguments RPAR |
  type_designator_or_type_name ;

type_designator_or_type_name ::=
  module_name DOT type_name |
  type_name ;

enumeration_type ::=
  PUBLIC ENUM LBRACE constant_name_comma_list RBRACE
  ENUM LBRACE constant_name_comma_list RBRACE ;

constant_name_comma_list ::=
  constant_name |
  constant_name_comma_list COMMA constant_name ;

compound_type ::=
  union_type |
  tuple_type |
  list_type |
  file_type |
```

```
agent_type |
functional_type |
set_type |
tree_type ;

union_type ::=
  QUESTION_MARK |
  LPAR type_expression BAR type_expression RPAR ;

tuple_type ::=
  TUPLE LPAR tuple_elements_opt RPAR ;

tuple_elements_opt ::=
  tuple_elements |
  /* lambda */ ;

tuple_elements ::=
  tuple_element |
  tuple_elements COMMA tuple_element ;

tuple_element ::=
  function_name COLON type_expression |
  type_expression ;

list_type ::=
  LIST1 OF type_expression ;

file_type ::=
  FILE OF type_expression ;

agent_type ::=
  AGENT1 OF module_name ;

functional_type ::=
  LPAR type_expression RPAR SARROW LPAR type_expression RPAR ;

set_type ::=
  SET1 OF type_expression ;
```

---

```
tree_type ::=
  LBRACK tree_element_list_opt RBRACK ;

tree_element_list_opt ::=
  tree_element_list |
  /* lambda */ ;

tree_element_list ::=
  type_identification |
  STRING_LIT ;

type_arguments ::=
  type_argument |
  type_arguments COMMA type_argument ;

type_argument ::=
  type_expression ;

abstraction_type ::=
  ACTION LPAR parameter_type_comma_list RPAR
  ACTION ;

parameter_type_comma_list ::=
  parameter_type |
  parameter_type_comma_list COMMA parameter_type ;

parameter_type ::=
  passing_type_opt parameter_name_opt type_expression ;

passing_type_opt ::=
  passing_type |
  /* lambda */ ;

passing_type ::=
  IN |
  OUT |
  INOUT ;
```

```
parameter_name_opt ::=
  parameter_name COLON |
  /* lambda */ ;

expression ::=
  simple_expression |
  compound_expression ;

simple_expression ::=
  simple_expression relational_op range_expression |
  range_expression ;

range_expression ::=
  adding_expression DD adding_expression |
  adding_expression ;

adding_expression ::=
  adding_expression adding_op multiplying_expression |
  multiplying_expression ;

multiplying_expression ::=
  multiplying_expression multiplying_op unary_expression |
  unary_expression ;

unary_expression ::=
  unary_op in_expression |
  in_expression ;

in_expression ::=
  IN constructing_expression |
  constructing_expression ;

constructing_expression ::=
  constructing_expression CC primary |
  primary ;

primary ::=
```

---

```
literal |
set_expression |
node_expression |
LPAR expression_comma_list_opt RPAR |
call_or_variable_or_coersion_or_type |
DEFAULT LPAR type_identification RPAR ;

call_or_variable_or_coersion_or_type ::=
name |
call_or_variable_or_coersion_or_type DOT name ;

function_call_or_variable_designator ::=
name |
function_call_or_variable_designator DOT name ;

name ::=
ID |
ID LPAR expression_comma_list RPAR ;

expression_comma_list_opt ::=
expression_comma_list |
/* lambda */ ;

expression_comma_list ::=
expression |
expression_comma_list COMMA expression ;

set_expression ::=
LBRACE expression_comma_list_opt RPAR ;

node_expression ::=
LBRACK node_element_list_opt RBRACK ;

node_element_list_opt ::=
node_element_list |
/* lambda */ ;
```

```
node_element_list ::=
  node_element |
  node_element_list node_element ;

node_element ::=
  STRING_LIT |
  node_expression |
  function_call_or_variable_designator ;

relational_op ::=
  EQUALS |
  NEQUALS |
  LT |
  GT |
  LTE |
  GTE |
  IS ;

adding_op ::=
  PLUS |
  MINUS |
  OR |
  XOR ;

multiplying_op ::=
  TIMES |
  PER |
  PERC |
  AND ;

unary_op ::=
  MINUS |
  PLUS |
  NOT ;
```



---

```
compound_expression ::=
  if_expression |
  case_expression |
  with_expression ;

if_expression ::=
  IF expression THEN expression else_exp_part END ;

else_exp_part ::=
  elseif_exp_part_list_opt ELSE expression ;

elseif_exp_part_list_opt ::=
  elseif_exp_part_list |
  /* lambda */ ;

elseif_exp_part_list ::=
  elseif_exp_part |
  elseif_exp_part_list elseif_exp_part ;

elseif_exp_part ::=
  ELSEIF expression THEN expression ;

case_expression ::=
  CASE expression case_exp_clause_list_opt default_case_exp_opt END ;

case_exp_clause_list_opt ::=
  case_exp_clause_list |
  /* lambda */ ;

case_exp_clause_list ::=
  case_exp_clause |
  case_exp_clause_list case_exp_clause ;

case_exp_clause ::=
  OF expression DARROW expression ;

default_case_exp_opt ::=
  OTHERWISE DARROW expression |
```

```
/* lambda */ ;

with_expression ::=
  WITH expression with_exp_clause_list_opt default_with_opt END ;

with_exp_clause_list_opt ::=
  with_exp_clause_list |
  /* lambda */ ;

with_exp_clause_list ::=
  with_exp_clause |
  with_exp_clause_list with_exp_clause ;

with_exp_clause ::=
  AS with_exp_pattern DARROW expression ;

with_exp_pattern ::=
  function_name COLON pattern |
  pattern ;

default_with_opt ::=
  OTHERWISE DARROW expression |
  /* lambda*/ ;

pattern ::=
  type_identification |
  tuple_or_list_pattern |
  node_pattern ;

tuple_or_list_pattern ::=
  empty_tuple_or_list_pattern |
  LPAR mixed CC function_name RPAR |
  LPAR mixed RPAR ;

empty_tuple_or_list_pattern ::=
  LPAR RPAR ;

mixed ::=
```

---

```
mix |
mixed COMMA mix ;

mix ::=
  function_name COLON element |
  element ;

element ::=
  module_name DOT type_name |
  type_or_function_name |
  builtin_type |
  tuple_or_list_pattern |
  node_pattern ;

type_or_function_name ::=
  ID ;

node_pattern ::=
  LBRACK node_pattern_element_list_opt RBRACK ;

node_pattern_element_list_opt ::=
  node_pattern_element_list |
  /* lambda */ ;

node_pattern_element_list ::=
  node_pattern_element |
  node_pattern_element_list node_pattern_element ;

node_pattern_element ::=
  STRING_LIT |
  function_name ;

type_designator ::=
  module_name DOT type_name |
  type_name ;

agent_interface ::=
  INTERFACE interface_name interface_body END interface_name |
```

```
INTERFACE interface_name interface_body END ;

interface_body ::=
  interface_section |
  interface_body interface_section ;

interface_section ::=
  exported_type |
  exported_abstraction ;

exported_type ::=
  TYPE type_name SEMI |
  TYPE type_name ;

exported_abstraction ::=
  ACTION rule_name formal_parameters_opt ;

formal_parameters_opt ::=
  formal_parameters |
  /* lambda */ ;

formal_parameters ::=
  formal_parameter |
  formal_parameters COMMA formal_parameter ;

formal_parameter ::=
  transmission_mode_opt parameter_name COLON type_expression |
  transmission_mode_opt type_expression ;

transmission_mode_opt ::=
  transmission_mode |
  /* lambda */ ;

transmission_mode ::=
  IN |
  OUT |
  INOUT ;
```

---

```
machina_definition ::=
  MACHINA machina_name machina_body END machina_name |
  MACHINA machina_name machina_body END ;

machina_body ::=
  machina_directive_list_opt ;

machina_directive_list_opt ::=
  machina_directive_list |
  /* lambda */ ;

machina_directive_list ::=
  machina_directive |
  machina_directive_list machina_directive ;

machina_directive ::=
  AGENT1 OF module_name LPAR INTEGER_LIT RPAR semi_opt |
  AGENT1 OF module_name semi_opt;

semi_opt ::=
  SEMI |
  /* lambda */ ;
```



# Referências Bibliográficas

- [Aho et al., 1986] Aho, A.; Sethi, R. e Ullman, J. (1986). *Compilers, Principles, Techniques, and Tools*. Addison-Wesley.
- [Aho e Ullman, 1972] Aho, A. V. e Ullman, J. D. (1972). *The Theory of Parsing, Translation, and Compiling*. Prentice Hall Professional Technical Reference.
- [AnaGram, 2006] AnaGram (2006). Anagram. <http://www.parsifalsoft.com>.  
Último acesso: 13/01/2007.
- [Beach, 1974] Beach, B. (1974). Storage organization for a type of sparse matrix. In *5th Southeastern Conference on Combinatorics, Graph Theory and Computing*, pp. 245–252.
- [Bell, 1974] Bell, J. (1974). A compression method for compiler precedence tables. In *IFIP Congress*, pp. 359–362.
- [Bigonha e Bigonha, 1983] Bigonha, R. e Bigonha, M. (1983). A method for efficient compactation of lalr(1) parsing tables. Technical report, Department of Computer Science, Federal University of Minas Gerais.
- [Bigonha et al., 2006] Bigonha, R.; Tirelo, F.; Iorio, V. e Bigonha, M. (2006). A linguagem de especificação formal máquina 2.0. Technical report, Federal University of Minas Gerais - Department of Computer Science, Programming Languages Laboratory.
- [Bison, 2007] Bison (2007).
- [Blythe et al., 1994] Blythe, S.; James, M. e Rodger, S. (1994). Llpase and lrparse: visual and interactive tools for parsing. In *SIGCSE '94: Proceedings of the twenty-fifth SIGCSE symposium on Computer science education*, pp. 208–212. ACM Press.
- [Booch et al., 2005] Booch, G.; Rumbaugh, J. e Jacobson, I. (2005). *The Unified Modeling Language User Guide*. Addison-Wesley Longman Publishing.

- [Charles, 1991] Charles, P. (1991). *A Practical Method for Constructing Efficient LALR(k) Parsers with Automatic Error Recovery*. PhD thesis, New York University.
- [Cormen et al., 2001] Cormen, T. H.; Stein, C.; Rivest, R. L. e Leiserson, C. E. (2001). *Introduction to Algorithms*. McGraw-Hill Higher Education.
- [CUP, 2007] CUP (2007). Cup: Lalr parser generator in java. <http://www2.cs.tum.edu/projects/cup/>.  
Último acesso: 13/01/2007.
- [Dencker et al., 1984] Dencker, P.; Dürre, K. e Heuft, H. (1984). Optimization of parser tables for portable compilers. *ACM Trans. Program. Lang. Syst.*, 6(4):546–572.
- [DeRemer e Pennello, 1982] DeRemer, F. e Pennello, T. (1982). Efficient computation of lalr(1) look-ahead sets. *ACM Trans. Program. Lang. Syst.*, 4(4):615–649.
- [DeRemer, 1969] DeRemer, F. L. (1969). *Practical Translators for LR(k) Languages*. PhD thesis, Massachusetts Institute of Technology.
- [DeRemer, 1971] DeRemer, F. L. (1971). Simple lr(k) grammars. *Commun. ACM*, 14(7):453–460.
- [Dürre et al., 1976] Dürre, K.; Goss, G. e Schmitt, A. (1976). Minimal representation of lr-parsers. Unpublished manuscript.
- [Eve e Kurkio-Suonio, 1977] Eve, J. e Kurkio-Suonio, R. (1977). On computing the transitive closure of a relation. 8.
- [GCC, 2006] GCC (2006). Gcc: Gnu c compiler. <http://gcc.gnu.org>.  
Último acesso: 15/02/2006.
- [Gurevich, ] Gurevich, Y. Evolving algebras 1993: Lipari Guide. In Börger, E., editor, *Specification and Validation Methods*. Oxford University Press.
- [Johnson, 1979] Johnson, S. (1979). Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pp. 353–387. Holt, Rinehart, and Winston.
- [Joliat, 1974] Joliat, M. L. (1974). Practical minimisation of lr(k) parser tables. In *IFIP Congress*, pp. 376–389. Elsevier North-Holland.
- [Kaplan e D., 2000] Kaplan, A. e D., S. (2000). Cupv: a visualization tool for generated parsers. In *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pp. 11–15. ACM Press.



- [Khuri e Sugono, 1998] Khuri, S. e Sugono, Y. (1998). Animating parsing algorithms. In *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pp. 232–236. ACM Press.
- [Knuth, 1965] Knuth, D. E. (1965). On the translation of languages from left to right. 8:607–639.
- [Kristensen e Madsen, 1981] Kristensen, B. B. e Madsen, O. L. (1981). Methods for computing lalr(k) lookahead. *ACM Trans. Program. Lang. Syst.*, 3(1):60–82.
- [LaLonde et al., 1971] LaLonde, W. R.; Lee, E. S. e Horning, J. J. (1971). An lalr( $k$ ) parser generator. pp. 151–153. IFIP Congress.
- [Lovato e Kleyn, 1995] Lovato, M. e Kleyn, M. (1995). Parser visualizations for developing grammars with yacc. In *SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education*, pp. 345–349. ACM Press.
- [Motif, 2007] Motif (2007). Motif. <http://www.opengroup.org/motif/>.  
Last access: 13/01/2007.
- [Parse++, 2005] Parse++, V. (2005). Visual parse++. <http://www.sand-stone.com/Visual%20Parse++.htm>.  
Último acesso: 13/01/2007.
- [Passos et al., 2007] Passos, L. T.; Bigonha, M. A. e Bigonha, R. S. (2007). A methodology for removing lalr conflicts. XI Simpósio Brasileiro de Linguagens de Programação.
- [SableCC, 2007] SableCC (2007). Sablecc parser generator. <http://www.sablecc.org>.  
Último acesso: 13/01/2007.
- [Schmitt, 1979] Schmitt, A. (1979). Minimizing storage space of sparse matrices by graph coloring algorithms. In *Graphs, Data Structures, Algorithms*, pp. 157–168.
- [Shoup, 1999] Shoup, D. (1999). Masters project report. Technical report, Department of Computer Science, Clemson-University, Clemson, SC.
- [Stasko, 1992] Stasko, J. (1992). Animating algorithms with xtango. *SIGACT News*, 23(2):67–71.
- [Tarjan, 1972] Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2).

- [Tewarson, 1968] Tewarson, R. (1967/1968). Row column permutation of sparse matrices. *Computer Journal*, pp. 300–305.
- [Tirelo e Bigonha, 2006] Tirelo, F. e Bigonha, R. (2006). Notus. Technical report, Federal University of Minas Gerais - Department of Computer Science, Programming Languages Laboratory.
- [Vegdahl, 2000] Vegdahl, S. (2000). Using visualization tools to teach compiler design. In *Proceedings of the second annual CCSC on Computing in Small Colleges Northwestern conference*, pp. 72–83. Consortium for Computing Sciences in Colleges.
- [Yacc, 2006] Yacc, V. (2006). Visual yacc. <http://cs.gmu.edu/~white/Pages/cra.html>.  
Último acesso: 23/03/2006.
- [Ziegler, 1977] Ziegler, S. F. (1977). Smaller faster driven parser. Unpublished manuscript.