

## SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO.....</b>	<b>3</b>
1.1	OBJETIVOS .....	3
1.1	HISTÓRICO .....	4
1.2	FUNCIONAMENTO DOS COMPILADORES .....	4
1.3	ORGANIZAÇÃO DE UM COMPILADOR .....	6
1.4	CARREGADORES E EDITORES DE LIGAÇÃO .....	7
<b>2</b>	<b>INTRODUÇÃO À LINGUAGENS FORMAIS E AUTÔMATOS.....</b>	<b>8</b>
2.1	LINGUAGENS.....	8
2.2	GRAMÁTICAS .....	9
2.3	ÁRVORES DE DERIVAÇÃO .....	10
2.4	HIERARQUIA DE CHOMSKY .....	13
2.5	AUTÔMATOS FINITOS.....	16
2.5.1	<i>Autômatos Finitos Determinísticos.</i> .....	17
2.5.2	<i>Autômatos Finitos Não Determinísticos</i> .....	19
2.5.3	<i>Implementação de Autômatos Finitos.</i> .....	20
2.5.4	<i>Tabelas de Transição para Autômatos Finitos.</i> .....	21
<b>3</b>	<b>ANÁLISE LÉXICA .....</b>	<b>26</b>
3.1	FUNÇÕES DA ANÁLISE LÉXICA .....	26
3.2	FUNCIONAMENTO DO ANALISADOR LÉXICO .....	27
3.3	AUTÔMATOS FINITOS.....	29
3.3.1	<i>Exercício:</i> .....	31
3.4	EXPRESSÕES REGULARES .....	34
3.5	RECONHECIMENTO DE NÚMEROS.....	36
3.5.1	<i>Exercício:</i> .....	37
3.6	RECONHECIMENTO DE IDENTIFICADORES .....	37
3.7	JUNTANDO AS PARTES .....	38
3.8	TABELA DE PALAVRAS RESERVADAS .....	39
3.9	TABELA DE SÍMBOLOS .....	41
3.10	GERADORES AUTOMÁTICOS DE ANALISADOR LÉXICO.....	44
3.10.1	<i>Especificação das Sentenças Regulares</i> .....	44
3.10.2	<i>Linguagem de Especificação de Padrões</i> .....	45
3.10.3	<i>Exemplo de Especificação de Sentenças</i> .....	46
3.10.4	<i>Resumindo: Geração do Analisador Léxico</i> .....	52
<b>4</b>	<b>ANÁLISE SINTÁTICA.....</b>	<b>53</b>
4.1	FUNÇÕES DA ANÁLISE SINTÁTICA .....	53
4.2	AMBIGUIDADE .....	53
4.3	PRODUÇÕES VAZIAS.....	55
4.4	RECURSAO À ESQUERDA.....	55
4.5	FATORAÇÃO À ESQUERDA .....	58
4.6	ANÁLISE SINTÁTICA DESCENDENTE .....	58
4.6.1	<i>Análise Sintática Descendente com Retrocesso</i> .....	58
4.6.2	<i>Análise Sintática Descendente Recursiva</i> .....	59
4.6.3	<i>Análise Sintática Descendente Recursiva Preditiva</i> .....	62
4.6.4	<i>Análise Sintática Preditiva Não Recursiva</i> .....	66
4.6.4.1	<i>Primeiro e Seguinte</i> .....	68
4.6.4.2	<i>Tabelas Sintáticas Preditivas</i> .....	70
4.6.4.3	<i>Gramáticas LL(1)</i> .....	70

4.6.5	<i>Exercícios</i> .....	72
4.7	ANÁLISE SINTÁTICA ASCENDENTE .....	74
4.7.1	<i>Construção de Analisadores Ascendentes</i> .....	75
4.7.2	<i>Análise de Precedência de Operadores</i> .....	77
4.7.2.1	Relações de Precedência de Operador .....	77
4.7.2.2	Construção da Tabela de Precedência de Operadores .....	80
4.7.2.3	Funções de Precedência .....	84
4.7.3	<i>Análise LR(<math>k</math>)</i> .....	86
4.7.4	<i>Construção de Analisadores Sintáticos SLR</i> .....	89
4.7.4.1	Operação Closure .....	90
4.7.4.2	Operação Goto .....	91
4.7.4.3	Algoritmo para a Construção do Conjunto de Itens LR(0) .....	91
4.7.4.4	Construção da Tabela Sintática SLR .....	94
4.8	GERADORES AUTOMÁTICOS DE ANALISADORES SINTÁTICOS .....	95
4.8.1	<i>YACC / BISON</i> .....	95
4.8.2	<i>ANTLR</i> .....	99
<b>5</b>	<b>TRADUÇÃO DIRIGIDA PELA SINTAXE</b> .....	<b>119</b>
5.1	DEFINIÇÃO DIRIGIDA PELA SINTAXE .....	120
5.2	ATRIBUTOS SINTETIZADOS .....	121
5.3	ATRIBUTOS HERDADOS .....	121
5.4	GRAFOS DE DEPENDÊNCIA .....	122
5.5	CONSTRUÇÃO DE ÁRVORES SINTÁTICAS .....	124
5.6	AVALIAÇÃO ASCENDENTE DE DEFINIÇÕES S-ATRIBUÍDAS .....	127
5.7	DEFINIÇÕES L-ATRIBUÍDAS .....	129
5.8	ESQUEMAS DE TRADUÇÃO .....	130
5.9	TRADUÇÃO DESCENDENTE (TOP-DOWN) .....	131
5.10	PROJETO DE UM TRADUTOR PREDITIVO .....	135
<b>6</b>	<b>GERAÇÃO DE CÓDIGO INTERMEDIÁRIO</b> .....	<b>137</b>
6.1	REPRESENTAÇÕES INTERMEDIÁRIAS .....	137
6.1.1	<i>Árvores e Grafos Sintáticos</i> .....	137
6.1.2	<i>Notações Infixa, Pós-fixada e Pré-fixada</i> .....	139
6.1.3	<i>Código de Três Endereços</i> .....	141
6.1.4	<i>Quádruplas e Tripulas</i> .....	143
6.1.5	<i>Reutilização de temporários</i> .....	145
6.2	CONSTRUÇÃO DE TABELAS DE SÍMBOLOS .....	145
6.3	COMPILAÇÃO DE ATRIBUIÇÕES .....	149
6.3.1	<i>Atribuições</i> .....	150
6.3.2	<i>Verificação e Conversão de Tipos</i> .....	151
6.3.3	<i>Endereçando Elementos de Matrizes</i> .....	152
6.4	EXPRESSÕES LÓGICAS E COMANDOS DE CONTROLE .....	156
6.4.1	<i>Representação Numérica</i> .....	156
6.4.2	<i>Representação por Fluxo de Controle</i> .....	159
6.4.3	<i>Backpatching</i> .....	163
<b>7</b>	<b>REFERÊNCIAS BIBLIOGRÁFICAS</b> .....	<b>164</b>

## 1 Introdução

### 1.1 Objetivos

A disciplina de *Compiladores* tem por objetivo a introdução dos conceitos e estrutura funcional dos compiladores.

Ao concluir a disciplina o aluno deverá estar apto a:

- ❑ Distinguir as etapas relativas a um projeto de um compilador, bem como as diferenças existentes entre a Compilação, Montagem, Ligação e Interpretação;
- ❑ Reconhecer os tipos de Gramáticas, Linguagens e Reconhecedores existentes;
- ❑ Reconhecer as funções e as ações desempenhadas por um Analisador Léxico, sendo capaz de implementá-las;
- ❑ Reconhecer os tipos de Analisadores Sintáticos, suas funções e ações desempenhadas, sendo capaz de implementá-los;
- ❑ Reconhecer as atividades desempenhadas por um Analisador Semântico, Gerador de Código e Otimizador de Código;
- ❑ Utilizar as ferramentas e as técnicas necessárias para a construção de compiladores;
- ❑ Projetar e implementar um Compilador para uma máquina com um conjunto de instruções simples.

Em conclusão, a disciplina propicia ao aluno a aplicação de todos os conceitos e recursos estudados durante o curso, permitindo que ele possa analisar e implementar um projeto em computação com um maior nível de complexidade. Deste modo, espera-se contribuir para a formação de um profissional com uma visão mais ampla, calçada em conceitos largamente difundidos na Ciência da Computação e Engenharia da Computação, não o restringindo apenas à implementação e manutenção de Sistemas de Informações. Além disso, sendo normalmente estes conceitos um pré-requisito para cursos de pós-graduação, espera-se estar contribuindo para que o egresso possa continuar seus estudos a nível de mestrado e doutorado.

Neste sentido, a disciplina vem somar contribuições para a formação de um profissional que, no mercado, possa vir a propor e utilizar ambientes e plataformas operacionais, administrar meios e recursos relacionados às atividades de processamento de dados, além de dar subsídios para que ele possa entender para transmitir e difundir as novas tecnologias nas áreas da informática e das telecomunicações.

## 1.1 Histórico

Nos primeiros dias da computação, computadores eram programados diretamente, em linguagem de máquina. Os programadores, para implementar um programa, necessitavam combinar certas chaves e fios para que as tarefas fossem executadas. As dificuldades eram tremendas, sendo que um programa levava dias para ser feito e depurado.

A necessidade por um modo mais fácil de trabalho, além de uma maior segurança na confecção dos programas, levou, na década de 50, ao surgimento dos primeiros computadores a cartão perfurado e as linguagens de montagem (Assembly). Os montadores facilitavam muito a vida dos programadores, mas uma linguagem mais próxima à utilizada pelo homem seria mais ideal. Deste modo, surgiu, por volta do final da década de 50, as primeiras linguagens de alto nível, como o Fortran e o Algol. Com elas surgiram os primeiros compiladores para estas linguagens.

Atualmente existem um grande número de linguagens disponíveis, sendo distribuídas dentre quatro grandes paradigmas: *Procedimental* ou *Imperativas*, *Funcional*, *Lógico* e *Orientado à Objeto*. A escolha de uma linguagem e de um paradigma depende dos objetivos a serem atingidos. Entretanto, para fins comerciais e científicos, normalmente se utiliza linguagens dos paradigmas procedural ou orientado a objetos. Dentre estas linguagens podemos citar: C, C++, Pascal, Object Pascal, Delphi, Visual Basic, entre outras. Para implementação de inteligência artificial, normalmente são utilizadas linguagens dos paradigmas lógico e funcional. Nestes paradigmas, as linguagens mais conhecidas são: Prolog, Gödel, Isabelle, Haskel, Lisp e Scheme.

Nesta disciplina iremos enfocar aspectos de compilação para linguagens procedimentais. Os mesmos princípios se aplicam para os paradigmas orientados a objeto, lógico e funcional. Entretanto, para estes são necessários o acréscimo de algumas funcionalidades ou a construção de máquinas virtuais, o que foge do nosso escopo.

## 1.2 Funcionamento dos Compiladores

Basicamente, um compilador, denominado aqui por C, é um programa de computador escrito em uma linguagem  $L$ , para uma máquina  $M$ , cuja finalidade é converter um programa  $P_f$ , denominado *programa-fonte*, escrito em uma linguagem  $L_f$ , denominada *linguagem-fonte*, para uma máquina  $M_f$ , para um programa  $P_o$ , denominado *programa-objeto*, em uma linguagem  $L_o$ , denominada *linguagem-objeto*, o qual será executado em uma máquina  $M_o$ . Graficamente teríamos:

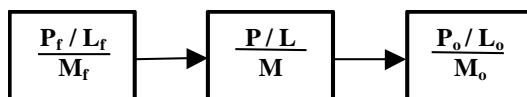


Figura 1: Esquema de tradução

O compilador C pode ser composto de várias fases, denominadas neste caso de *passos do compilador*. Em cada passo é usado uma *linguagem-fonte* e uma *linguagem-objeto* originais, próprias desse passo. No primeiro passo temos a *linguagem-fonte* a ser compilada,  $L_f$ , e no último passo a *linguagem-objeto* final desejada,  $L_o$ . As outras linguagens envolvidas são denominadas *linguagens-intermediárias*. Graficamente teríamos o mesmo que representado pela Figura 2.

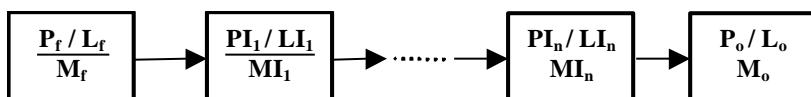


Figura 2: Esquema de tradução em vários passos

Em alguns casos a *linguagem-objeto* gerada pelo compilador é uma linguagem de montagem (Assembly), necessitando assim de um passo adicional, que é a montagem. Neste caso, o programa responsável por esta tarefa é denominado montador (Assembler). A função básica deste programa é a tradução do código fonte Assembly para linguagem de máquina na qual o programa será executado. A principal característica deste tipo de tradutor é que para cada instrução Assembly será gerada uma única instrução de máquina.

As etapas realizadas pelo montador são:

- ❑ Substituição dos mnemônicos encontrados por instruções de máquina;
- ❑ Substituição dos nomes simbólicos (nomes de variáveis e rótulos de desvio) por endereços de memória;
- ❑ Reservar espaço de memória para o armazenamento das instruções e dados;
- ❑ Converter valores de constantes para código binário; e
- ❑ Examinar a correção de cada instrução

Em outros casos a *linguagem-objeto* encontra-se em uma forma intermediária, não podendo ser executado diretamente pela máquina. E em outros casos, não se deseja compilar o *programa-fonte*, mas executá-lo diretamente. Nestes casos é necessária a presença de um *Interpretador*, o qual simula uma máquina virtual, possibilitando assim a execução do programa sobre uma máquina real.

Basicamente, a *interpretação* se caracteriza por realizar as fases de compilação e execução de cada um dos comandos encontrados no programa. Na realidade, o *interpretador* lê cada linha do programa fonte, extrai os comandos nela presentes, verifica a sua sintaxe, e os executa diretamente. Não existem etapas intermediárias. Caso uma linha de programa seja executada mais de uma vez, ela deverá ser novamente interpretada. São exemplos de linguagens interpretadas: *Basic*, *SmallTalk*, *Apl*, etc.

## 1.3 Organização de um Compilador

As fases de um compilador podem ser classificadas em dois grupos: *análise* e *síntese*. Na fase de *análise*, encontram-se as fases de *análise léxica*, *análise sintática* e *análise semântica*. Já na fase de *síntese* encontram-se as fases de *geração de código intermediário*, *otimização* e *geração de código*. Adicionalmente as estas fases, existem ainda duas fases adicionais que interagem com todas as fases do compilador: o *gerenciamento de tabelas* e o *tratamento de erros*. A Figura 3 ilustra todas estas fases.

A fase de *análise léxica* tem por objetivo ler o programa fonte e transformar sequências de caracteres em uma representação interna, denominada *itens léxicos*. Por exemplo, suponha a expressão abaixo, descrita em *Pascal*:

*Exp* := (*A* + *B*) \* 1.5 ;.

Os itens léxicos contidos nesta expressão, são: ***Exp***, ***:=***, ***(***, ***A***, ***,***, ***B***, ***)***, ***\****, ***1.5*** e ***;***. Os itens léxicos a serem reconhecidos pelo *analizador léxico* são determinados pela *gramática da linguagem-fonte*. Deste modo, caso um *item léxico* não seja definido por esta *gramática*, um *erro léxico* é gerado. Por exemplo, suponhamos que uma linguagem só suporte valores inteiros. Então o valor ***1.5*** iria ocasionar um erro, o qual deverá ser tratado.

Estes itens léxicos são fornecidos ao *analizador sintático*, assim que solicitados, o qual os agrupa em uma estrutura denominada de *árvore sintética*. Para isto, ele usa uma série de regras de sintaxe definidas pela *gramática da linguagem-fonte*. Caso algum *item léxico* não seja reconhecido pela *regra sintática* sendo analisada, ocorre então um *erro sintático*, o qual deverá ser tratado. Para o nosso exemplo, a árvore sintética da expressão acima é dada pela Figura 4.

A fase de *análise semântica* analisa a árvore sintática gerada pelo *analizador sintático* em busca de inconsistências semânticas. Uma das tarefas mais importantes é a *verificação de tipos*, ou *análise de contexto*. É nesta fase que são detectadas, por exemplo, os conflitos entre tipos, a ausência de declarações de variáveis, funções e procedimentos.

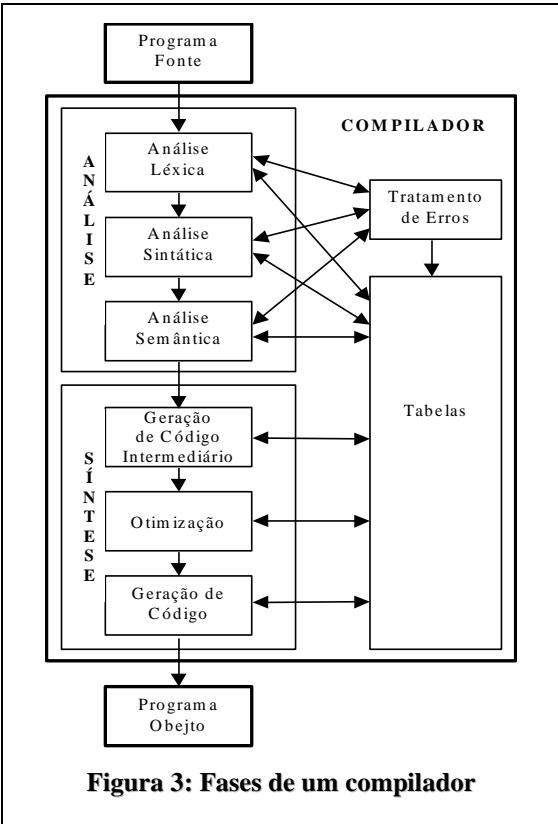


Figura 3: Fases de um compilador

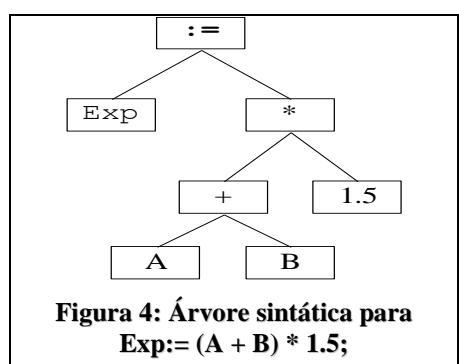


Figura 4: Árvore sintática para *Exp := (A + B) \* 1.5 ;*

A fase de *geração de código intermediário* permite a geração de instruções para uma máquina abstrata, normalmente em *código de três endereços*, mais adequadas a fase de *otimização*. Esta forma intermediária não é executada diretamente pela máquina alvo.

A fase de *otimização* analisa o código no formato intermediário e tenta melhorá-lo de tal forma que venha a resultar em um *código de máquina* mais rápido em termos de *tempo de execução*. Uma das tarefas executadas pelo *otimizador* é a detecção e a eliminação de movimento de dados redundantes e a repetição de operações dentro de um mesmo bloco de programa.

E por fim, a fase de *geração de código* tem como objetivo analisar o código já otimizado e gerar o *código objeto* definitivo para uma máquina alvo. Normalmente este *código objeto* é um *código de máquina relocável* ou um *código de montagem*. Nesta etapa as localizações de memória são selecionadas para cada uma das variáveis usadas pelo programa. Então, as instruções intermediárias são, cada uma, traduzidas numa sequência de instruções de máquina que realizam a mesma tarefa.

## 1.4 Carregadores e Editores de Ligação

A tradução completa de um *programa fonte* requer dois passos: *compilação* ou *montagem* e a *ligação* (*link-edição*). Na compilação (ou montagem) é gerado um *código objeto* no formato *relocável*, o qual não é executado diretamente, visto que em um programa podem ocorrer referências a outros programas ou dados (*referências externas*) os quais se encontram em outros programas, compilados separadamente, ou em bibliotecas (*librarys*) da linguagem sendo compilada. Deste modo, a função do *editor de ligação* (*Linker*) é coletar programas traduzidos separadamente e ligá-los em um único módulo, normalmente denominado *módulo absoluto de carga* ou simplesmente *programa executável*. Já a função do *carregador* é carregar o *módulo absoluto de carga* na memória principal, substituindo os *endereços relativos* ao módulo de carga por endereços reais de memória.

## 2 Introdução à Linguagens Formais e Autômatos

### 2.1 Linguagens

- **Alfabeto:** conjunto finito de símbolos, podendo ser eventualmente vazio. Por exemplo:  $\Sigma = \{a, e, i, o, u\}$ ,  $\Sigma = \{a, b, c, d\}$ ,  $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ , etc. Os elementos de um alfabeto são normalmente denominados de *átomos da linguagem ou símbolos terminais*.
- **Palavra, Cadeia ou Sentença:** sequência finita de símbolos, do alfabeto, justapostos. Por exemplo, seja  $\Sigma = \{a, b, c\}$  um alfabeto, então  $\alpha = aaabcc$ ,  $\beta = cbcaacbc$ ,  $\omega = bacba$  e  $\varepsilon$  são cadeias sobre este alfabeto, sendo esta última denominada de *cadeia vazia*.
- **Tamanho ou Comprimento:** se  $\omega$  é uma cadeia sobre um alfabeto  $\Sigma$ , então  $|\omega|$  denota o seu comprimento, isto é, o número de símbolos que compõem  $\omega$ . Por exemplo, se  $\Sigma = \{a, b\}$ , então  $|a| = 1$ ,  $|ab| = 2$  e  $|\varepsilon| = 0$ .
  - Para se designar o conjunto de todas as cadeias de comprimento  $k$  sobre o alfabeto  $\Sigma$ , utiliza-se a notação  $\Sigma^k$ , sendo que  $\Sigma^0 = \{\varepsilon\}$ . Por exemplo,  $\Sigma^2 = \{aa, ab, ba, bb\}$ .
- **Fecho transitivo reflexivo (ou simplesmente fecho) –  $|\alpha| \geq 0$ :** seja o alfabeto  $\Sigma = \{a, b\}$ . Então,
$$\Sigma^* = \{\varepsilon, a, b, aa, bb, ab, ba, aab, abb, baa, bba, \dots\}$$
- **Fecho transitivo (ou fecho positivo) –  $|\alpha| > 0$ :** seja o alfabeto  $\Sigma = \{a, b\}$ . Então,
$$\Sigma^+ = \Sigma^* - \{\varepsilon\} = \{a, b, aa, bb, ab, ba, aab, abb, baa, bba, \dots\}$$
- **Linguagem Formal:** coleção de cadeias de símbolos, de comprimento finito, denominadas *sentenças da linguagem*.
  - Se  $\alpha$  é uma cadeia qualquer sobre uma linguagem  $L$ , então  $\alpha$  é a concatenação de  $n$  símbolos do alfabeto, sendo que  $|\alpha| = n$ ;
  - Sendo  $\alpha$  e  $\beta$  duas cadeias sobre uma linguagem  $L$ , então  $\gamma = \alpha\beta$  é a concatenação de  $\alpha$  e  $\beta$ , sendo também uma cadeia, e, portanto,  $|\gamma| = |\alpha| + |\beta|$ ;
  - Se  $\alpha$  é uma cadeia sobre uma linguagem  $L$ ,  $\alpha^n$  é a concatenação sucessiva de  $\alpha$  com ela mesma  $n$  vezes. Isto é, se  $\alpha = a$ , então  $\alpha^0 = \varepsilon$ ,  $\alpha^1 = a$ ,  $\alpha^2 = aa$ ,  $\alpha^3 = aaa$ , etc.;
  - Podemos definir uma linguagem por três modos: *enumeração de sentenças, gramáticas ou reconhecedores*.

## 2.2 Gramáticas

- **Gramática:** especificação de um conjunto de leis de formação de sentenças da linguagem, podendo ser definida formalmente pela quádrupla ordenada:

$$G = (V_N, V_T, P, S)$$

onde:

**$V_N$ :** representa o conjunto de todos os símbolos utilizados pela gramática  $G$ , para definir as regras de formação das sentenças da linguagem. Estes símbolos são denominados de *não-terminais*.

**$V_T$ :** representa o conjunto de todos os símbolos utilizados pela gramática para a formação das cadeias aceitas pela linguagem, normalmente denominados de *símbolos terminais* ou simplesmente *terminais*. Ao conjunto  $V = V_N \cup V_T$  dá-se *vocabulário* da gramática  $G$ . Note que  $V_N \cap V_T = \emptyset$ ;

**$P$ :** representa o conjunto de todas as regras de formação utilizadas pela gramática para a produção de cadeias da linguagem. A cada uma destas regras dá-se o nome de *regra de produção* da gramática ou simplesmente *produção*. Esta regras têm a forma:  $\alpha \rightarrow \beta$ , onde  $\alpha$  é uma cadeia contendo no mínimo um *não-terminal*, ou seja  $\alpha \in V^*$ ,  $V_N \subset V^*$ , podendo ser eventualmente vazia, e  $\beta \in V^*$ . Uma sequência de produções do tipo  $\alpha \rightarrow \beta_1, \alpha \rightarrow \beta_2, \dots, \alpha \rightarrow \beta_n$ , pode ser abreviada por  $\alpha \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$ .

**$S$ :** é um elemento de  $V_N$  cuja função é dar início ao processo de geração de sentenças, também chamado de *símbolo inicial* da gramática.

- **Forma Sentencial:** qualquer cadeia de elementos de  $V$  obtida a partir de  $S$  por produções de  $P$  da gramática  $G = (V_N, V_T, P, S)$ . Deste modo:
  - a)  $S$  é uma *forma sentencial*;
  - b) Se  $\alpha\beta\gamma$  é uma *forma sentencial* e  $\beta \rightarrow \delta$ , então  $\alpha\delta\gamma$  também é uma *forma sentencial*, sendo que  $\alpha, \delta, \gamma \in V^*$ , e  $\beta \in V^* V_N V^*$ .
- Observe que uma *sentença* é, portanto, uma forma sentencial  $F$  tal que  $F \in V_T^*$ .
- **Derivação direta:** aplicação de uma regra de produção para obtenção de uma *forma sentencial*. Portanto, se  $\alpha\beta\gamma \in V^*$  e  $\beta \rightarrow \delta \in P$ , então  $\alpha\beta\gamma \Rightarrow \alpha\delta\gamma$  é uma *derivação direta*. Para indicarmos a ocorrência de  $n$  passos de derivação direta, escrevemos  

$$\alpha \Rightarrow^n \beta,$$
- **Derivação não-trivial:** aplicação de no mínimo uma derivação, denotada por  

$$\Rightarrow^+.$$
- **Derivação:** aplicação de zero ou mais derivações diretas, sendo denotada por  

$$\Rightarrow^*.$$

- **Linguagem Gerada:** uma *linguagem gerada* por uma gramática  $G = (V_N, V_T, P, S)$ , é um conjunto de todas as possíveis sentenças por ela geradas através de derivações a partir do símbolo inicial  $S$ . Isto é:

$$L(G) = \{ \omega \in V_T^* \mid S \Rightarrow^+ \omega \}$$

- **Exemplo 1:** seja a gramática  $G_1 = (V_N, V_T, P, S)$ , onde:

$$V_N = \{A, B\}$$

$$V_T = \{0, 1, \epsilon\}$$

$$P = \{A \rightarrow 0A, A \rightarrow B, B \rightarrow 1B, B \rightarrow \epsilon\}$$

$$S = A$$

Através desta gramática podemos construir as sentenças:

$$\{\epsilon, 0, 1, 00, 01, 00, 001, 011, 111, 000, 0001, 0011, \dots\}$$

Tais sentenças também podem ser definidas pela linguagem:

$$L(G_1) = \{ 0^n 1^m \in V_T^* \mid S \Rightarrow 0^* 1^* \text{ e } n, m \geq 0 \}$$

Por exemplo, suponhamos a sentença  $\omega = 001$ . Ela pode ser obtida pelas seguintes derivações:

$$\underline{A} \Rightarrow 0\underline{A} \Rightarrow 00\underline{B} \Rightarrow 001\underline{B} \Rightarrow 001$$

- **Exemplo 2:** dada uma linguagem  $L$ , é possível definir uma gramática  $G_2$  para esta linguagem, tal que  $L(G_2) = L$ . Por exemplo, suponha a linguagem:

$$L = \{ 0^n 1^n \in V_T^* \mid S \Rightarrow 0^* 1^* \text{ e } n \geq 0 \}$$

Observe que para esta linguagem, o número de 1's deve coincidir exatamente com o mesmo número de 0's. Neste caso, teríamos as seguintes regras de produção para a gramática:  $S \rightarrow 0S1$  e  $S \rightarrow \epsilon$ . Portanto, nossa gramática seria definida por:

$$G_2 = (\{S\}, \{0, 1\}, \{S \rightarrow 0S1, S \rightarrow \epsilon\}, S)$$

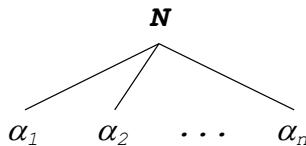
Por exemplo, as seguintes sentenças serão geradas por esta gramática:

$$\{\epsilon, 01, 0011, 000111, 00001111, \dots\}$$

## 2.3 Árvores de Derivação

- **Árvore de Derivação:** forma de representar os passos de derivação de uma sentença para uma gramática  $G = (V_N, V_T, P, S)$ , definida da seguinte forma:

1. A árvore constituída inicialmente pelo *nó* rotulado  $S$  é uma árvore de derivação;
2. A árvore obtida pela substituição de uma *folha* (nó final) rotulada  $N$  de uma árvore de derivação, pela árvore



onde  $N \rightarrow \alpha_1\alpha_2\dots\alpha_n \in P$ ; é também uma árvore de derivação.

Algumas considerações importantes sobre árvores de derivação, são:

1. A construção da árvore termina quando todas as folhas forem símbolos *terminais*;
  2. A cada árvore de derivação corresponde pelo menos uma geração;
  3. A cada geração corresponde apenas uma única árvore de derivação, embora várias gerações possam ter a mesma árvore de derivação;
  4. Qualquer nó, exceto aqueles que são folhas, é rotulado com um símbolo *não-terminal* de  $G$ ;
  5. Um nó *não-terminal*  $N$ , e seus nós descendentes imediatos  $\alpha_1, \alpha_2, \dots, \alpha_n$  correspondem a uma derivação direta  $N \rightarrow \alpha_1\alpha_2\dots\alpha_n$ .
- **Vantagem:** a grande vantagem do uso de árvores de derivação reside no fato delas preservarem a *estrutura gramatical* da sentença gerada, o que torna muito mais simples a visualização e o acompanhamento de cada passo de derivação.
  - **Exemplo 3:** a árvore de derivação para a sentença **0011**, de acordo com a gramática

$$G_3 = (\{A\}, \{0, 1\}, \{A \rightarrow 0A1, A \rightarrow \epsilon\}, A)$$

é indicada pela Figura 5, a qual corresponde as derivações:

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 00\epsilon11 \Rightarrow 0011$$

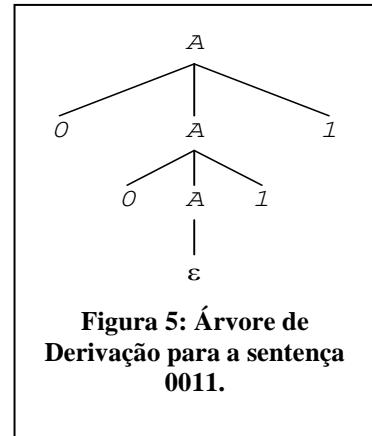
**Exemplo 4:** seja a gramática  $G_4 = (V_N, V_T, P, S)$ , onde:

$$V_N = \{ E \}$$

$$V_T = \{ 1, 2 \}$$

$$P = \{ E \rightarrow E + E \mid E^* E \mid 1 \mid 2 \}$$

$$S = E$$



Agora suponha a sentença  $1 + 2 * 2$ . Observe que existem duas árvores de derivação para a sentença, conforme indicado pela Figura 6 e Figura 7.

A árvore da Figura 6 representa uma **derivação à mais a direita**, enquanto a árvore da Figura 7 representa uma **derivação mais a esquerda**. Neste caso, dizemos que esta gramática é **ambígua**, isto é, possui mais de uma árvore de derivação para a mesma sentença.

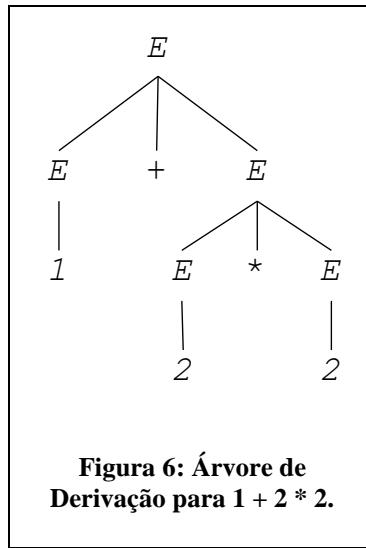


Figura 6: Árvore de Derivação para  $1 + 2 * 2$ .

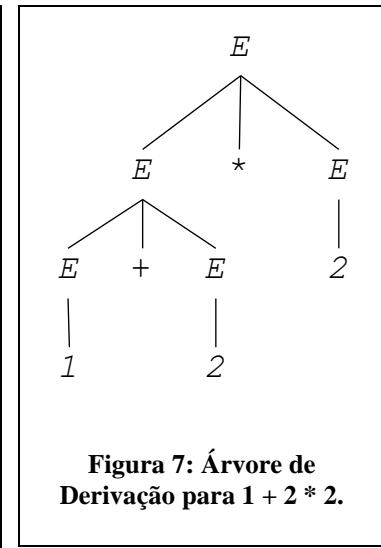


Figura 7: Árvore de Derivação para  $1 + 2 * 2$ .

- **Definição de Gramática Ambígua:** uma gramática  $G$  é dita ser uma **gramática ambígua** se, e somente se, existe uma sentença  $\omega \in L(G)$  que possua duas ou mais árvores de derivação.
- Visto que uma árvore de derivação corresponde a representação gráfica de uma derivação, então também podemos definir um gramática ambígua conforme indicado a seguir.
- **Definição de Gramática Ambígua:** uma gramática  $G$  é dita ser uma **gramática ambígua** se, e somente se, existe uma sentença  $\omega \in L(G)$  que possua mais de uma derivação à esquerda ou à direita.
- Quando uma gramática é ambígua, fica difícil decidir qual é a derivação correta para uma sentença. Por exemplo, as seguintes derivações são permitidas para a sentença  $1 + 2 * 2$ :
  - Derivações mais à esquerda:
 
$$\begin{aligned} E &\Rightarrow E + E \Rightarrow 1 + E \Rightarrow 1 + E * E \Rightarrow 1 + 2 * E \Rightarrow 1 + 2 * 2 \\ E &\Rightarrow E * E \Rightarrow E + E * E \Rightarrow 1 + E * E \Rightarrow 1 + 2 * E \Rightarrow 1 + 2 * 2 \end{aligned}$$
  - Derivações mais à direita:
 
$$\begin{aligned} E &\Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * 2 \Rightarrow E + 2 * 2 \Rightarrow 1 + 2 * 2 \\ E &\Rightarrow E * E \Rightarrow E * 2 \Rightarrow E + E * 2 \Rightarrow E + 2 * 2 \Rightarrow 1 + 2 * 2 \end{aligned}$$
- Pelas derivações acima fica difícil decidir qual caminho a seguir na geração da sentença  $1 + 2 * 2$ . Assim, para alguns casos é desejável que a ambigüidade de uma gramática seja eliminada. Isto é feito rescrevendo-se as produções da gramática de modo a criar uma gramática equivalente, mas sem ambigüidade. Entretanto, nem sempre isto é possível, pois a linguagem pode ser **inherentemente ambígua**.

- **Definição de Linguagem Inerentemente Ambígua:** uma linguagem é dita ser **inerentemente ambígua** se, e somente se, todas as gramáticas que a define são ambíguas.

A linguagem  $L = \{ a^n b^m c b^m a^n \mid n, m \geq 1 \}$  é um exemplo de linguagem inerentemente ambígua.

- Para o exemplo da gramática  $G_4$ , anteriormente descrita, seguindo as convenções matemáticas, verificamos que a árvore de derivação da Figura 6 é a correta, e não a da Figura 7. Deste modo, devemos rescrever as regras de produção de tal forma que a multiplicação seja avaliada antes da adição, o que é indicado pela nova gramática  $G_5 = (V_N, V_T, P, E)$ , onde:

$$V_N = \{ E \}$$

$$V_T = \{ 1, 2, +, * \}$$

$$P = \{ E \rightarrow E + T \mid T, T \rightarrow T * F \mid F, F \rightarrow 1 \mid 2 \}$$

Deve ser observado que  $G_5$  não é ambígua, pois só existe uma única derivação (esquerda e/ou direita) para a sentença  $1 + 2 * 2$ :

- Derivações mais à esquerda:

$$\begin{aligned} E &\Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow 1 + T \Rightarrow 1 + T * F \Rightarrow 1 + F * F \Rightarrow 1 + 2 * F \Rightarrow \\ &1 + 2 * 2 \end{aligned}$$

- Derivações mais à direita:

$$\begin{aligned} E &\Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * 2 \Rightarrow E + F * 2 \Rightarrow T + 2 * 2 \Rightarrow F + 2 * 2 \Rightarrow \\ &1 + 2 * 2 \end{aligned}$$

Entretanto,  $L(G_4) = L(G_5)$  e portanto  $G_5$  é equivalente a  $G_4$ .

## 2.4 Hierarquia de Chomsky

- Podemos classificar as gramáticas e as linguagens por elas geradas em quatro grandes grupos, de acordo com a *hierarquia de Chomsky*:
  - **Gramáticas Regulares**, ou do **Tipo 3**, sendo as linguagens por elas geradas denominadas de *linguagens regulares* ou *linguagens do tipo 3*;
  - **Gramáticas Livres de Contexto**, ou do **Tipo 2**, sendo as linguagens por elas geradas denominadas de *linguagens livres de contexto*, ou *linguagens do tipo 2*;
  - **Gramáticas Sensíveis ao Contexto**, ou do **Tipo 1**, sendo as linguagens por elas geradas denominadas de *linguagens sensíveis ao contexto*, ou *linguagens do tipo 1*; e as
  - **Gramáticas Irrestritas**, ou **Recursivamente Enumeráveis**, ou ainda, do **Tipo 0**, sendo as linguagens por elas geradas denominadas de *linguagens irrestritas*, *linguagens recursivamente enumeráveis*, ou *linguagens do tipo 0*.

- Esta hierarquia é definida em função da presença ou não de certas restrições, conforme analisado nas próximas seções.
- Deve ser salientado que todas as *linguagens regulares* estão contidas no universo das *linguagens livres de contexto*. Todas as *linguagens livres de contexto* estão contidas no universo das *linguagens sensíveis ao contexto*. E, todas as *linguagens sensíveis ao contexto* estão contidas no universo das *linguagens irrestritas*. A Figura 8 ilustra esta relação.
- **Definição - Gramáticas Regulares:** dada uma gramática  $\mathbf{G} = (V_N, V_T, P, S)$ , dizemos que esta gramática é *regular*, ou do *tipo 3*, se e somente se as produções de  $P$  estão na forma:

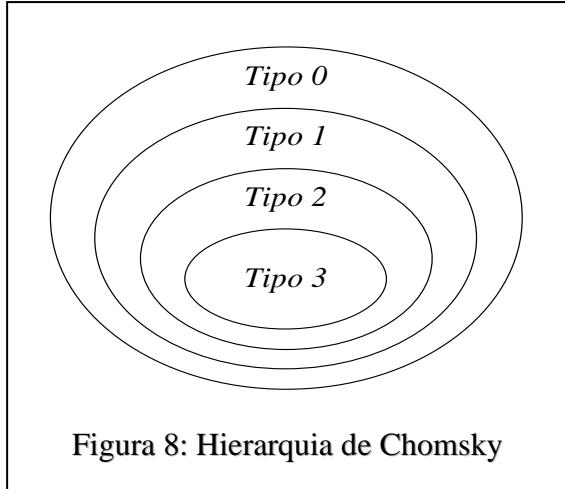


Figura 8: Hierarquia de Chomsky

$$S \rightarrow \alpha A \text{ ou } S \rightarrow \beta$$

onde  $S, A \in V_N$  e  $\alpha, \beta \in V_T$ , tal que  $\beta \neq \epsilon$ . De uma maneira mais geral, uma gramática é dita ser *regular* se e somente se suas produções possuem, do lado esquerdo, apenas um *não-terminal* e, do lado direito, um *terminal* seguido ou não de uma *não-terminal*.

- **Exemplo 5 – Gramática Regular:** seja a gramática  $\mathbf{G}_6 = (V_N, V_T, P, S)$ , onde

$$V_N = \{S, B\}$$

$$V_T = \{a, b\}$$

$$P = \{ S \rightarrow aS, S \rightarrow aB, B \rightarrow bB, B \rightarrow b \}$$

Esta gramática gera a linguagem

$$L(\mathbf{G}_6) = \{ a^n b^m \mid a, b \in V_T^*, S \Rightarrow a^* b^* \text{ e } n, m > 0 \}$$

Note que esta gramática gera sentenças do tipo

$$\{ ab, aab, abb, aaab, aabb, abbb, aaaab, aaabb, aabbb, abbbb, \dots \}$$

ou seja, todas as sentenças geradas devem conter pelo menos um **a** e um **b**.

- **Definição - Gramáticas Livres de Contexto:** dada uma gramática  $\mathbf{G} = (V_N, V_T, P, S)$ , dizemos que esta gramática é *livre de contexto*, ou do *tipo 2*, se e somente se as produções de  $P$  estão na forma:

$$S \rightarrow \alpha$$

onde  $S \in V_N$  e  $\alpha \in V^*$  (lembre-se que  $V = V_N \cup V_T$ ). De uma maneira mais geral, uma gramática é dita ser *livre de contexto* se e somente se suas produções possuem, do lado esquerdo, apenas um *não-terminal* e, do lado direito, *terminais* e/ou *não-terminais*, concatenados em qualquer ordem.

- **Exemplo 6 – Gramática Livre de Contexto:** seja a gramática  $G_7 = (V_N, V_T, P, S)$ , onde:

$$V_N = \{S, A, B\}$$

$$V_T = \{a, b\}$$

$$P = \{S \rightarrow aB, S \rightarrow bA, A \rightarrow a, A \rightarrow aS, A \rightarrow bAA, B \rightarrow b, B \rightarrow bS, B \rightarrow aBB\}$$

Esta gramática gera a linguagem

$$L(G_7) = \{a^n b^n \mid a, b \in V_T^*, S \Rightarrow a^* b^* \text{ e } n > 0\}$$

isto é, gera sentenças do tipo

$$\{ab, aabb, aaabbb, aaaabbbb, aaaaabbbbb, \dots\}$$

ou seja, todas as sentenças geradas possuem o mesmo número de **a**'s e **b**'s.

Observe que a gramática:

$$G_8 = (\{S\}, \{a, b\}, \{S \rightarrow aSb, S \rightarrow ab\}, S)$$

também gera sentenças que contêm o mesmo número de **a**'s e **b**'s. Logo a linguagem gerada por elas é a mesma, apesar das gramáticas serem diferentes. Neste caso

$$L(G_7) = L(G_8)$$

e, portanto,  $G_7$  é equivalente a  $G_8$ .

- **Exemplo 8 – Gramática Livre de Contexto:** outro exemplo de gramática livre de contexto é aquela que lida com expressões aritméticas, respeitando a precedência dos operadores. As regras de produção desta gramática são dadas por:

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow id \mid (E)$$

onde **id** representa qualquer número válido.

É importante observar que as gramáticas *livres de contexto* são mais úteis à implementação de compiladores do que as gramáticas *regulares*, já que estas últimas são mais restritas. Portanto, para a especificação de regras de sintaxe de uma linguagem são utilizadas gramáticas *livres de contexto*, enquanto que para o reconhecimento de sentenças são utilizadas gramáticas *regulares*.

Deve-se salientar ainda que as árvores de derivação geradas por gramáticas *regulares* são sempre compostas por derivações a direita, enquanto que gramáticas *livres de contexto* permitem derivações tanto a esquerda como a direita.

- **Definição - Gramáticas Sensíveis ao Contexto:** dada uma gramática  $G = (V_N, V_T, P, S)$ , dizemos que esta gramática é **sensível ao contexto**, ou do *tipo 1*, se e somente se as produções de  $P$  estão na forma:

$$\alpha \rightarrow \beta$$

onde  $\alpha \in V^*V_NV^*$ ,  $\beta \in V^*$  e  $|\alpha| \leq |\beta|$ . De uma maneira mais geral, uma gramática é dita ser *sensível ao contexto* se e somente se alguma de suas produções possui, do lado esquerdo, uma cadeia da forma  $\gamma A \delta$ , a qual permite a substituição do *não-terminal*  $A$  desde de que  $A$  esteja no contexto de  $\gamma$  e  $\delta$ . Isto é, que a sua esquerda ocorra um  $\gamma$  e a sua direita ocorra um  $\delta$ , sendo que  $\gamma, \delta \in V^*$  e  $A \in V_N$ . Daí o nome *sensível ao contexto*.

- **Exemplo 9 – Gramática Sensível ao Contexto:** seja a gramática  $G = (V_N, V_T, P, S)$ , onde:

$$V_N = \{S, A, B, C\}$$

$$V_T = \{a, b, c\}$$

$$P = \{S \rightarrow aSBC, S \rightarrow aBC, CB \rightarrow BC, aB \rightarrow ab, bB \rightarrow bb, bC \rightarrow bc, cC \rightarrow cc\}$$

Esta gramática gera a linguagem

$$L(G) = \{a^n b^n c^n \mid a, b, c \in V_T^*, S \Rightarrow a^* b^* c^* \text{ e } n > 0\}$$

- **Definição - Gramáticas Irrestritas:** dada uma gramática  $G = (V_N, V_T, P, S)$ , dizemos que esta gramática é *irrestrita*, ou do *tipo 0*, se e somente se nenhuma restrição é imposta às produções de  $P$ . Isto é:

$$\alpha \rightarrow \beta$$

onde  $\alpha \in V^*V_NV^*$ ,  $\beta \in V^*$ .

- **Exemplo 10 – Gramática Irrestrita:** seja a gramática  $G = (V_N, V_T, P, S)$ , onde:

$$V_N = \{S, A, B\}$$

$$V_T = \{a, b\}$$

$$P = \{S \rightarrow AB, AB \rightarrow BA, A \rightarrow a, B \rightarrow b\}$$

Esta gramática gera a linguagem  $L(G) = \{ab, ba\}$ .

## 2.5 Autômatos Finitos

- Verificamos nas seções anteriores como definir uma gramática capaz de gerar uma linguagem. Entretanto, conforme citado anteriormente, podemos definir uma linguagem através da especificação de mecanismos para o reconhecimento de sentenças nesta linguagem, denominados reconhecedores.
- A função destes mecanismos é submeter uma sentença da linguagem à um conjunto de regras de aceitação, as quais verificarão se a sentença pertence ou não à linguagem que elas definem.

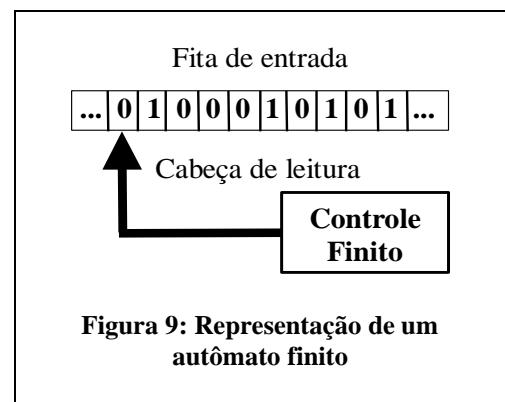
- A Tabela 1 ilustra os tipos de reconhecedores existentes, as linguagens por eles reconhecidas e a equivalência com as gramáticas que geram tais linguagens.
- As Máquinas de Turing e as Máquinas de Turing com Memória Limitada, não serão estudadas neste texto. O aluno, poderá encontrá-las em qualquer livro de Linguagens Formais e Autômatos.

Tabela 1: Tipos de reconhecedores

LINGUAGENS	GRAMÁTICAS	RECONHECEDORES
Irrestritas	Irrestritas	Máquinas de Turing
Sensíveis ao Contexto	Sensíveis ao Contexto	<i>Máquinas de Turing com memória limitada</i>
Livres de Contexto	Livres de Contexto	<i>Autômatos de Pilha</i>
Regulares	Regulares	<i>Autômatos Finitos</i>

## 2.5.1 Autômatos Finitos Determinísticos

- **Definição Formal:** um **autômato de estado finito**  $M$ , ou simplesmente **autômato finito**, é uma quíntupla ordenada  $M = (Q, \Sigma, \delta, q_0, F)$ , onde:
  - $Q$ : representa um conjunto finito e não vazio,  $Q \neq \emptyset$ , denominado **conjunto de estados**;
  - $\Sigma$ : representa um conjunto finito e não vazio,  $\Sigma \neq \emptyset$ , de todos os símbolos aceitos pelo autômato, denominado **alfabeto do autômato**;
  - $\delta$ : representa uma **função de transição** de estados do autômato, a qual mapeia  $Q \times \Sigma$  em  $Q$ , isto é  $(Q \times \Sigma) \rightarrow Q$ , ou seja, para cada par **(estado, símbolo de entrada)** esta função fornece um **novo estado**  $q \in Q$  para qual o autômato deverá mover-se, o qual é denominado **configuração de  $M$** ;
  - $q_0$ : representa o **estado inicial**, com  $q_0 \in Q$ , ou seja, o estado no qual o autômato deverá iniciar;
  - $F$ : representa um conjunto finito de **estados finais** ou **estados de aceitação**, com  $F \subseteq Q$ . Isto é, estando o autômato em um estado  $q$ ,  $q \in Q$ , e um  $\epsilon$  for lido (normalmente interpretado como o fim da sentença ou o fim do arquivo), o autômato deverá parar, sendo a sentença **aceita** pelo autômato.



- Graficamente, podemos representar um autômato finito pela Figura 9.
- Por exemplo, o autômato finito  $M_1 = (\mathbf{Q}, \Sigma, \delta, q_0, F)$  a seguir reconhece sentenças do tipo  $\{a, aba, ababa, abababa, \dots\}$ , isto é sentenças da linguagem  $L = \{a(ba)^n \mid a, b \in \Sigma, n \geq 0\}$ , onde:

$$\mathbf{Q} = \{1, 2\}$$

$$\Sigma = \{a, b\}$$

$$\delta = \{(1, a) = 2, (1, b) = 3, (2, a) = 3, (2, b) = 1, (3, a) = 3, (3, b) = 3\}$$

$$q_0 = 1$$

$$F = \{2\}$$

- O autômato  $M_1$  também pode ser visualizado pelo diagrama de transição da Figura 10.
- **Linguagem Aceita:** se  $A$  é o conjunto de todas as sentenças aceitas pelo autômato  $M$ , então dizemos que  $A$  é a linguagem reconhecida por este autômato, o qual denotamos por  $L(M) = A$ , isto é:  $L(M) = \{\omega \mid \delta(q_0, \omega) = F\}$ .
- O diagrama de transição da Figura 11 reconhece a linguagem  $L(M_2) = \{(ab)^n(a/\varepsilon) \mid n \geq 0\}$ , sendo  $M_2 = (\mathbf{Q}, \Sigma, \delta, q_0, F)$ , onde:

$$\mathbf{Q} = \{1, 2\}$$

$$\Sigma = \{a, b\}$$

$$\delta = \{(1, a) = 2, (2, b) = 1\}$$

$$q_0 = 1$$

$$F = \{1, 2\}$$

- Observe pela Figura 10 e Figura 11 que apesar de  $M_1$  e  $M_2$  parecerem iguais,  $L(M_1) \neq L(M_2)$ .
- Ambos os autômatos,  $M_1$  e  $M_2$ , são **determinísticos (AFD)**. Entretanto, podemos construir também autômatos finitos **não determinísticos (AFN)**, bastando para isto alterarmos um pouco a especificação formal.

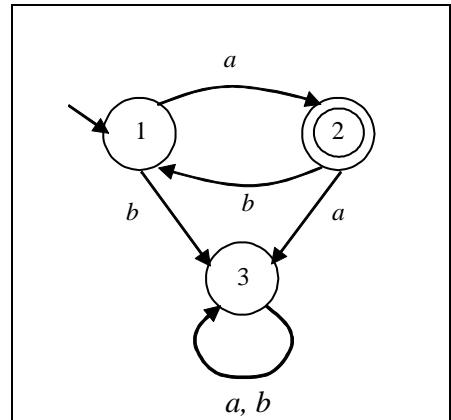


Figura 10: Diagrama de estados do autômato finito  $M_1$ .

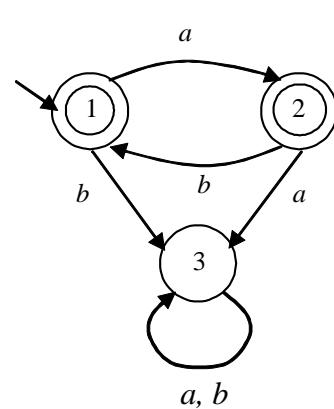


Figura 11: Diagrama de estados do autômato finito  $M_2$ .

## 2.5.2 Autômatos Finitos Não Determinísticos

- **Definição Formal de AFN:** um **autômato finito não determinístico (AFN)**  $M$ , é uma quíntupla ordenada  $M = (Q, \Sigma, \delta, q_0, F)$ , onde:
  - $Q, \Sigma, q_0$  e  $F$  são idênticos ao AFD;
  - $\delta$ : é a função de transição a qual mapeia  $Q \times (\Sigma \cup \{\epsilon\})^1$  em  $2^Q$ , isto é  $(Q \times (\Sigma \cup \{\epsilon\})) \rightarrow 2^Q$ , ou seja, para cada par **(estado, símbolo de entrada)** esta função fornece um **novo conjunto de estados**  $\{q_1, \dots, q_n\} \subseteq Q$  para qual o autômato poderá se mover.
- Basicamente, podemos dizer que um autômato é:
  - **Determinístico (AFD):** se a partir de qualquer **estado** e para **cada** um dos **símbolo de entrada** for possível **determinar** uma **única função de transição** a ser aplicada. Note, portanto, que a definição formal de  $\delta$  é bem definida.
  - **Não Determinístico (AFN):** se a partir de qualquer **estado** existir **mais de uma função de transição** possível de ser aplicada para o mesmo **símbolo de entrada**, ou existir transições vazias ou não definidas, conforme ilustrado pelos diagramas da Figura 12 e Figura 13.
- Os diagramas de estados indicados pela Figura 12 e Figura 13, correspondentes aos **AFNs**  $M_3$  e  $M_4$ , reconhecem a mesma linguagem, isto é,  $L = \{\omega \in \Sigma^* \mid \Sigma = \{a, b\} \text{ e } \omega \text{ é formada por zero ou mais combinações das cadeias } ab \text{ e } aba\}$ . Portanto,  $M_3$  é equivalente a  $M_4$ .
- Um **AFD**  $M_5$ , o qual também reconhece a linguagem  $L$  acima, é indicado pela Figura 14. Note que tanto  $M_3$  como  $M_4$  são mais facilmente obtidos do que  $M_5$ .

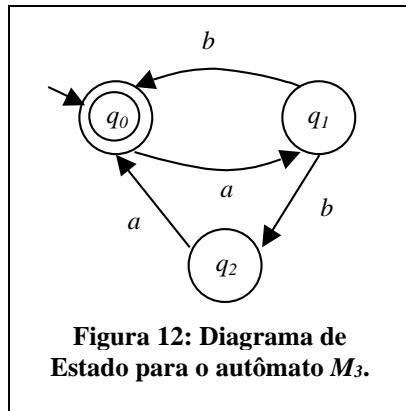


Figura 12: Diagrama de Estado para o autômato  $M_3$ .

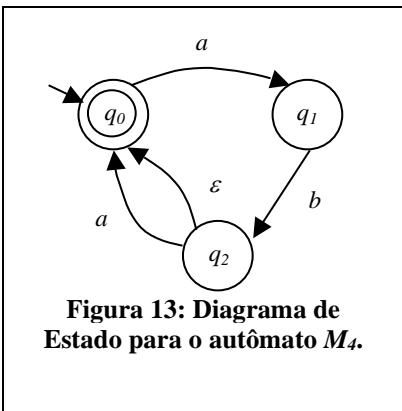


Figura 13: Diagrama de Estado para o autômato  $M_4$ .

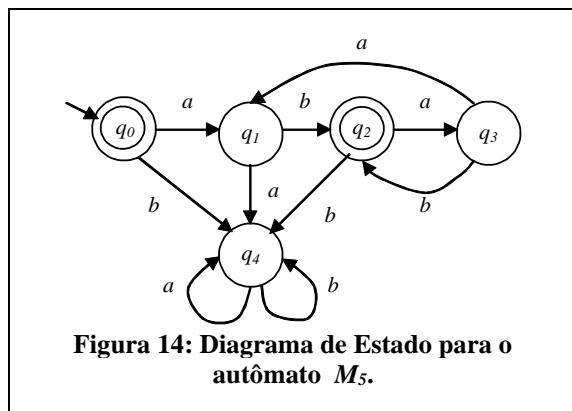


Figura 14: Diagrama de Estado para o autômato  $M_5$ .

<sup>1</sup>  $(\Sigma \cup \{\epsilon\})$  é também representado por  $\Sigma_\epsilon$ .

## 2.5.3 Implementação de Autômatos Finitos

- Um autômato finito pode ser facilmente implementado em qualquer linguagem de programação. A seguir analisamos uma implementação em *português estruturado* para um autômato que reconhece números *inteiros e reais*. O diagrama de transição é dado pela Figura 14, o qual corresponde ao autômato  $M_I = (Q, \Sigma, \delta, q_0, F)$ , onde:

$$Q = \{1, 2, 3, 4\}$$

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$$

$$\delta = \{(1, \text{digito}) = 2, (2, \text{digito}) = 2, (2, .) = 3, (3, \text{digito}) = 4, (4, \text{digito}) = 4\}$$

$$q_0 = 1$$

$$F = \{2, 4\}$$

- A função `Ler(Entrada)` presente no programa têm como objetivo ler um caractere da cadeia `Entrada`, retornando-o. A função `EOF(Entrada)` têm como objetivo verificar se a cadeia `Entrada` foi toda percorrida, retornando `Verdade`,

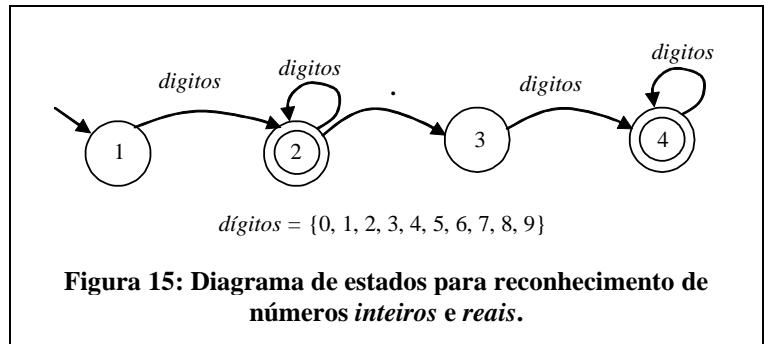


Figura 15: Diagrama de estados para reconhecimento de números *inteiros e reais*.

caso afirmativo, e `Falso`, caso contrário. Já a função `Mostre(Msg)` permite mostrar a mensagem dada por `Msg` na tela do computador.

```

Procedimento Automato (Declare Entrada como String)
Rótulos 1, 2, 3, 4, Fim, Erro
Dígito = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
Declare C como Caracter
Início
 1:   C <- Ler(Entrada)
        Se C for um Dígito, então vá para 2
        Senão vá para Erro
 2:   C <- Ler(Entrada)
        Se C for um Dígito, então vá para 2
        Senão
          Se C = '.', então vá para 3
          Senão Se EOF(Entrada) = Verdade, então vá para Fim
          Senão vá para Erro
 3:   C <- Ler(Entrada)
        Se C for um Dígito, então vá para 4
        Senão vá para Erro
 4:   C <- Ler(Entrada)
        Se C for um Dígito, então vá para 4
        Senão
          Se EOF(Entrada) = Verdade, então vá para Fim
          Senão vá para Erro

```

```

Fim: Mostre("Sentença reconhecida")
      Termine o programa
Erro: Mostre("Sentença não reconhecida")
Fim

```

- Existem outras formas de implementar este autômato sem se utilizar o comando **Vá Para** (conhecido na maioria das linguagens por **Goto**), o que para muitos fere a essência da programação estruturada. A seguir é ilustrado um programa estruturado.

```

Procedimento Automato(Declare Entrada como String)
Declare C como Caracter
      E como Inteiro
      Erro como Lógico
Início
  C <- Ler(Entrada)
  E <- 1;
  Erro <- Falso
  Enquanto NÃO EOF(Entrada) e NÃO Erro Faça
    Caso E seja
      1:   se C é um dígito, então E <- 2
            Senão Erro <- Verdade
      2:   se C é um dígito, então E <- 2
            Senão
                  Se C = '.', então E <- 3
                  Senão Erro <- Verdade
      3:   se C é um dígito, então E <- 4
            Senão Erro <- Verdade
      4:   se C é um dígito, então E <- 4
            Senão Erro <- Verdade
    Fim Caso
    C <- Ler(Entrada)
  Fim Enquanto
  Se Erro OU ((E <> 2) E (E <> 4)), então
    Mostre('Sentença não reconhecida')
  Senão Mostre('Sentença reconhecida')
Fim

```

## 2.5.4 Tabelas de Transição para Autômatos Finitos

- Uma forma alternativa de representação de autômatos finitos é feita pelo uso de *tabelas de transição*.
- Tais tabelas relacionam em suas *linhas* todas as possíveis transições efetuadas por todos os símbolos reconhecidos pelo autômato. Enquanto que cada *coluna* representa o *estado atual*, o *símbolo aceito* pelo estado, o *novo estado* para o qual o autômato deverá se mover após reconhecer tal símbolo e um *índice* para a linha da tabela onde se inicia a relação de entradas para o novo estado.
- A Tabela 2 ilustra uma tabela de transição para o diagrama da Figura 14, passível de ser implementada em um computador.

Tabela 2: Tabela de transição para reconhecimento de números

Linha	Estado	Símbolo Atual	Próximo Estado	Próximo Linha
1	1	0	2	11
2	1	1	2	11
3	1	2	2	11
4	1	3	2	11
5	1	4	2	11
6	1	5	2	11
7	1	6	2	11
8	1	7	2	11
9	1	8	2	11
10	1	9	2	11
11	2	0	2	11
12	2	1	2	11
13	2	2	2	11
14	2	3	2	11
15	2	4	2	11
16	2	5	2	11
17	2	6	2	11
18	2	7	2	11
19	2	8	2	11
20	2	9	2	11
21	2	.	3	22
22	3	0	4	32
23	3	1	4	32
24	3	2	4	32
25	3	3	4	32
26	3	4	4	32
27	3	5	4	32
28	3	6	4	32
29	3	7	4	32
30	3	8	4	32
31	3	9	4	32
32	4	0	4	32
33	4	1	4	32
34	4	2	4	32
35	4	3	4	32
36	4	4	4	32
37	4	5	4	32
38	4	6	4	32
39	4	7	4	32
40	4	8	4	32
41	4	9	4	32

- Para implementarmos esta tabela, em Pascal, por exemplo, primeiramente devemos declarar sua estrutura, que pode ser do tipo a seguir, assumindo que o número de transições seja dado pela constante `NoTransicao`:

```
Type
  TTab = record of
    Estado: integer;
    Simbolo: char;
    Proximo: integer;
    Indice: integer;
  End;
  TTabela = array[1..NoTransicao] of TTab;
```

- Agora basta declararmos a tabela conforme indicado a seguir:

```
Var
  Tabela: TTabela;
```

- Outro cuidado a ser tomado é a declaração de um array contendo todos os estados finais. Isto pode ser feito da seguinte forma, assumindo que o número de estados finais seja dado pela constante `NoFinais`:

```
Type
  TFinais = array[1..NoFinais] of integer;
Var
  Finais: TFinais;
```

- Um exemplo desta tabela para o diagrama da Figura 14 é dado abaixo:

Linha	Estado Final
1	2
2	4

- A partir da tabela de transição, a qual deverá ser previamente preenchida, tomando-se o cuidado de sua primeira linha ser o estado inicial, podemos escrever o seguinte procedimento, o qual verificará se uma sentença é aceita ou não pelo autômato que ela define:

```
Procedure Automato( Tabela: TTabela; // Tabela Transição
                    Finais: TFinais; // Estados Finais
                    Entrada: String); // Sentença
Var
  E: integer; // Estado atual
  I: integer; // Próxima linha
  C: Char; // Símbolo lido
  Erro: Boolean; // Indica a ocorrência de um erro
Begin
  I := 1;
  E := Tabela[I].Estado;
  C := Ler(Entrada);
  Erro := False;
  While not EOF(Entrada) and not Erro do Begin
    While true do begin
```

```

If      C = Tabela[I].Simbolo then
Begin
    E := Tabela[I].Proximo;
    I := Tabela[I].Indice;
    Break;
End
Else
    I := I + 1;
    If      (I > NoTransicao) or
            (E <> Tabela[I].Estado) then
    Begin
        Erro := True;
        Break;
    End;
End;
C := Ler(Entrada);
End;
If      Erro or not Final(Finais, E) then
    write('Sentença não reconhecida')
else write('Sentença reconhecida');
End;

```

- Neste programa, as tabela de transição, *Tabela*, e de estados finais, *Finais*, é recebida como parâmetro, devendo, portanto, terem sido montadas anteriormente. A função *Final(Finais, E)* verifica se o estado dado por *E* é um estado final ou não, retornando *True* ou *False*, respectivamente.
- Obviamente que poderíamos construir uma tabela utilizando uma lista de estados de transição, onde cada entrada na tabela corresponderia a um estado com suas correspondentes transições. Tal estrutura poderia ser representada por:

```

Type
  PTab =      ^TTab;
  TTab =      record of
    Simbolo: char;
    Estado: integer;
    Next: PTab;
  End;
TTabela = array[1..NoEstados] of PTab;

```

- Neste caso, *Simbolo* seria o símbolo a ser reconhecido pelo estado, *Estado* seria o novo estado para a transição e *Next* seria um ponteiro para o próximo símbolo reconhecido pelo estado atual. Obviamente, *TTabela* seria uma tabela com um ponteiro para uma lista de símbolos reconhecidos pelo estado. Novamente, o estado inicial corresponderia a primeira linha da tabela.
- A
- Tabela 3 abaixo ilustra uma configuração para o digrama de estados da Figura 14. A Figura 16 ilustra a configuração de memória para esta tabela.

**Tabela 3: Tabela de ponteiros para estados de transição**

**Prof. Rogério Melo Nepomuceno**

para o reconhecimento de números

Linha/Estado	Símbolos Reconhecidos	Próximo Estado
1	0..9	2
2	0..9	2
	.	3
3	0..9	4
4	0..9	4

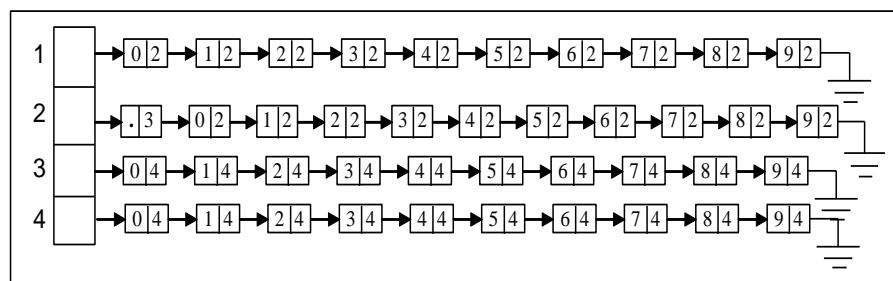


Figura 16: Configuração da memória para uma tabela de transição

## 3 Análise Léxica

### 3.1 Funções da Análise Léxica

O **analisador léxico** é a parte do compilador responsável por ler cada um dos elementos da sentença de entrada, neste caso, o programa fonte, e transformá-los em uma representação conveniente para o **analisador sintático**: uma sequência finita de **itens léxicos**.

Um **item léxico**, o qual normalmente é chamado também de **Token**, é uma unidade básica do texto correspondente ao programa fonte, sendo, normalmente, representado internamente pelo analisador léxico por três informações:

- **Classe**: classificação léxica do **token** – *identificador* (sentença que se inicia por uma letra seguida por qualquer combinação de letras e dígitos), *constante inteira*, *constante real*, *palavra reservada*, *operador aritmético*, *símbolo*, *cadeia de caracteres* (cadeia de caracteres delimitada por aspas ou apóstrofe), dentre outros.
- **Valor - Lexema**: corresponde ao valor léxico do **token**, o qual depende da classe. Pode ser de dois tipos:
  - **Tokens Simples**: são tokens que não possuem argumentos, pois a sua classe já o define completamente. São exemplos: operadores matemáticos, relacionais, lógicos e de atribuição, símbolos.
  - **Tokens com Argumento**: são tokens que possuem um valor associado. Correspondem aos elementos da linguagem definidos pelo programador, como por exemplo, os identificadores, as constantes numéricas e cadeias de caracteres. Por exemplo, para um token do tipo constante numérica, o valor pode ser o número inteiro representado pela constante, enquanto que para identificadores, o valor pode ser a sequência de caracteres ou um ponteiro para uma **tabela de símbolos**.
- **Posição**: identifica a localização do token no programa fonte, o que constitui uma informação importante para o processo de correção de erros.

**Exemplo:** seja a gramática  $G_1 = (V_N, V_T, P, E)$ , onde:

$$V_N = \{E, T, F, I, N\}$$

$$V_T = \{+, -, *, /, (,), 0..9, a..z, A..Z\}$$

$$\begin{aligned}P = \{ & \quad E \rightarrow E + T \mid E - T \mid T \\& \quad T \rightarrow T * F \mid T / F \mid F \\& \quad F \rightarrow I \mid N \mid (E) \\& \quad I \rightarrow Ia \mid ... \mid Iz \mid I0 \mid ... \mid I9/a \mid ... \mid z \\& \quad N \rightarrow N0 \mid ... \mid N9 \mid 0 \mid ... \mid 9\}\end{aligned}$$

Observe pelo exemplo anterior que a produção  $I$  permite construir sentenças que se iniciam por uma letra seguida por qualquer combinação de letras e dígitos – os **identificadores**. A produção  $N$  permite a construção de sentenças compostas por qualquer combinação de dígitos – as **constantes inteiras**. Por exemplo, são **identificadores** as seguintes cadeias: X, Código, B54, C1A, C2a5, etc. E são **constantes inteiras** as seguintes cadeias: 1, 0800, 1054, 125, etc.

Se abstrairmos os *identificadores* pela palavra ***id*** e os *números inteiros* pela palavra ***num***, então podemos reescrever a gramática acima como:

$$G_2 = (V_N, V_T, P, S)$$

onde:

$$V_N = \{E, T, F\}$$

$$V_T = \{+, -, *, /, (,), \text{id}, \text{num}\}$$

$$S = \{E\}$$

$$\begin{aligned}P = & \{ \quad E \rightarrow E + T \mid E - T \mid T \\& \quad T \rightarrow T * F \mid T / F \mid F \\& \quad F \rightarrow \text{id} \mid \text{num} \mid (E) \}\end{aligned}$$

Observando esta gramática, concluímos que ela nos permite construir expressões aritméticas, com ou sem parênteses, envolvendo as operações de soma, subtração, multiplicação e divisão entre operandos denominados genericamente de ***id*** (*identificadores*) e ***num*** (*constantes inteiras*). Estas expressões são normalmente encontradas na maioria das linguagens de programação.

Portanto, por esta gramática podemos evidenciar os seguintes ***tokens***: operadores aritméticos e parênteses (tokens simples), ***id*** e ***num*** (tokens com argumento).

## 3.2 Funcionamento do Analisador Léxico

O analisador léxico é normalmente visualizado conceitualmente como uma fase do compilador diferente da análise sintática. Isto traz certas vantagens, como:

1. Simplificação e modularização do projeto do compilador;
2. Os ***tokens*** podem ser descritos utilizando-se notações simples, tais como expressões regulares, enquanto que a estrutura sintática de comandos e expressões das linguagens de programação requer uma notação mais expressiva, como as linguagens livres de contexto; e
3. Os reconhecedores construídos a partir da descrição dos ***tokens*** (através de expressões regulares, autômatos finitos e gramáticas regulares) são mais eficientes e compactos do que os reconhecedores construídos a partir de gramáticas livres de contexto.

Entretanto, em geral, o **analisador léxico** acaba sendo tratado como uma rotina assessoria do **analisador sintático**, o qual é chamado sempre que houver a necessidade de um novo **token**. Desta forma, a cada chamada da rotina de análise léxica, uma cadeia de caracteres é lida e interpretada, sendo devolvido o seu **token** correspondente.

**Exemplo:** para a sentença de entrada **A \* (C1a + 25)**, o **analisador léxico** devolveria os seguintes **tokens** a cada chamada do analisador sintático:

Chamada	Cadeia lida (Lexema)	Classe léxica
1 <sup>a</sup>	<b>A</b>	<i>Identificador</i>
2 <sup>a</sup>	*	<i>Operador de Multiplicação</i>
3 <sup>a</sup>	(	<i>Parêntese Esquerdo</i>
4 <sup>a</sup>	<b>C1a</b>	<i>Identificador</i>
5 <sup>a</sup>	+	<i>Operador de Adição</i>
6 <sup>a</sup>	<b>25</b>	<i>Constante Numérica</i>
7 <sup>a</sup>	)	<i>Parêntese Direito</i>

Convencionando que as classes são representadas pelos tipos **cId** (identificador), **cNum** (constante inteira), **cAdd** (operador de adição), **cSub** (operador de subtração), **cMul** (operador de multiplicação), **cDiv** (operador de Divisão), **cLPar** (parêntese esquerdo) e **cDPar** (parêntese direito), ao ser chamado o analisador léxico poderia devolver os seguintes **tokens**:

Chamada	Classe léxica	Valor (ou Lexema)
1 <sup>a</sup>	<b>cId</b>	'A'
2 <sup>a</sup>	<b>cMul</b>	
3 <sup>a</sup>	<b>cLPar</b>	
4 <sup>a</sup>	<b>cId</b>	'C1a'
5 <sup>a</sup>	<b>cAdd</b>	
6 <sup>a</sup>	<b>cNum</b>	<b>25</b>
7 <sup>a</sup>	<b>cDPar</b>	

É importante observar que uma classe léxica pode também representar vários itens léxicos e não apenas um. Este artifício é normalmente utilizado, pois pode facilitar a construção das regras sintáticas para a linguagem a ser analisada. Por exemplo, se nossa gramática suportasse os operadores relacionais = (igual), < (menor), <= (menor ou igual), > (maior), >= (maior ou igual) e <> (diferente), então poderíamos identificá-los apenas pela classe **cRel**. Usando esta abordagem, o **token** deve fornecer informações adicionais sobre qual operador relacional está sendo analisado. Isto pode ser feito, por exemplo, atribuindo-se ao campo **Valor** do **token** a

classe correspondente do item léxico. Por exemplo, se definirmos as classes **cIgual**, **cMenor**, **cMenorIgual**, **cMaior**, **cMaiorIgual** e **cDiferente** para os operadores relacionais e o operador “ $\leq$ ” estiver sendo analisado, então poderíamos retornar ao analisador sintático a tupla (*Classe, Valor*) correspondente ao *token*, que nesse caso seria (**cOpRel**, **cMenorIgual**). Entretanto, caso você decida identificar cada operador relacional apenas por sua classe correspondente, então você terá que definir regras gramaticais individuais para tratar de cada operador em separado.

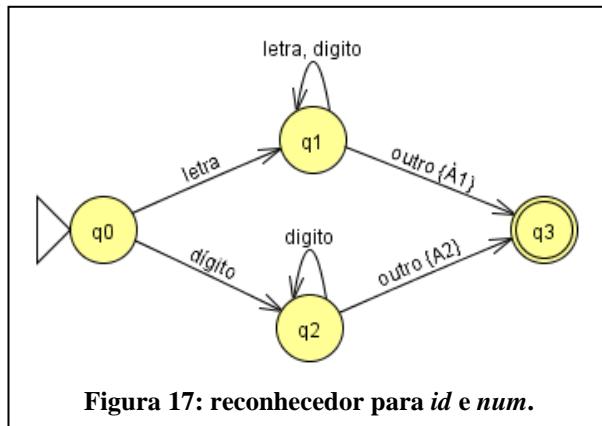
Após ter definido quais são os ***tokens*** que constituem a linguagem a ser tratada, a implementação do analisador léxico pode ser iniciada. Podemos implementá-lo de dois modos: construindo um programa manualmente, em uma linguagem de programação apropriada, que simule um autômato finito ou o reconhecimento de padrões regulares; ou utilizando um gerador de analisadores léxicos, como o LEX, por exemplo.

### 3.3 Autômatos Finitos

Suponha a gramática **G<sub>2</sub>**, definida anteriormente. Nela temos os terminais ***id*** e ***num***.

De acordo com o que vimos, um ***id*** é uma sequência de letras e dígitos, devendo iniciar com uma letra; enquanto um ***num*** é um dígito seguido por zero ou mais dígitos.

Um autômato finito que reconhece estes dois tipos de itens léxicos é o indicado pela Figura 17.



As seguintes considerações são necessárias ao se analisar a Figura 17:

1. A presença da palavra **outro**, nas transições que ligam os estados (q1, q3) e (q2, q3), deve ser entendida como qualquer outro símbolo do alfabeto que não seja o reconhecido pelo estado de origem. Sua presença serve para indicar que ao se ler qualquer outro símbolo que não os esperados, o item léxico em questão deve ser reconhecido, devendo seu ***token*** ser retornado.

Portanto, como outro indica que um símbolo foi lido da entrada, então ele deve ser recolocado de volta na entrada para que ele seja lido na próxima vez que o autômato for chamado. Tal estratégia é um artifício muito utilizado para facilitar a especificação de autômatos reconhecedores.

2. {A1} e {A2} correspondem a ações que deverão ser executadas quando a transição em questão for aplicada. No caso de {A1} a ação correspondente é `retorne(cId, InsTabSim(Lexema))`. Já para {A2} teríamos `retorne(cNum, Int(Lexema))`.

A implementação de um programa que simule o autômato da Figura 17 pode ser feita a partir do algoritmo descrito a seguir. O tipo **TToken** é composto pelos campos *Classe* (do tipo **TClasse**) e *Valor* (do tipo **TValor**). O tipo **TClasse** é um tipo enumerado composto por *cId* (identificador), *cNum* (número) e *cEOF* (fim da entrada), enquanto o tipo **TValor** é um tipo polimórfico que pode conter um valor inteiro ou ponteiro para um endereço da tabela de símbolos. A operação `Lexema = Lexema + C` indica uma concatenação,

```
Procedimento Reconhecedor (Declare Entrada como String)
  Declare
    C como Caracter
    E como Inteiro
    Lexema como string
    Token como TToken
  Inicio
    E = 0
    Lexema = ""
    C = Ler(Entrada)
    Enquanto NÃO EOF(Entrada) Faça
      Caso E seja
        0:   se C é uma LETRA, então
              Lexema = Lexema + C
              C = Ler(Entrada)
              E = 1
      Senão
        se C é um DIGITO, então
          Lexema = Lexema + C
          C = Ler(Entrada)
          E = 2
      Senão
        EmitirMensagem("Erro: ", C)
        C = Ler(Entrada)
      Fim Se
    Fim Se
```

```
1:   se C é uma LETRA ou um DIGITO, então
      Lexema = Lexema + C
      C = Ler(Entrada)
      E = 1
      Senão
          Token.Classe = cId
          Token.Valor = InstabSim(Lexema)
          E = 3
      Fim Se
2:   se C é um DIGITO, então
      Lexema = Lexema + C
      C = Ler(Entrada)
      E = 2
      Senão
          Token.Classe = cNum
          Token.Valor = StrToInt(Lexema)
          E = 3
      Fim Se
3:   Retroceder(Entrada)
   Retorne Token
   Fim Caso
Fim Enquanto
Token.Classe = cEOF
Retorne Token
Fim
Fim Algoritmo
```

### 3.3.1 Exercício:

- a) Suponha que desejemos reconhecer sentenças de uma linguagem  $L_1$ , formada apenas por identificadores, números inteiros, números reais e números em notação científica, como por exemplo: a, X20, Código, 25, 14.786, 3E5, 768E+9 e 1.573E-7. Construa um autômato finito que reconheça sentenças desta linguagem.
- b) De posse do autômato finito da questão anterior, o qual reconhece sentenças de uma linguagem  $L_1$ , implemente um programa em Java capaz de varrer uma *string* de entrada e reconhecer todas as possíveis sentenças de  $L_1$ . Caso uma sentença não pertença a  $L_1$ , emita uma mensagem de erro apropriada e continue o processo de reconhecimento para as próximas sentenças.
- c) Suponha que desejemos reconhecer sentenças de uma linguagem  $L_2$ , a qual é composta por todas as sentenças de  $L_1$ , mais os operadores relacionais =, <, <=, >, >= e <>. Construa um autômato finito que reconheça sentenças desta linguagem.

- d) De posse do autômato finito da questão anterior, o qual reconhece sentenças de uma linguagem  $L_2$ , implemente um programa em Java capaz varrer uma *string* de entrada e identificar todas as possíveis sentenças de  $L_2$ . Caso uma sentença não pertença a  $L_2$ , emita uma mensagem de erro apropriada e continue o processo de reconhecimento para as próximas sentenças.
- e) (Trabalho) Desenvolva em Java um programa que permita abrir um arquivo texto e contar o número de palavras, de caracteres, de identificadores, de números (inteiros e reais), de operadores (relacionais e aritméticos) e o número de linhas. Além dos itens anteriores o programa deverá criar um índice alfabético para todos os identificadores encontrados no texto, indicando a quantidade de vezes que ele ocorre, a linha onde ele ocorre e quantas vezes ele ocorre em cada linha. Ao final, o programa deverá imprimir todas as informações coletadas. Por exemplo, suponha que o arquivo aberto contenha o texto a seguir:

```
1 Funcao Fatorial(Declare n como Inteiro) Retorne Inteiro
2 Inicio
3     Se (n = 0 ou n = 1)
4         Retorne 1
5     Fim Se
6     Retorne n * Fatorial(n - 1)
7 Fim
```

As informações a serem impressas após o processamento do arquivo estão indicadas a seguir. A impressão destes resultados pelo programa deverá seguir o formato apresentado pelo exemplo (não deve haver distinção entre letras maiúsculas e minúsculas):

Estatística:

Quantidade de caracteres (incluindo espaço): 141  
Quantidade de caracteres (sem considerar o espaço): 105  
Quantidade de palavras: 26  
Quantidade de identificadores: 13  
Quantidade de números (inteiros e reais): 4  
Quantidade de operadores (relacionais e aritméticos): 4  
Quantidade de linhas: 7

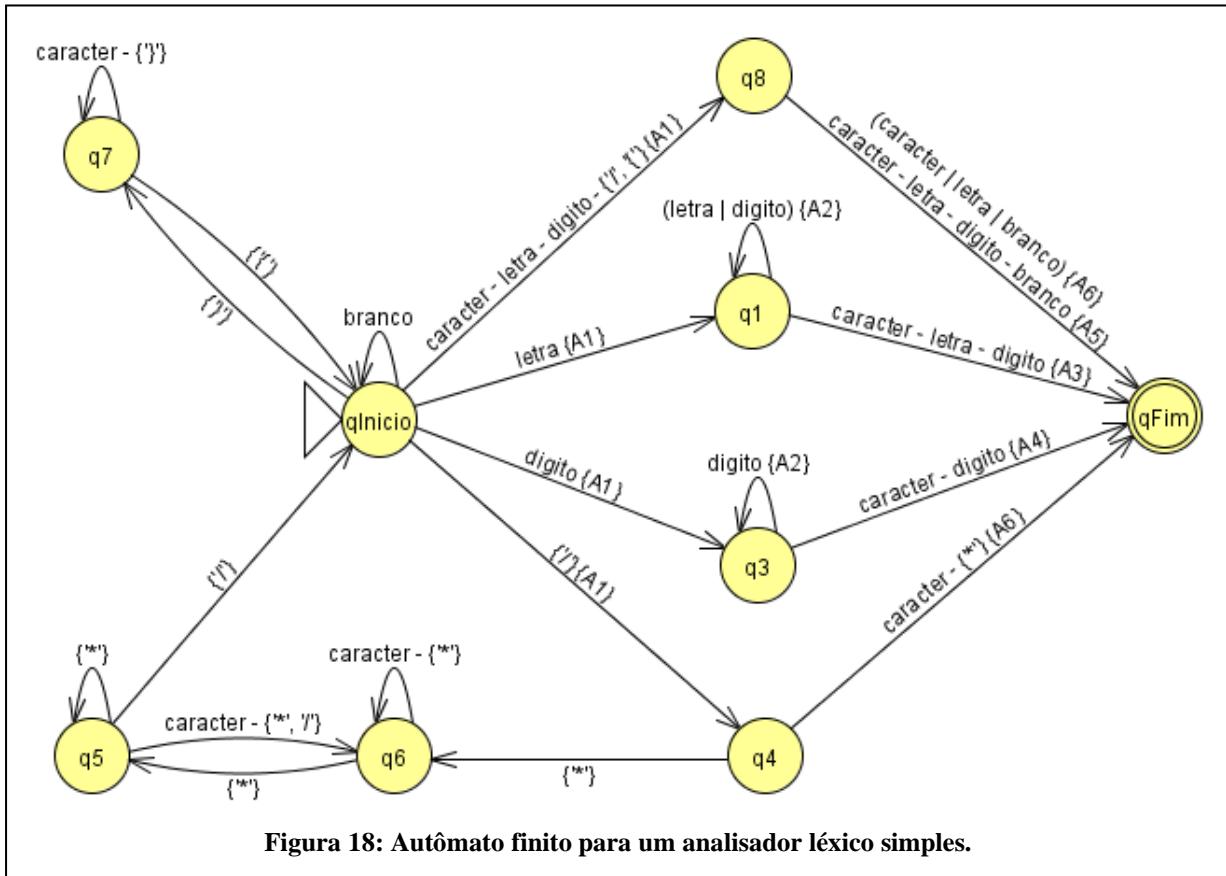
Índice Alfabético:

C  
 Como (1): 1(1)  
D  
 Declare (1): 1(1)  
F  
 Fatorial (2): 1(1), 6(1)  
 Fim (2): 5(1), 7(1)

```
  Funcao (1): 1(1)
I   Inicio (1): 2(1)
    Inteiro (2): 5(1), 7(1)
N   N (5): 1(1), 3(2), 6(2)
R   Retorne (3): 1(1), 4(1), 6(1)
S   Se (2): 3(1), 5(1)
```

□

Na Figura 18 encontra-se indicado um diagrama de estados correspondente a um *autômato finito* para reconhecimento dos identificadores, números e demais símbolos utilizados pela maioria das linguagens de programação. No diagrama os itens rotulados por  $A_i$  correspondem às ações desempenhadas pelo analisador léxico, como por exemplo, a identificação de **palavras reservadas**, busca e inserção de identificadores na **tabela de símbolos**, dentre outras ações que se julgue necessárias.



## 3.4 Expressões Regulares

Toda *linguagem regular* pode ser descrita por uma expressão *regular*, a qual é definida a partir de operações *regulares* sobre conjuntos.

Uma expressão regular **R** sobre um alfabeto  $\Sigma$ , a qual representa a linguagem  $L(R)$ , é indutivamente definida por:

1.  $\emptyset$  é uma expressão regular e denota a *linguagem vazia*;
2.  $\epsilon$  é uma expressão regular, a qual denota a linguagem contendo exclusivamente a palavra vazia, ou seja,  $\{\epsilon\}$ ;
3. Qualquer símbolo  $a \in \Sigma$  é uma expressão regular, a qual denota a linguagem contendo a palavra unitária  $a$ , ou seja,  $\{a\}$ ;
4. Se **R** é uma expressão regular, então  $(R)$  também é uma expressão regular;
5. Se **R**<sub>1</sub> e **R**<sub>2</sub> são expressões regulares, então  $(R_1 \cup R_2)$  é uma expressão regular – sendo também denotada por  $(R_1 + R_2)$  ou  $(R_1 | R_2)$  – a qual é denominada de operação de união;

6. Se  $R_1$  e  $R_2$  são expressões regulares, então  $(R_1R_2)$  é uma expressão regular – a qual é denominada de operação de concatenação;
7. Se  $R$  é uma expressão regular, então  $(R^*)$  é uma expressão regular – a qual é denominada de operação estrela de Kleene.

**Precedência dos Operadores Regulares:** os parênteses das expressões regulares podem ser removidos, desde que respeitado as precedências:

- 1º. Estrela de Kleene;
- 2º. Concatenação;
- 3º. União.

**Exemplo:** as expressões regulares a seguir geram a classe de números inteiros, números reais e identificadores, onde letra = {a,...,z} e digito = {0,...,9}.

- **Identificadores:** letra (letra | digito)\*
- **Números Inteiros:** digito digito\*
- **Números Reais:** digito digito\* . digito digito\*

A seguir são ilustrados exemplos da implementação de um reconhecedor para números inteiros e reais e um reconhecedor para identificadores, utilizando como modelo as expressões regulares indicadas pelo exemplo anterior.

Nos exemplos, CARACTER corresponde a um símbolo do alfabeto da linguagem a ser reconhecida. A expressão “LER NOVO CARACTER” corresponde a ler um símbolo da entrada que está sendo analisada. A classe *TToken* é composta pelos mesmos campos do exemplo de implementação do autômato (isto é, *Classe* e *Valor*), exceto que agora o tipo *TClasse* é composto pelos itens enumerados *cId* (identificador), *cInt* (número inteiro), *cReal* (número real) e *cEOF* (fim da entrada); e o tipo *TValor* pode conter um valor inteiro, um valor real ou ponteiro para um endereço da tabela de símbolos. A operação *Lexema* = *Lexema* + CARACTER indica uma concatenação, enquanto a operação *Token.Valor* = *Valor(Lexema)* indica a conversão do tipo *string* *Lexema* para o seu correspondente *inteiro* ou *real*, dependendo da classe especificada. Além disso, para fins de implementação, é considerado que a variável *Token* (do tipo *TToken*) e *Lexema* (do tipo *string*) sejam públicas ao programa. Outra coisa importante a ser mencionada é que antes de uma rotina ser chamada, a rotina ancestral deverá já ter lido um símbolo da entrada para a variável CARACTER.

## 3.5 Reconhecimento de Números

Os tipos de números podem ser inteiros ou reais, onde ambos serão cadeias de dígitos, sendo que os reais conterão o ponto decimal em algum lugar da cadeia. Assim, serão números as seguintes cadeias: 1, 1054, 5.25, 347.8, etc.

A implementação para este reconhecedor é indicada a seguir, onde as classes **cInt** e **cReal** representam números inteiros e reais, respectivamente.

```
Rotina Número()
  Início
    Se CARACTER é um DÍGITO então faça
      Lexema = ''
      Enquanto CARACTER é um DÍGITO faça
        Lexema = Lexema + CARACTER
        LER um novo CARACTER
      Fim Faça
      Token.Classe = cInt
      Se CARACTER é um PONTO DECIMAL então faça
        Lexema = Lexema + CARACTER
        LER novo CARACTER
        Se CARACTER é um DÍGITO então
          Enquanto CARACTER é um DÍGITO faça
            Lexema = Lexema + CARACTER
            LER novo CARACTER
          Fim faça
        Senão
          Ocorreu um Erro Léxico - Falta parte decimal
          Lexema = Lexema + '0'
        Fim Se
        Token.Classe = cReal
      Fim Se
      Token.Valor = Valor(Lexema)
    Fim Se
  Fim
```

A rotina acima lerá a sentença de entrada, concatenando caractere por caractere na variável Lexema, desde que estes sejam dígitos ou o ponto. A primeira etapa da rotina pára quando for detectado um caractere distinto de um dígito. Se este caractere for um ponto decimal, ele será concatenado em Lexema, juntamente com os próximos possíveis dígitos. A ausência de caracteres na sequência de entrada é tomada como uma sequência de caracteres do tipo *espaço em branco*, portanto, a rotina acima parará também quando encontrar um espaço em branco.

Um reconhecedor completo para números deverá reconhecer números reais e inteiros, precedidos ou não de sinal, e números em representação científica (na forma exponencial).

## 3.5.1 Exercício:

Construa uma rotina de análise léxica que reconheça números inteiros, números reais e números em notação científica. Por exemplo, os seguintes tipos de números deverão ser reconhecidos: 15, 2.57, 7E3, 2E+5, 8.5E7, 9.3E-5.

□

## 3.6 Reconhecimento de Identificadores

Como citado anteriormente, *identificadores* são cadeias de letras e dígitos desde que começando por uma letra.

Uma vez que não permitimos que outro caractere que não seja letra ou dígito componha um identificador, nosso reconhecedor ficará mais simples, como será visto a seguir.

```
Rotina Identificador()
  Inicio
    Se CARACTER é uma LETRA então faça
      Lexema = ''
      Enquanto CARACTER é uma LETRA ou um DÍGITO faça
        Lexema = Lexema + CARACTER
        LER novo CARACTER
      Fim Faça
      Se Lexema é uma Palavra Reservada então
        Token.Classe = Classe_da_Palavra_Reservada
      Senão
        Token.Classe = cId
      Fim se
      Se Lexema não está na Tabela de Símbolos então
        Inseri-lo na Tabela, juntamente com sua Classe
      Fim se
      Token.Valor = Endereço do item na Tabela de Símbolos
    Fim Se
  Fim
```

Esta rotina ficará lendo os caracteres da sentença de entrada, um a um, até que encontre um caractere que não seja uma letra ou um dígito. Note que, tanto esta rotina como aquela que reconhece os números termina com um caractere que não foi usado, assim, devemos tomar o cuidado para que os retornos das rotinas sejam compatíveis com as saídas dos mesmos.

É comum, nas linguagens de programação, a existência de identificadores que têm o seu uso reservado para controlar as ações do compilador - são as **palavras reservadas** (*keywords*). Esses identificadores devem ser encarados pelo analisador léxico como símbolos próprios da gramática. Portanto, o que normalmente se faz é consultar uma **Tabela de Palavras Reservadas** para verificar se o identificador recentemente reconhecido não é, na verdade, uma *palavra reservada* que, por isso, terá um código interno próprio. Por exemplo, no *Pascal*, as seguintes palavras são

reservadas: *const, type, var, begin, end, while, do, for, downto, if, then, else, case, of, array, function, procedure, label, record, exit, break, continue*.

Outro problema similar é o de diferenciar identificadores já definidos, entre eles os identificadores de nomes de variáveis inteiras, reais, booleanas, complexas, etc., dos identificadores não definidos. É fácil notar que o código interno para um identificador de variável inteira deve ser diferente do de uma variável real, e assim por diante. Essa distinção pode ser feita através da consulta a uma tabela, denominada **Tabela de Símbolos**, onde estarão as informações, nos chamados **descritores** ou **atributos**, acerca das variáveis.

## 3.7 Juntando as Partes

Os dois módulos indicados anteriormente podem ser combinados através de um módulo principal, neste caso, o módulo do **analisador léxico**. Lembre-se, este módulo será chamado pelo analisador sintático toda vez que um item léxico for exigido.

No analisador léxico podemos incluir recursos adicionais, como, por exemplo, eliminar os espaços em branco, eliminar comentários e efetuar a contagem de linhas. A rotina a seguir ilustra como seriam combinados os módulos discutidos anteriormente, e como eliminar os espaços em branco e os símbolos especiais, que por ventura ocorram no programa fonte, além de reconhecer o símbolo dois pontos (“.”) e o de atribuição (“:=”) do *Pascal*.

```
Rotina Léxico()
Espaço são os caracteres ASCII menor ou igual a 32
LF é o caracter ASCII 10
Inicio
    Enquanto não for Fim da Entrada faça
        Lexema = ''
        Enquanto o Caracter é Espaço e não é Fim da Entrada faça
            Se o CARACTER é um LF então
                Incremente o Contador de Linha
            Fim Se
            LER um novo CARACTER
        Fim Enquanto
        Se o Caracter é um DÍGITO então
            Número()
            Retorne
        Senão
        Se o Caracter é uma LETRA então
            Identificador()
            Retorne
        Senão
        Se o Caracter é ':' então
            Token.Classe = cDoisPontos
            LER um novo CARACTER
            Se o Caracter é '=' então
                Token.Classe = cAtribuição
                LER um novo CARACTER
            Fim Se
            Retorne
        Senão
        Ocorreu um Erro Léxico - caracter não reconhecido
        LER um novo CARACTER
    Fim Se
    Fim Enquanto
    Token.Classe = cEOF
Fim

Rotina AnalisadorSintático()
Inicio
    Ler um novo CARACTER
    Token.Classe = cId
    Enquanto Token.Classe <> cEOF faça
        Lexico()
        Mostre "Lexema: " + Lexema +
            "Classe: " + Token.Classe +
            "Valor: " + Token.Valor
    Fim Enquanto
Fim
```

## 3.8 Tabela de Palavras Reservadas

É comum em quase todas as linguagens de programação, a utilização de um conjunto de identificadores que são reservados, denominados de **palavras reservadas**. Estes são utilizados para facilitar o reconhecimento de algumas

construções sintáticas, não podendo ser utilizados como nomes de variáveis, tipos, funções, etc..

No *Pascal*, por exemplo, as seguintes palavras são reservadas, dentre outras:

```
and      array     begin     case      const     div       do
downto   else      end       file      for       function  goto
if       in        label    mod       nil       not      of
or       packed   procedure program  record   repeat   set
then     to        type     until   var      while   with
```

É tarefa do analisador léxico, ao reconhecer um identificador, verificar se o mesmo não é uma palavra reservada, classificando-o corretamente. Desta forma, ao se construir o analisador léxico, é necessário a definição de uma tabela contendo todas as palavras reservadas da linguagem. Uma rotina que permita verificar se o identificador reconhecido se encontra ou não na tabela, deverá ser também implementada.

O algoritmo a seguir ilustra como poderíamos implementar esta tabela e a função de verificação. É assumido que a tabela de palavras reservadas é global e esteja ordenada alfabeticamente. A função `IsKeyword(s)` recebe como entrada a string `s` a ser pesquisada e retorna `Verdade` se for uma palavra reservada ou `Falso` caso contrário. O método de pesquisa adotado pela função é a busca binária. Portanto, a tabela deverá estar ordenada corretamente.

```
Constante TamTabRes = 35
Declare TabRes como Array[1 até TamTabRes] de string contendo
      'and', 'array', 'begin', 'case', 'const', 'div',
      'do', 'downto', 'else', 'end', 'file', 'for',
      'function', 'goto', 'if', 'in', 'label', 'mod',
      'nil', 'not', 'of', 'or', 'packed', 'procedure',
      'program', 'record', 'repeat', 'set', 'then',
      'to', 'type', 'until', 'var', 'while', 'with'

Rotina IsKeyword(declare s como string)
Declare
  m, n, k como inteiro
Inicio
  s <- minúsculo(s)
  m <- 1
  n <- TamTabRes

  Enquanto (m <= n) faça
    k <- m + (n - m) / 2
    Se (s = TabRes[k]) então
      Retorne (Verdade)
    Senão
      Se (s > TabRes[k]) então
        m <- k + 1
      Senão
        n <- k - 1
      Fim se
    Fim se
```

```
Fim enquanto
  Retorne(Falso)
Fim rotina
```

## 3.9 Tabela de Símbolos

Enquanto a tabela de palavras reservadas é uma estrutura estática, utilizada apenas para consulta, a tabela de símbolos é uma estrutura dinâmica. Nela são mantidos todos os identificadores, e seus atributos, encontrados durante o processo de análise léxica. As informações aqui inseridas, pelo analisador léxico, serão constantemente consultadas e atualizadas durante as demais fases de análise e síntese do compilador.

Por exemplo, a seguinte construção *Pascal*:

```
Function Fat(n: integer): longint;
Var
  i: integer
Begin
  If n >= 0 then
    begin
      Fat := 1;
      For i := 1 to n do Fat := i * Fat;
    End;
End;
```

poderia resultar nas seguintes informações, as quais seriam inseridas na tabela de símbolos:

Lexema	Categoría	Tipo	Endereço
fat	Função	LongInt	-2
n	Parâmetro	Integer	-1
i	Variável	Integer	0

Neste caso, *Lexema* contém a cadeia de caracteres reconhecida como identificador; *Categoría* reflete a sua classificação no programa (*função*, *variável*, *parâmetro*, *procedimento*, *tipo*, etc.), o que é dependente do contexto onde ocorre; *Tipo* corresponde ao tipo de dados associado; *Endereço* identifica a sua posição de memória correspondente. Outros atributos poderiam ser evidenciados na tabela, de acordo com a necessidade.

Conforme já citado, é de responsabilidade do analisador léxico inserir na tabela de símbolos todos os novos identificadores encontrados durante a análise do programa fonte. Assim, a tabela de símbolos deverá ser pesquisada toda vez ao se reconhecer um identificador. Caso este seja encontrado, o seu endereço de acesso à tabela de símbolos é retornado. Caso contrário, ele será inserido e o seu endereço de acesso retornado.

Apesar da operação citada acima ser realizada durante a análise léxica, todas as operações abaixo devem ser possíveis:

1. Determinar se um dado nome está na tabela;
2. Adicionar um novo nome à tabela;
3. Acessar as informações associadas com um dado nome;
4. Adicionar novas informações para um dado nome;
5. Excluir um nome ou um grupo de nomes da tabela.

Desta forma, para implementarmos uma tabela de símbolos, a primeira coisa que devemos decidir é como iremos organizá-la, o que irá influenciar no tipo de busca a ser adotada e nas demais operações.

Basicamente, podemos implementar uma tabela de símbolos utilizando-se de estruturas do tipo vetor, lista lineares, árvores binárias e tabelas *hash*. Para implementações utilizando vetores e listas lineares, a busca deverá ser sequencial. Já as árvores binárias e tabelas *hash* oferecem mecanismos de busca mais otimizados.

Na prática, tabelas *hash* são as mais indicadas. Entretanto, a escolha deste método demanda uma boa definição da função *hash*, o que permitirá minimizar colisões. Outro ponto importante é tentarmos sempre definir uma tabela onde o número de elementos seja primo.

O exemplo a seguir ilustra o uso de tabelas *hash* como tabela de símbolos. A função *hash* escolhida é simples: *dada a soma de todos os códigos ASCII do identificador, calcula o módulo da divisão por tabSimLen, um número primo que corresponde ao número de elementos de tabSim*. As colisões são resolvidas inserindo-se os elementos em uma lista encadeada. Entretanto, outras estratégias, como a árvore binária, por exemplo, pode ser adotada.

```
Constantes
tabSimLen    = 211

Tipos
TCategoria   = (variável, função, parâmetro, ...)
PTabSimRec   = Ponteiro para TTabSimRec;
TTabSimRec   é um registro de
    Lexema   como cadeia de caracteres
    Categoria como TCategoria
    Próximo   como PTabSimRec
Fim do registro
TTabSim      = vetor contendo de 1 até tabSimLen de PTabSimRec

Declare
TabSim como TTabSim

Função Hash(Id como cadeia de caracteres) Retorna Integer
Declare
```

```

Índice como inteiro
Soma como inteiro
Início
    Soma <- 0
    Para Índice de 1 até Comprimento(Id) faz
        Soma <- Soma + ASCII(Id[Índice])
    Fim Para
    Retorne((Soma Módulo tabSimLen) + 1)
Fim

Função InsTabSim(Id como cadeia de caracteres) Retorna PTabSimRec
Declare
    h como inteiro
    p como PTabSimRec
Início
    h <- Hash(Id)
    p <- TabSim[h]
    Enquanto p <> nulo faz
        Se p->Lexema = Id então
            Retorne(p)
        Senão
            p <- p->Próximo
        Fim Se
    Fim Enquanto
    Aloque(p)
    p->Lexema <- Id
    p->Próximo <- TabSim[h]
    TabSim[h] <- p
    Retorne(p)
Fim

```

A função hash a seguir, denominada de função **HashPJW**, oferece um melhor desempenho que a anterior, minimizando o número de colisões.

```

Função Hash(Id como cadeia de caracteres) Retorna Integer
Declare
    lenId como inteiro
    h, g como inteiro
    p como inteiro
Início
    h <- 0
    g <- 0
    lenId <- Comprimento(Id);
    Para p de 1 até lenId faz
        h <- (h desloque 4 bits para esquerda) + ASCII(Id[p])
        g <- h E-Lógico 0xF0000000
        Se g <> 0 então
            h <- h OU-Exclusivo (g desloque 24 bits para direita)
            h <- h OU-Exclusivo g
        Fim Se
    Fim Para
    Retorne((h Módulo tabSimLen) + 1)
Fim

```

## 3.10 Geradores Automáticos de Analisador Léxico

Conforme verificado, é possível escrever um analisador léxico a partir da especificação de um autômato finito ou de expressões regulares. Entretanto, para linguagens mais complexas, essa estratégia de implementação é extremamente trabalhosa.

Para simplificar esta tarefa, diversas ferramentas de apoio foram desenvolvidas. Uma dessas ferramentas é os geradores de analisadores léxicos, que automatizam o processo de criação do autômato e o processo de reconhecimento de sentenças regulares a partir da especificação das expressões regulares correspondentes.

Uma das ferramentas mais tradicionais dessa classe é o programa *Lex*, originalmente desenvolvido para o sistema operacional Unix. O objetivo de *Lex* é gerar uma rotina para o analisador léxico (*scanner*) em C a partir de um arquivo, contendo a especificação das expressões regulares e trechos de código C que serão executados quando sentenças daquelas expressões forem reconhecidas.

Atualmente há diversas implementações de *Lex* para diferentes sistemas, assim como ferramentas similares que trabalham com outras linguagens de programação. Por exemplo, o *FLEX* é uma variação do *Lex*, produzido pelo *GNU*, o qual também gera código C e que também roda no Windows. Já o *JLex* é uma variação que gera código Java. Ainda há variações em Pascal, C++, dentre outras.

### 3.10.1 Especificação das Sentenças Regulares

O ponto de partida para a criação de um analisador léxico, usando *FLEX*, é criar o arquivo com a especificação das expressões regulares que descrevem os *itens léxicos* que são aceitos. Este arquivo é composto por até três seções, as quais são separadas pelo símbolo **%%**. São elas:

- **Declarações:** Esta é a primeira seção a ser declarada. É nela que se encontram as declarações de variáveis, de constantes manifestas e de definições regulares a serem usadas pelas regras de tradução.
- **Regras de Tradução:** Esta é a segunda seção a ser declarada. Nela encontram-se especificadas as expressões regulares válidas e as correspondentes ações do programa. Cada regra é expressa na forma de um par:

Padrão { Ação }

onde Padrão é uma expressão regular que pode ser reconhecida pelo analisador léxico gerado e Ação é um fragmento de programa descrevendo que ação o analisador léxico deverá tomar quando tal expressão é reconhecida. Para a descrição do Padrão, *Flex* define uma linguagem para descrição de expressões regulares, conforme será analisado a seguir. Já para a Ação, estas são descritas em C; entretanto, qualquer outra linguagem pode ser utilizada.

- **Procedimentos Auxiliares:** Esta é a terceira e última seção do arquivo de especificação. Nela são colocadas as definições de procedimentos necessários para a realização das ações especificadas ou auxiliares ao analisador léxico

### 3.10.2 Linguagem de Especificação de Padrões

A descrição de um padrão em **FLEX** é feita por meio de uma linguagem para descrição de expressões regulares. Esta linguagem utiliza a notação para expressões regulares, já definidas anteriormente, e acrescenta algumas modificações e extensões, conforme indicadas pela tabela a seguir.

Tabela 4: Notação para expressões regulares utilizadas pelo FLEX.

.	Todos os caracteres, exceto o avanço de linha
[abc]	Denota uma <b>classe de caracteres</b> a ser reconhecida – corresponde à expressão regular (a   b   c)
[a-f]	Denota uma <b>classe de caracteres</b> a ser reconhecida, a qual é composta pela faixa de a à f – corresponde a expressão regular (a   b   c   d   e   f)
[^abc]	Denota uma <b>classe de caracteres negada</b> , isto é, qualquer caráter exceto a, b ou c
R*	Zero ou mais ocorrências da expressão regular R
R+	Uma ou mais ocorrências da expressão regular R
R?	0 ou uma ocorrência da expressão regular R
R{4}	Exatamente quatro ocorrências da expressão regular R
R{2,}	Pelo menos duas ocorrências da expressão regular R
R{2,4}	Entre duas e quatro ocorrências da expressão regular R
^R	A expressão regular R ocorrendo apenas no início de uma linha
R\$	A expressão regular R ocorrendo apenas no final de uma linha
<<EOF>>	Fim de arquivo

Como exemplo, vejamos a especificação das definições regulares para identificadores e números:

ID [A-Za-z] ([A-Za-z] | [0-9]) \*

NUM [0-9]+ (\. [0-9]+) ?

Observe na definição de NUM a utilização de '\.'. A utilização do **símbolo de escape**'\' é necessária para que o **FLEX** não confunda o caracter '.', o qual deverá ser reconhecido, com a classe de todos os caracteres.

### 3.10.3 Exemplo de Especificação de Sentenças

O exemplo a seguir ilustra a criação de um arquivo de entrada para o **FLEX**, o qual deverá mostrar o número de caracteres, palavras e linhas do arquivo de entrada, que no caso será o teclado.

```
%option noyywrap

/* Igual ao Unix wc */
|{
int chars = 0;
int words = 0;
int lines = 0;
}

%%

[a-zA-Z]+ { words++; chars += strlen(yytext); }
\n        { chars++; lines++; }
.         { chars++; }

%%

int main(int argc, char **argv)
{
    yylex();
    printf("Linhas    : %8d\n", lines);
    printf("Palavras  : %8d\n", words);
    printf("Caracteres: %8d\n", chars);
}
```

Observando o programa identificamos alguns elementos aos quais devemos analisar:

1. A diretiva `%option noyywrap` informa ao **FLEX** para que seja usada uma função padrão para `yywrap`, função esta que será chamada quando o fim do arquivo for atingido;
2. Na seção de declarações, todo o texto envolvido por `{` e `}` será copiado integralmente para o arquivo de saída gerado;
3. Para que o analisador léxico gerado pelo **FLEX** processe toda a entrada, uma rotina `main()` é definida na seção de procedimentos auxiliares. Nela, encontra-se a chamada da função responsável pela análise léxica, que é a `yylex();`
4. A variável `yytext` permite identificar no texto o lexema reconhecido.

Considerando que o arquivo de especificação definido anteriormente tenha o nome de `Exemplol.l` (normalmente os arquivos de especificação para o **Flex** têm a extensão '`.l`', ), para gerar o arquivo em C contendo o analisador léxico, basta

chamar `flex Exemplo1.l`. O arquivo gerado pelo **FLEX**, contendo o código em C, é o `lex.yy.c`. Para compilar é só chamar um compilador C para gerar o executável, como por exemplo `gcc lex.yy.c`. No caso do uso do `gcc`, o arquivo executável gerado será o `a.exe`. Caso se deseje modificar o nome de saída, a diretiva `-o` deverá ser usada no `gcc`; neste caso, faríamos `gcc lex.yy.c -o Exemplo1.exe`. Agora, é só executar o programa gerado, digitar algumas palavras e apertar **CTRL-C** para finalizar. Eis um exemplo de execução:

```
Uberaba, fevereiro de 2019<ENTER><CTRL-C>
Linhas : 1
Palavras : 3
Caracteres: 27
```

O exemplo a seguir ilustra a criação de um arquivo de entrada para o **FLEX**, o qual reconhece um subconjunto de itens léxicos da linguagem Pascal.

```
%option noyywrap

/* Scanner para uma linguagem estilo Pascal */
{{

/* Necessário para o chamar o atof() */
#include <math.h>

}

digito      [0-9]
id         [a-zA-Z][a-zA-Z0-9]*
intnum     {digito}+
realnum    {digito}+.{digito}+

%%

if|then   printf( "Keyword      : %s\n", yytext );
begin|end printf( "Keyword      : %s\n", yytext );
{id}      printf( "Identificador : %s\n", yytext );
{intnum}  printf( "Inteiro       : %s (%d)\n", yytext, atoi( yytext ) );
{realnum} printf( "Real          : %s (%g)\n", yytext, atof( yytext ) );
"+|-/*|"/"   printf( "Operador      : %s\n", yytext );
"{"[^{}]\n}*"}" /* consome comentários de uma linha */
[ \t\n]+        /* consome espaços em branco */
.              printf( "Nao reconhecido: %s\n", yytext );

%%

int main( int argc, char **argv )
{
    ++argv, --argc; /* Salta o nome do programa */
```

```
if ( argc > 0 )
    yyin = fopen( argv[0], "r" );
else
    yyin = stdin;

yylex();
}
```

Pelo programa apresentado podemos observar que:

1. Após a especificação de uma definição regular, como `digito` por exemplo, ao nos referirmos a ela deveremos envolvê-la por chaves, conforme pode ser observado em `{digito}`;
2. A variável global `yyin` permite modificar o arquivo de entrada a ser analisado.

Novamente, considerando que o arquivo de especificação tenha o nome de `Exemplo2.1`, para gerar o arquivo em C contendo o analisador léxico basta chamar `flex Exemplo2.1` e a seguir compilá-lo com `gcc lex.yy.c`, por exemplo.

O exemplo a seguir ilustra a criação de um arquivo de entrada para o **FLEX**, o qual deverá reconhecer identificadores, números inteiros e números reais, além de instalar os identificadores em uma tabela de símbolos. Trata-se de um exemplo mais completo, onde pode ser notado a interação com um possível analisador sintático, o qual identifica o **token** analisado e toma as ações pertinentes.

```
%option noyywrap

/* seção de declaração */
|{
#include <math.h>
#include <string.h>

/* declaração das constantes manifestas */
#define CID 0
#define CINT 1
#define CREAL 2
#define CEOF 3

typedef char TLexema[30];

struct TIIdList {
    TLexema    lexema;
    struct     TIIdList *proximo;
};

union TYYLVAL {
    int        numInt;
    float      numFloat;
```

```

        struct      TIdList *idPtr;
};

TLexema    lexema;
union       YYLVAL      yylval;
struct      TIdList*   idList = NULL;

/* Insere o lexema na tabela de simbolos */
struct TIdList * insTabSim(TLexema lexema)
{
    struct TIdList *p, *ant;

    p = idList;

    while (p)
    {
        if (strcmp(p->lexema, lexema) == 0) return p;
        p = p->proximo;
    }
    p = (struct TIdList *) malloc(sizeof(struct TIdList));
    strcpy(p->lexema, lexema);
    p->proximo = idList;
    idList = p;

    return p;
}

%%

{id}      {strcpy(lexema, yytext); yylval.idPtr = insTabSim(lexema);
return cID;}
{intnum}  {yylval.numInt = atoi(yytext); return cINT;}
{realnum} {yylval.numFloat = atof(yytext); return cREAL;}
[ \t\n]+ /* consome espaços em branco */
.        /* consome os outros caracteres */
<<EOF>> {return cEOF;}

%%

int main( int argc, char **argv )
{
    ++argv, --argc; /* Salta o nome do programa */

    if ( argc > 0 )
        yyin = fopen( argv[0], "r" );
    else
        yyin = stdin;

    int token;

```

```

while ((token = yylex()) != cEOF)
{
    if (token == cID)
        printf("Identificador: %s\n", yyval.idPtr->lexema);
    else
        if (token == cINT)
            printf("Inteiro      : %d\n", yyval.numInt);
        else
            if (token == cREAL)
                printf("Real        : %f\n", yyval.numFloat);
}
}

```

Pelo exemplo pode-se notar a presença da variável global `yyval`, que é do tipo `union`. Esta variável permite armazenar os diferentes tipos de itens léxicos analisados. Ela é a correspondente para o `token.Valor`. Outro ponto a ser observado é a presença do comando `return`. Tal comando é utilizado para sinalizar ao analisador sintático qual foi o **`token`** reconhecido.

O exemplo a seguir ilustra a criação de um arquivo de entrada para o ***FLEX***, o qual deverá reconhecer identificadores e números, além de instalar os identificadores em uma tabela de símbolos e permitir a integração com o ***Bison***.

```

%option noyywrap

/* seção de declaração */
|{
#include <math.h>

/*      declaração das constantes manifestas */
#define cID     0
#define cINT    1
#define cREAL   2
}

digito [0-9]
id    [a-zA-Z][a-zA-Z0-9]*
intnum {digito}+
realnum {digito}+."{digito}+

%%
{id}           {yyval = instalar_id(); return cID;}
{intnum}        {yyval = atoi(yytext); return cINT;}
{realnum}       {yyval = atof(yytext); return cREAL;}
[ \t\n]+        /* consome espaços em branco */

%%
instalar_id() {
    /* procedimento para instalar o lexema, cujo primeiro
       caractere é apontado por yytext e cujo comprimento é
       dado por yyleng, na tabela de símbolos e retornar um

```

```
apontador para o mesmo
*/
}
```

Observando o exemplo identificamos novamente a variável `yyval`, a qual não é definida pelo programa. Na verdade, esta variável é definida globalmente pelo **YACC** (um gerador de analisador sintático), sendo utilizada para trocar informações com a função `yylex()`, identificando o item léxico reconhecido. Novamente, o comando `return` é utilizado para sinalizar ao analisador sintático qual foi o **token** reconhecido.

O próximo exemplo apresenta uma especificação mais completa, a qual reconhece também os operadores relacionais. Note pelo exemplo que os operadores relacionais são classificados como `RELOP`, mas são corretamente identificados por `yyval`.

```
%option noyywrap

/* seção de declaração */
{
/* declaração das constantes manifestas */
#define LT, LE, EQ, NE, GT, GE, IF, THEN, ELSE, ID, NUM, RELOP
}
/* definições regulares */
delim  [ \t\n]
ws      {delim}+
letra   [A-Za-z]
digito  [0-9]
id      {letra} ({letra} | digito)*
num     {digito}+ (\.{digito}+)? (E[+\-]? {digito}+)?

%%
{ws}    {/* nenhuma ação e nenhum valor retornado */}
if     {return (IF);}
then   {return (THEN);}
else   {return (ELSE);}
{id}    {yyval = instalar_id(); return (ID);}
{num}   {yyval = instalar_num(); return (NUM);}
"<"    {yyval = LT; return (RELOP);}
"<="   {yyval = LE; return (RELOP);}
"="    {yyval = EQ; return (RELOP);}
"<>"  {yyval = NE; return (RELOP);}
">"   {yyval = GT; return (RELOP);}
">="  {yyval = GE; return (RELOP);}

%%
/* procedimentos auxiliares */
instalar_id() {
    /* procedimento para instalar o lexema, cujo primeiro
       caractere é apontado por yytext e cujo comprimento é
       dado por yylen, na tabela de símbolos e retornar um
```

```
        apontador para o mesmo
    */
}

installar_num() {
    /* procedimento similar para instalar um lexema que seja
     * um número
    */
}
```

## 3.10.4 Resumindo: Geração do Analisador Léxico

Conforme já comentado anteriormente, após ter criado o arquivo contendo as especificações de entrada para o analisador léxico, normalmente um arquivo de extensão '.l', basta executarmos o **FLex**, especificando o nome deste arquivo. Como saída será gerado um arquivo de nome `lex.yy.c`, o qual deverá ser posteriormente modificado, se necessário, e compilado.

O ponto de entrada para o analisador léxico é a função `yylex()`, a qual quando chamada analisará o arquivo de entrada, especificado pela variável global `yyin`, e devolverá o **token** reconhecido. O arquivo de entrada é normalmente o teclado. Para modificar esta definição basta alterarmos o conteúdo da variável `yyin`.

## 3.10.5 Exercício

Desenvolva, utilizando o **FLex**, um analisador léxico que permita abrir um arquivo texto especificado e contar o número de palavras, de caracteres, de identificadores, de números (inteiros e reais), de operadores (relacionais e aritméticos) e o número de linhas. Além dos itens anteriores o programa deverá criar um índice alfabético para todos os identificadores encontrados no texto, indicando a quantidade de vezes que ele ocorre, a linha onde ele ocorre e quantas vezes ele ocorre em cada linha. Ao final, o programa deverá imprimir todas as informações coletadas.

## 4 Análise Sintática

### 4.1 Funções da Análise Sintática

O objetivo do **analisador sintático** é obter uma cadeia de **tokens**, provenientes de chamadas ao **analisador léxico**, e verificar se esta cadeia pode ser gerada pela **gramática** da linguagem fonte. Como saída, o **analisador sintático** pode gerar a **árvore gramatical** correspondente à cadeia de tokens identificada durante a **análise léxica**, além de relatar eventuais erros de sintaxe encontrados durante a fase de análise.

Os métodos mais comuns de análise sintática são classificados como:

- **Análise Sintática Descendente (Top-Down)**: o analisador tenta construir a árvore gramatical a partir de sua **raiz** em direção às **folhas**.
- **Análise Sintática Ascendente (Bottom-up)**: o analisador tenta construir a árvore gramatical começando pelas folhas em direção à raiz.

Em ambos os casos a entrada é varrida da esquerda para a direita, um símbolo de cada vez.

Deve ser salientado que os métodos de **análise sintática** mais eficientes, tanto *top-down* como *bottom-up*, trabalham apenas com determinadas subclasses de gramáticas. Entretanto, várias destas subclasses são suficientemente expressivas para descrever a maioria das linguagens de programação.

### 4.2 Ambiguidade

Antes de iniciarmos nosso estudo sobre a análise sintática, é importante discutirmos alguns problemas enfrentados durante a fase de projeto de analisadores sintáticos para determinadas gramáticas.

O primeiro destes problemas é a **ambiguidade**. Conforme analisado anteriormente, uma gramática é ambígua quando ela produz mais de uma árvore gramatical para a mesma entrada. Para certos tipos de analisadores sintáticos, é extremamente importante que a gramática não seja ambígua, pois, caso contrário, o analisador não saberá decidir sobre qual árvore gramatical gerar.

Um exemplo clássico para o problema da ambiguidade é o do **else vazio**. Considere a forma sentencial abaixo:

```
<cmd> ::= if <expL> then <cmd> |
           if <expL> then <cmd> else <cmd>
```

De acordo com esta gramática, a sentença:

```
if E1 then S1 else if E2 then S2 else S3
```

possui a árvore gramatical ilustrada pela Figura 19. Entretanto, a sentença:

```
if E1 then if E2 then S1 else S2
```

possui as árvores gramaticais ilustradas pela Figura 20, o que nos permite observar que tal gramática é ambígua.

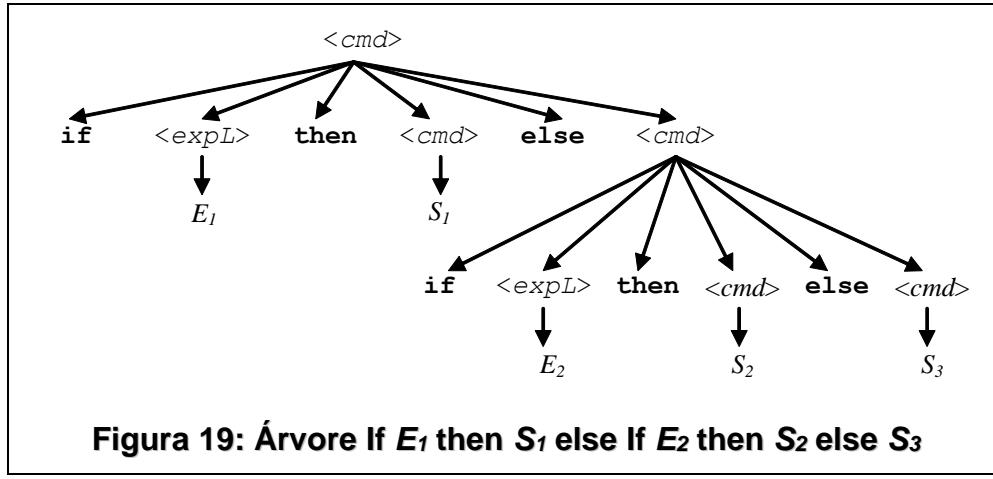


Figura 19: Árvore If E<sub>1</sub> then S<sub>1</sub> else If E<sub>2</sub> then S<sub>2</sub> else S<sub>3</sub>

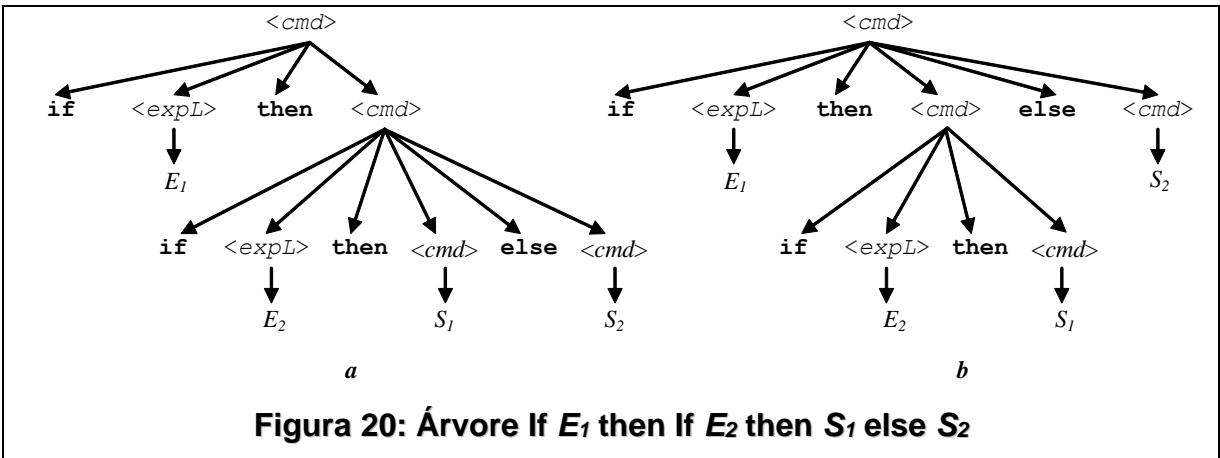


Figura 20: Árvore If E<sub>1</sub> then If E<sub>2</sub> then S<sub>1</sub> else S<sub>2</sub>

Observe que ambas árvores estão corretas, mas a primeira é a comumente utilizada em implementações de linguagens. Portanto, devemos efetuar algumas modificações na gramática de tal forma que apenas a árvore *a* seja gerada. Como regra, devemos associar cada *else* ao *if* mais próximo que ainda não tenha sido associado. Reconstruindo a gramática obteremos:

```

<cmd>      ::= <cmd_1> | <cmd_2>
<cmd_1>    ::= if <expr> then <cmd_1> else <cmd_1>
<cmd_2>    ::= if <expr> then <cmd> |
                  if <expr> then <cmd_1> else <cmd_2>
  
```

## 4.3 Produções vazias

Uma gramática é dita ser  **$\epsilon$ -livre** se ela não possui produções do tipo  $A \rightarrow \epsilon$ , exceto se  $A$  é o não terminal de partida.

Podemos transformar qualquer GLC  $G$  com produções vazias em uma  $G'$   **$\epsilon$ -livre**. Para tal, devemos remover todas as produções do tipo  $A \rightarrow \epsilon$ , onde  $A$  não é o símbolo de partida. Então, o que devemos fazer é: para cada ocorrência de um não terminal  $A$  no lado direito de uma regra, adiciona-se uma nova regra com a ocorrência removida. Em outras palavras, se  $R \rightarrow \alpha A \beta$  é uma regra e  $\alpha, \beta \in (V_N \cup V_T)^*$ , então adiciona-se a regra  $R \rightarrow \alpha \beta$ , com  $A$  removida. Faz-se isto para cada ocorrência de  $A$ . Repete-se este passo até que não existam mais produções vazias.

Por exemplo, suponha as produções abaixo:

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sd \mid \epsilon \end{aligned}$$

Removendo as produções vazias, que neste caso corresponde a  $A \rightarrow \epsilon$ , temos:

$$\begin{aligned} S &\rightarrow Aa \mid b \mid a \\ A &\rightarrow Ac \mid Sd \mid c \end{aligned}$$

que são produções  **$\epsilon$ -livre**.

## 4.4 Recursão à Esquerda

O segundo problema a considerar é a **recursão à esquerda**. Uma gramática é recursiva à esquerda se possui um não terminal  $A$ , tal que exista uma derivação  $S \Rightarrow^+ S\alpha$ ,  $S \in V_N$  e  $\alpha \in V^*$ , para alguma cadeia  $\alpha$ . Os métodos de *análise sintática descendente* não podem processar gramáticas recursivas à esquerda, portanto devemos eliminá-la.

Suponha uma gramática cujas regras de produção são:

$$\begin{aligned} S &\rightarrow S\alpha \\ S &\rightarrow \beta \\ S &\rightarrow \delta \end{aligned}$$

onde  $S \in V_N$  e  $\alpha, \beta, \delta \in V^*$ , sendo que  $\beta$  e  $\delta$  não começam com  $S$ . Esta gramática é recursiva à esquerda. Para eliminarmos a recursão aplicamos a seguinte regra:

1. Agrupamos todas as produções  $S$ , da forma:  $S \rightarrow S\alpha \mid \beta \mid \delta$ ; e
2. Substituímos as produções  $S$  por:

$$S \rightarrow \beta S' \mid \delta S'$$

$$S' \rightarrow \alpha S' \mid \epsilon$$

Caso não seja permitido produções vazias, o passo dois acima poderia também ser realizado da seguinte forma:

$$S \rightarrow \beta \mid \beta S' \mid \delta \mid \delta S'$$

$$S' \rightarrow \alpha S' \mid \alpha$$

Entretanto, este segundo modo introduz um novo problema, que é o não determinismo, o qual deverá ser resolvido com a fatoração à esquerda, conforme verificado a seguir.

Como exemplo, suponha as seguintes regras de produção, obtidas de uma gramática para expressões aritméticas:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E) \mid id \mid num$$

onde *id* e *num* representam, respectivamente, qualquer *identificador* ou *número* válido.

Primeiramente agrupamos as produções recursivas, ou seja:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

Agora substituímos as produções, o que nos daria as produções a seguir:

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

$$F \rightarrow (E) \mid id \mid num$$

O método apresentado acima elimina todas as recursões à esquerda envolvendo apenas um passo (recursões diretas), mas não elimina a recursão em mais de um passo (recursões indiretas), como a indicada abaixo:

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid e$$

Para eliminação de recursões indiretas, devemos efetuar os seguintes procedimentos, os quais correspondem a passos da [Forma Normal de Greibach](#):

- Renomeação dos não terminais em uma ordem crescente:** os símbolos não terminais devem ser renomeados em uma ordem crescente e as produções devem ser reescritas.

Por exemplo, considerando as produções temos:  $S = S_1$  e  $A = S_2$ . Logo, reescrevendo as produções temos:

$$\begin{aligned} S_1 &\rightarrow S_2a \mid b \\ S_2 &\rightarrow S_2c \mid S_1d \mid e \end{aligned}$$

- Transformação das produções para a forma  $A_r \rightarrow A_s\alpha$ ,  $r \leq s$ :** as produções são modificadas garantindo que o primeiro não terminal do lado direito seja maior ou igual ao do lado esquerdo, considerando a ordenação realizada no passo anterior. As produções  $A_r \rightarrow A_s\alpha$ , tais que  $r > s$ , são modificadas substituindo o não terminal  $A_s$  pelo lado direito de suas produções correspondentes,  $A_s \rightarrow \beta_1 \mid \dots \mid \beta_n$ , resultando em  $A_r \rightarrow \beta_1\alpha \mid \dots \mid \beta_n\alpha$ . O processo deve ser repetido até que todas as produções estejam na forma  $A_r \rightarrow A_s\alpha$ , com  $r \leq s$ . Ao término deste processo, as produções alteradas serão da forma  $A_r \rightarrow a\alpha$  ou  $A_r \rightarrow A_r\alpha$ , com  $a \in V_T$ ,  $A_r \in V_N$  e  $\alpha \in (V_N \cup V_T)^*$ .

Considerando as produções obtidas no passo anterior temos:

$$\begin{aligned} S_1 &\rightarrow S_2a \mid b \\ S_2 &\rightarrow S_2c \mid S_2ad \mid bd \mid e \end{aligned}$$

- Exclusão das recursões da forma  $A_r \rightarrow A_r\alpha$ :** o passo anterior pode produzir **recursões à esquerda**, ou seja, o primeiro não terminal do lado direito da produção é igual ao do lado esquerdo. A eliminação da recursão à esquerda de uma produção  $A_r \rightarrow A_r\alpha \mid \beta$ , é realizada introduzindo um novo símbolo não terminal  $B_r$  à gramática e substituindo esta produção pelas produções  $A_r \rightarrow \beta \mid \beta B_r$  e  $B_r \rightarrow \alpha \mid \alpha B_r$ . O processo deve ser repetido até que todas as produções  $A_r \rightarrow A_r\alpha$  tenham sido eliminadas.

Considerando as produções obtidas no passo anterior temos:

$$\begin{aligned} S_1 &\rightarrow S_2a \mid b \\ S_2 &\rightarrow bd \mid bdB \mid e \mid eB \\ B &\rightarrow c \mid cB \mid ad \mid adB \end{aligned}$$

ou, considerando que produções vazias sejam permitidas:

$$\begin{aligned} S_1 &\rightarrow S_2a \mid b \\ S_2 &\rightarrow bdB \mid eB \\ B &\rightarrow cB \mid adB \mid \epsilon \end{aligned}$$

## 4.5 Fatoração à Esquerda

O terceiro problema a ser considerado diz respeito à **fatoração à esquerda**, o qual cria gramáticas adequadas à **análise sintática preditiva**. Por exemplo, suponha as regras de produção abaixo

$$S \rightarrow aA \mid aB$$

Observe que não é possível decidir sobre qual das produções aplicar. Neste caso, a ideia é atrasar o máximo possível está decisão até termos uma ideia clara sobre qual produção aplicar. Deste modo, devemos reescrever tais regras da seguinte forma:

$$S \rightarrow aS'$$

$$S' \rightarrow A \mid B$$

As regras assim obtidas estão *fatoradas à esquerda*. Como regra geral podemos efetuar a fatoração através do algoritmo abaixo.

Seja uma gramática **G** cujas regras de produção são dadas por:

$$S \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \delta$$

onde  $S \in V_N$  e  $\alpha, \beta, \delta \in V^*$ , sendo que  $\alpha$  não ocorre em  $\delta$ . Para cada não terminal  $S$ , encontrar o mais longo prefixo  $\alpha$ , comum à duas ou mais alternativas. Se existe um prefixo comum, ou seja  $\alpha \neq \varepsilon$ , então substituir todas as produções  $S$ ,  $S \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \delta$ , por:

$$S \rightarrow \alpha S' \mid \delta$$

$$S' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

Aplicar repetidamente esta transformação até que não haja duas alternativas com um prefixo comum.

## 4.6 Análise Sintática Descendente

Como citado anteriormente, na análise descendente é feita uma tentativa para se construir uma árvore gramatical, para uma sentença de entrada, a partir de sua *raiz* em direção às *folhas*. Tal sentença deverá ser analisada da *esquerda* para a *direita*. A análise estará completa se cada uma das *folhas* da árvore contiver símbolos *terminais* da cadeia de entrada. Analisa-se a seguir alguns casos de análise descendente.

### 4.6.1 Análise Sintática Descendente com Retrocesso

O método de *análise descendente recursiva com retrocesso* foi um dos primeiros a serem utilizados. Entretanto, é um método muito ineficiente, não sendo, portanto, visto com muita frequência.

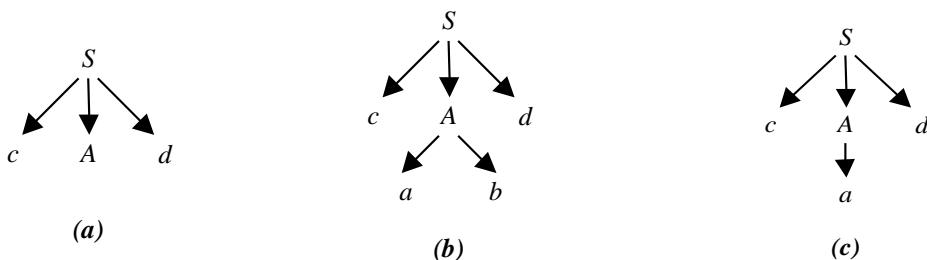
Para ilustrar este método, suponha uma gramática cujas produções são dadas a

seguir.

$$S \rightarrow cAd$$

$$A \rightarrow ab \mid a$$

Agora, suponha a cadeia de entrada  $cad$ . Aplicando as regras de produção teríamos as seguintes árvores gramaticais para a sentença:



Observe que a árvore  $a$  corresponde a derivação  $S \Rightarrow cAd$ . Para derivarmos o não terminal  $A$  existem duas alternativas. Neste ponto o analisador deve escolher uma das alternativas, mas deverá se lembrar das próximas alternativas e qual a situação atual da derivação, criando assim um **ponto de escolha**. Escolhendo-se a primeira alternativa,  $A \rightarrow ab$ , obtemos a árvore  $b$ , a qual corresponde à derivação  $S \Rightarrow^+ cabd$ . Neste ponto ocorre uma falha, pois a sentença gerada não corresponde à sentença de entrada. Deste modo, o analisador deve efetuar um **retrocesso** ao último *ponto de escolha*, restaurar a situação de derivação quando da escolha da alternativa que falhou, e continuar o processo de derivação a partir dai. Assim, a situação anterior era a forma sentencial  $cAd$ . Como a primeira alternativa falhou, resta agora seguir a segunda, ou seja,  $A \rightarrow a$ , o que produzirá a derivação  $S \Rightarrow^+ cad$ , a qual corresponde a árvore  $c$ . Como foi possível gerar a sentença inicial a partir desta gramática, então tal sentença pode ser reconhecida por esta gramática.

O problema por trás deste tipo de implementação é a dificuldade de se restaurar a situação no *ponto de escolha* e o atraso que isto provoca.

A seguir é analisado outro tipo de analisador descendente recursivo, mas sem retrocesso.

## 4.6.2 Análise Sintática Descendente Recursiva

O método de **análise descendente recursiva** necessita que a gramática não contenha *recursões à esquerda* e que esteja *fatorada à esquerda*. O método se baseia na escrita de um procedimento recursivo para cada *não-terminal* da gramática. Durante a derivação, quando um *terminal* for derivado, um novo *item léxico* deverá ser lido. E quando um *não-terminal* é encontrado, o procedimento que o define é chamado. As chamadas a estes procedimentos são recursivas e, portanto, a linguagem a ser utilizada na implementação deverá suportar esta característica de modo eficiente.

## Exemplo 1:

O exemplo abaixo ilustra um *analisador sintático descendente recursivo* para um reconhecedor de expressões aritméticas envolvendo *identificadores*, *valores inteiros* e *reais*. As regras gramaticais, já eliminada a recursão à esquerda, são:

```
<expressao>      ::= <termo> <maistermos>
<maistermos>    ::= + <termo> <maistermos>
                  | - <termo> <maistermos>
                  | ε
<termo>          ::= <fator> <maisfatores>
<maisfatores>   ::= * <fator> <maisfatores>
                  | / <fator> <maisfatores>
                  | ε
<fator>          ::= ( <expressao> ) | id | num
```

onde:

```
id      → letra (letra | digito)*
num    → digito+ | digito+ . digito+
letra   → [A-Za-z]
digito  → [0-9]
```

Deve ser observado que nesta gramática algumas produções podem derivar o símbolo  $\epsilon$ . Tal derivação deve ser entendida como uma *derivação default*. Isto é, se o analisador não for capaz de aplicar nenhuma das outras produções derivadas a partir do *não terminal*, então ele aplica a produção  $\epsilon$ , ou seja, não faz nada.

A implementação pode ser feita do seguinte modo:

```
// <expressao> ::= <termo> <maistermos>
Rotina Expressão()
Início
    Chamar Termo()
    Chamar MaisTermos()
Fim
```

```

// <maistermos>      ::= + <termo> <maistermos>
//                           | - <termo> <maistermos>
//                           | ε

Rotina MaisTermos()
Inicio
    Se Lexema é '+' ou '-' então
        Chamar Lexico()
        Chamar Termo()
        Chamar MaisTermos()
    Fim Se
Fim

// <termo>      ::= <fator> <maisfatores>

Rotina Termo()
Inicio
    Chamar Fator()
    Chamar MaisFatores()
Fim

// <maisfatores>      ::= * <fator> <maisfatores>
//                           | / <fator> <maisfatores>
//                           | ε

Rotina MaisFatores()
Inicio
    Se Lexema é '*' ou '/' então
        Chamar Lexico()
        Chamar Fator()
        Chamar MaisFatores()
    Fim

// <fator>      ::= ( <expressao> ) | id | num

Rotina Fator()
Inicio
    Se Lexema é '(' então
        Chamar Lexico()
        chamar Expressao
        Se Lexema é ')' então
            Chamar Lexico()
        Senão
            Erro(') esperado')
        Fim se
    Senão
        Se Token é id ou num então
            Chamar Lexico()
        Senão
            Erro('Operando esperado')
        Fim se
    Fim se

```

Fim

```
Programa AnalisadorSintaticoRecursivo
Inicio
    Chamar Lexico()
    Chamar Expressão()
    Se      Token <> Eof então
        Erro('Fim de arquivo não esperado')
    Fim    se
Fim
```

### 4.6.3 Análise Sintática Descendente Recursiva Preditiva

O método de **análise descendente recursiva preditiva** necessita que a gramática não contenha recursões à esquerda e que esteja *fatorada à esquerda*.

Para se construir um *analizador preditivo* precisamos conhecer, dado o símbolo de entrada  $\alpha$  e a produção:

$$S \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

com  $S, \beta_i \in V_N$  e  $\alpha \in V_T$ , qual das alternativas  $\beta_i$ ,  $1 \leq i \leq n$ , é a única que deriva a forma sentencial começando com  $\alpha$ . Isto é, deve ser possível detectar a alternativa a ser aplicada examinando apenas o primeiro símbolo da cadeia que a mesma deriva.

A seguir, basta se escrever um procedimento recursivo para cada não-terminal  $\beta_i$ , o qual deverá analisar a cadeia derivável a partir dele. Deste modo, o procedimento correspondente a um *não-terminal* deverá ser chamado sempre que for necessário derivar uma cadeia a partir deste *não-terminal*.

#### Exemplo 2:

Sejam as produções indicadas a seguir:

$$\begin{aligned} S &\rightarrow AS \mid BA \\ A &\rightarrow \mathbf{a}B \mid C \\ B &\rightarrow \mathbf{b}A \mid \mathbf{d} \\ C &\rightarrow \mathbf{c} \end{aligned}$$

O exemplo a seguir ilustra um *analizador sintático preditivo* para essas produções:

```
// S → AS | BA

Rotina S()
Inicio
    Caso Lexema é
        'a', 'c':    Chamar A()
                      Chamar S()
        'b', 'd':    Chamar B()
                      Chamar A()
        Senão:       Chamar Erro()
    Fim do Caso
Fim

// A → aB | C

Rotina A()
Inicio
    Caso Lexema é
        'a':         Chamar Lexico()
                      Chamar B()
        'c':         Chamar C()
        Senão:       Chamar Erro()
    Fim do Caso
Fim

// B → bA | d

Rotina B()
Inicio
    Caso Lexema é
        'b':         Chamar Lexico()
                      Chamar A()
        'd':         Chamar Lexico()
                      Chamar Erro()
        Senão:       Chamar Erro()
    Fim do Caso
Fim

// C → c Rotina C()

Rotina C()
Inicio
    Caso Lexema é
        'c':         Chamar Lexico()
        Senão:       Chamar Erro()
    Fim do Caso
Fim

Programa AnalisadorSintaticoPreditivo
Inicio
    Chamar Lexico()
    Chamar S()
    Se    Token <> Eof então
        Erro('Fim de arquivo não esperado')
    Fim    Se
Fim do Programa AnalisadorSintaticoPreditivo
```

### Exemplo 3:

O próximo exemplo ilustra um *analisador sintático preditivo*, o qual permite verificar se uma expressão aritmética está correta. A gramática é indicada abaixo, já se encontrando com as recursões à esquerda eliminadas e fatoradas à esquerda:

```

<expressao>      ::= <termo> <maistermos>
<maistermos>    ::= + <termo> <maistermos>
                  | - <termo> <maistermos>
                  | ε
<termo>          ::= <fator> <maisfatores>
<maisfatores>   ::= * <fator> <maisfatores>
                  | / <fator> <maisfatores>
                  | ε
<fator>          ::= ( <expressao> ) | id | num
  
```

A seguir temos a codificação para essa gramática.

```

Rotina Expressao ()
Inicio
  Se Token é id ou num ou '(' então
    Chamar Termo()
    Se Lexema é '+' ou '-' então
      Chamar MaisTermos()
    Fim Se
  Senão
    Erro('Operando esperado')
  Fim Se
Fim

Rotina MaisTermos ()
Inicio
  Se Token é '+' ou '-' então
    Chamar Lexico()
    Se Token é id ou num ou '(' então
      Chamar Termo()
      Se Lexema é '+' ou '-' então
        Chamar MaisTermos()
      Fim Se
    Senão
      Erro('Operando esperado')
    Fim Se
  Fim Se
Fim
  
```

```
Rotina Termo()
Início
    Se      Token é id ou num ou '(' então
        Chamar Fator()
        Se      Lexema é '*' ou '/' então
            Chamar MaisFatores()
        Fim    Se
    Senão
        Erro('Operando esperado')
    Fim    Se
Fim

Rotina MaisFatores()
Início
    Se      Token é '*' ou '/' então
        Chamar Lexico()
        Se      Token é id ou num ou '(' então
            Chamar Fator()
            Se      Lexema é '*' ou '/' então
                Chamar MaisFatores()
            Fim    Se
        Senão
            Erro('Operando esperado')
        Fim    Se
    Fim    Se
Fim

Rotina Fator()
Início
    Se      Token é id ou num então
        Chamar Lexico()
    Senão
        Se      Lexema é '(' então
            Chamar Lexico()
            Se      Token é id ou num ou '(' então
                Chamar Expressao()
                Se      Lexema é ')' então
                    Chamar Lexico()
                Senão
                    Erro(') esperado')
                Fim    se
            Senão
                Erro('Operando esperado')
            Fim    Se
        Senão
            Erro('Operando esperado')
        Fim    Se
    Fim    Se
Fim

Programa AnalisadorSintaticoDescendentePreditivo
Início
    Chamar Lexico()
    Chamar Expressao()
```

```

Se      Token <> Eof então
        Erro('Fim de arquivo não esperado')
    Fim   Se
Fim

```

## 4.6.4 Análise Sintática Preditiva Não Recursiva

É possível construir um *analisador sintático preditivo* não recursivo mantendo uma *pilha*, ao invés de chamadas recursivas, e consultando uma *tabela sintática* para a aplicação de uma produção a um dado *não-terminal*. A Figura 21 ilustra um *analisador sintático* deste tipo.

Pela figura podemos observar 5 componentes com tarefas distintas. A *entrada* é composta pela cadeia a ser analisada, finalizada pelo símbolo  $\$$ , o qual representa o fim da cadeia. A *pilha* contém uma sequência de símbolos gramaticais, com o símbolo  $\$$  indicando o seu fundo. Inicialmente, a pilha contém o símbolo de partida da gramática sobre o topo da pilha, isto é, acima de  $\$$ . A *tabela sintética*, aqui representada por  $M$ , é uma matriz bidimensional  $M[A, a]$ , onde  $A$  é um *não terminal* e  $a$  é um *terminal* ou o símbolo  $\$$ . O conteúdo de  $M[A, a]$  é uma produção a ser aplicada quando  $A$  está sobre o topo da pilha e  $a$  é o símbolo de entrada, ou uma condição de erro. Já o *Programa* funciona conforme indicado a seguir.

Seja  $X$  o símbolo no topo da *pilha* e  $a$  o símbolo presente na entrada. O *Programa* deve seguir os seguintes passos:

1. Se  $X = a = \$$ , o programa para e anuncia o término, com sucesso, da análise sintática;
2. Se  $X = a \neq \$$ , o programa remove  $X$  da *pilha* e avança o ponteiro da entrada para o próximo símbolo;
3. Se  $X$  é um não terminal, o programa consulta a entrada  $M[X, a]$  da *tabela sintética*. Essa entrada será uma produção para  $X$ , da gramática, ou uma entrada de erro. Se por exemplo,  $M[X, a] = \{X \rightarrow UVW\}$ , o programa substitui  $X$ , no topo da *pilha*, por  $UVW$ , com  $U$  ao topo. Como saída o programa executa a ação definida pela produção aplicada. No nosso caso, iremos assumir que será impresso a produção utilizada. Se  $M[X, a] = \text{erro}$ , o programa chama uma rotina de recuperação de erro.

Em termos gerais, podemos definir um algoritmo para um *analisador sintático não recursivo preditivo*, conforme se segue, tendo como entrada uma cadeia  $\omega$  a ser analisada e uma tabela sintética  $M$ , e como saída uma derivação mais a esquerda de  $\omega$ , se  $\omega$  estiver em  $L(G)$ , ou uma condição de erro, caso contrário.

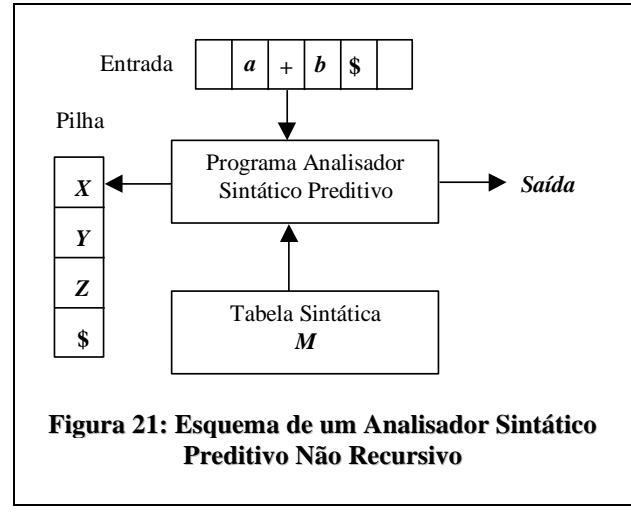


Figura 21: Esquema de um Analisador Sintático Preditivo Não Recursivo

```

Colocar  $\$S$  na pilha, com  $S$ , o símbolo de partida, ao topo
Fazer  $w\$$  a entrada
Faça  $ip$  apontar para o primeiro símbolo de  $w\$$ 
Repetir
    Seja  $X$  o símbolo ao topo da pilha
    Seja  $a$  o símbolo apontado por  $ip$ 
    Se  $X$  for um terminal ou  $\$$ , então
        Se  $X = a$ , então
            Remover  $X$  da pilha e avançar  $ip$ 
        Senão
            Erro()
        Fim Se
    Senão
        Se  $M[X, a] = X \rightarrow Y_1Y_2\dots Y_n$ , então
            Remover  $X$  da pilha
            Empilhar  $Y_nY_{n-1}\dots Y_1$ , com  $Y_1$  ao topo da pilha
            Escrever a produção  $X \rightarrow Y_1Y_2\dots Y_n$ 
        Senão
            Erro()
        Fim Se
    Fim Se
Até que  $X = \$$ 

```

Para exemplificar o funcionamento do algoritmo, considere a gramática abaixo, já eliminada a recursão a esquerda e fatorada a esquerda.

$$\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid \epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid \epsilon \\
F &\rightarrow (E) \mid id
\end{aligned}$$

A *tabela sintática* para esta gramática é dada pela Figura 22. As entradas em branco são entradas de erro, enquanto que as demais indicam uma produção com a qual se deve expandir o *não terminal* ao topo da *pilha*.

Suponha que a entrada contenha a sentença  $id + id * id$ . Para esta entrada, o analisador preditivo realizará a sequência de movimentos descritos pela Figura 23.

Não Terminal	Símbolos de Entrada					
	<i>Id</i>	+	*	(	)	\$
<i>E</i>	$E \rightarrow TE'$			$E \rightarrow TE'$		
<i>E'</i>		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
<i>T</i>	$T \rightarrow FT'$			$T \rightarrow FT'$		
<i>T'</i>		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
<i>F</i>	$F \rightarrow id$			$F \rightarrow (E)$		

Figura 22: Tabela Sintática  $M$  para Expressões Matemáticas Contendo Apenas Adição e Multiplicação.

Pilha	Entrada	Saída
$\$E$	<i>id</i> + <i>id</i> * <i>id</i> \$	
$\$E'T$	<i>id</i> + <i>id</i> * <i>id</i> \$	$E \rightarrow TE'$
$\$E'T'F$	<i>id</i> + <i>id</i> * <i>id</i> \$	$T \rightarrow FT'$
$\$E'T'id$	<i>id</i> + <i>id</i> * <i>id</i> \$	$F \rightarrow id$
$\$E'T'$	+ <i>id</i> * <i>id</i> \$	
$\$E'$	+ <i>id</i> * <i>id</i> \$	$T' \rightarrow \epsilon$
$\$E'T+$	+ <i>id</i> * <i>id</i> \$	$E' \rightarrow +TE'$
$\$E'T$	<i>id</i> * <i>id</i> \$	
$\$E'T'F$	<i>id</i> * <i>id</i> \$	$T \rightarrow FT'$
$\$E'T'id$	<i>id</i> * <i>id</i> \$	$F \rightarrow id$
$\$E'T'$	* <i>id</i> \$	
$\$E'T'F*$	* <i>id</i> \$	$T' \rightarrow *FT'$
$\$E'T'F$	<i>id</i> \$	
$\$E'T'id$	<i>id</i> \$	$F \rightarrow id$
$\$E'T'$	\$	
$\$E'$	\$	$T' \rightarrow \epsilon$
\$	\$	$E' \rightarrow \epsilon$

Figura 23: Movimentos feitos pelo Analisador Preditivo Não Recursivo para a entrada *id* + *id* \* *id*.

#### 4.6.4.1 Primeiro e Seguinte

A construção de analisadores sintáticos preditivos não recursivos é auxiliada por duas funções associadas à gramática  $G$ . Estas funções, denominadas de **Primeiro** (*First*) e **Seguinte** (*Follow*), nos permitem preencher as entradas de uma tabela sintática preditiva para  $G$ , sempre que possível. Os conjuntos de *tokens* produzidos pela função **Seguinte** podem também serem usados como *tokens* de sincronização durante a recuperação de erros.

Seja  $\alpha$  qualquer cadeia de símbolos gramaticais e  $A$  um não terminal que ocorre em alguma forma sentencial de  $G$ . Podemos definir intuitivamente  $Primeiro(\alpha)$  como

sendo o conjunto de todos os terminais que começam as cadeias derivadas a partir de  $\alpha$ , e  $Seguinte(A)$  como sendo o conjunto de terminais  $\alpha$  que podem figurar imediatamente à direita de  $A$  em alguma forma sentencial, isto é,  $S \Rightarrow^* \alpha A\beta$ , para algum  $\alpha$  e  $\beta$ .

De uma maneira mais precisa, podemos calcular  $Primeiro(X)$  para todos os símbolos gramaticais  $X$ , aplicando repetidamente as regras abaixo até que nenhum terminal ou  $\epsilon$  possa ser adicionado a qualquer conjunto  $Primeiro$ :

1. Se  $X$  for um *terminal*, então  $Primeiro(X)$  é  $\{X\}$ ;
2. Se  $X \rightarrow \epsilon$ , for uma produção, então adicionar  $\epsilon$  a  $Primeiro(X)$ ;
3. Se  $X$  for um *não terminal* e  $X \rightarrow Y_1Y_2 \dots Y_n$  uma produção, então colocar a  $\epsilon$  em  $Primeiro(X)$  se, para algum  $i$ , a estiver em  $Primeiro(Y_i)$  e  $\epsilon$  estiver em todos  $Primeiro(Y_1), \dots, Primeiro(Y_{i-1})$ ; isto é, se  $Y_1 \dots Y_{i-1} \Rightarrow^* \epsilon$ ;
4. Se  $X$  for um *não terminal*,  $X \rightarrow Y_1Y_2 \dots Y_n$  uma produção, e  $\epsilon$  estiver em todos  $Primeiro(Y_i)$ ,  $i = 1, 2, \dots, n$ , então adicionar  $\epsilon$  a  $Primeiro(X)$ ;
5. Se  $X$  for uma cadeia do tipo  $X_1X_2 \dots X_n$ , adicionar a  $Primeiro(X_1X_2 \dots X_n)$  todos os símbolos de  $Primeiro(X_1) - \{\epsilon\}$ . Se  $\epsilon \in Primeiro(X_1)$ , então adicionar todos os símbolos de  $Primeiro(X_2) - \{\epsilon\}$  a  $Primeiro(X_1X_2 \dots X_n)$ . Se  $\epsilon \in Primeiro(X_2)$ , então adicionar todos os símbolos de  $Primeiro(X_3) - \{\epsilon\}$  a  $Primeiro(X_1X_2 \dots X_n)$ , e assim sucessivamente. Finalmente, adicionar  $\epsilon$  a  $Primeiro(X_1X_2 \dots X_n)$ , se e somente se, para todo  $i$ ,  $i = 1, 2, \dots, n$ ,  $\epsilon \in Primeiro(X_i)$ .

Do mesmo modo, podemos calcular  $Seguinte(X)$  para todos os não terminais  $X$ , aplicando repetidamente as regras abaixo até que nada mais possa ser adicionado a qualquer conjunto  $Seguinte$ :

1. Colocar  $\$$  em  $Seguinte(S)$ , onde  $S$  é o símbolo de partida da gramática e  $\$$  o marcador de fim da entrada;
2. Se existir uma produção  $X \rightarrow \alpha A\beta$ , colocar em  $Seguinte(A)$ , tudo em  $Primeiro(\beta)$ , exceto  $\epsilon$ ;
3. Se existir uma produção  $X \rightarrow \alpha A$  ou uma produção  $X \rightarrow \alpha A\beta$ , onde  $\epsilon \in Primeiro(\beta)$ , isto é,  $\beta \Rightarrow^* \epsilon$ , então, adicionar tudo de  $Seguinte(X)$  em  $Seguinte(A)$ .

Como exemplo, considere as produções abaixo:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid id \end{aligned}$$

Os conjuntos *Primeiro* e *Seguinte* para estas produções encontram-se listados a seguir. Lembre-se que o primeiro de um terminal é o conjunto contendo o próprio terminal, o qual não será listado aqui.

$$\text{Primeiro}(E) = \text{Primeiro}(T) = \text{Primeiro}(F) = \{ (, \text{id}) \}$$

$$\text{Primeiro}(E) = \{ +, \epsilon \}$$

$$\text{Primeiro}(T) = \{ *, \epsilon \}$$

$$\text{Primeiro}(TE) = \text{Primeiro}(T) = \text{Primeiro}(F) = \{ (, \text{id}) \}$$

$$\text{Primeiro}(+TE) = \{ + \}$$

$$\text{Primeiro}(FT) = \text{Primeiro}(F) = \{ (, \text{id}) \}$$

$$\text{Primeiro}(^*FT) = \{ * \}$$

$$\text{Seguinte}(E) = \text{Seguinte}(E) = \{ ) , \$ \}$$

$$\text{Seguinte}(T) = \text{Seguinte}(T) = \{ +, ) , \$ \}$$

$$\text{Seguinte}(F) = \{ +, *, ) , \$ \}$$

#### 4.6.4.2 Tabelas Sintáticas Preditivas

Uma *tabela sintática*  $M$  para uma gramática  $G$  pode ser construída por intermédio do algoritmo a seguir:

1. Para cada produção  $X \rightarrow \alpha$  da gramática, execute os passos 2, 3 e 4;
2. Para cada terminal  $a$  em  $\text{Primeiro}(\alpha)$ , adicione  $X \rightarrow \alpha$  a  $M[X, a]$ ;
3. Se  $\epsilon \in \text{Primeiro}(\alpha)$ , então adicione  $X \rightarrow \alpha$  a  $M[X, b]$  para cada terminal  $b \in \text{Seguinte}(X)$ ;
4. Se  $\epsilon \in \text{Primeiro}(\alpha)$  e  $\$ \in \text{Seguinte}(X)$ , então adicione  $X \rightarrow \alpha$  a  $M[X, \$]$ ;
5. Faça cada entrada indefinida de  $M$  ser **erro**.

Aplicando-se o algoritmo acima à gramática da seção anterior obtemos a *tabela sintática* da Figura 22.

#### 4.6.4.3 Gramáticas LL(1)

Suponha a gramática do exemplo anterior. Para esta gramática, podemos implementar um *analisador sintático descendente preditivo* não recursivo analisando apenas um único símbolo da entrada (*lookahead*) para tomarmos as decisões sintáticas. Tal símbolo deve ser analisado a partir da *esquerda para a direita* (*Left to right*), produzindo uma *derivação linear mais à esquerda* (*left linear*). Portanto, dizemos que esta gramática é *LL(1)*.

Gramáticas *LL(1)* possuem várias propriedades distintas. Nenhuma gramática ambígua ou recursiva à esquerda pode ser *LL(1)*. Pode ser também demonstrado que uma gramática é *LL(1)* se, e somente se, sempre que  $A \rightarrow \alpha \mid \beta$  forem duas

produções distintas de  $G$  e vigorarem as seguintes condições:

1.  $\alpha$  e  $\beta$  não derivam, ao mesmo tempo, cadeias começando pelo mesmo terminal  $a$ , qualquer que seja  $a$  – isto é,  $\text{Primeiro}(\alpha) \cap \text{Primeiro}(\beta) = \emptyset$ ;
2. No máximo um dos dois,  $\alpha$  ou  $\beta$ , derivam  $\varepsilon$  – isto é,  $\{\varepsilon\} \notin \text{Primeiro}(\alpha) \cap \text{Primeiro}(\beta)$ ;
3. Se  $\beta \Rightarrow^* \varepsilon$ , então  $\alpha$  não deriva qualquer cadeia começando por um terminal em  $\text{Seguinte}(A)$  – isto é,  $\text{Primeiro}(\alpha) \cap \text{Seguinte}(A) = \emptyset$ .

Caso as restrições acima não sejam respeitadas, então a tabela sintática possuirá entradas duplamente definidas.

Por exemplo, a gramática a seguir não é  $LL(1)$ , pois ela é ambígua. Isto é, ela não atende à terceira condição citada acima.

$$\begin{aligned} S &\rightarrow i \ E \ t \ S \ S' \mid a \\ S' &\rightarrow e \ S' \mid \varepsilon \\ E &\rightarrow b \end{aligned}$$

Gramáticas *recursivas à esquerda* ou *ambíguas* não são  $LL(1)$ , pois pelo menos uma de suas entradas na *tabela sintática* são *multiplamente definidas*, conforme pode ser visto pela Figura 24. Nela, podemos observar que a entrada para  $M[S', e]$  contém tanto  $S \rightarrow e \ S'$  como  $S \rightarrow \varepsilon$ , uma vez que  $\text{Seguinte}(S') = \{e, \$\}$ .

Não Terminal	Símbolos de Entrada					
	<b>a</b>	<b>b</b>	<b>e</b>	<b>i</b>	<b>T</b>	<b>\$</b>
$S$	$S \rightarrow a$			$S \rightarrow iEtSS'$		
$S'$			$S' \rightarrow \varepsilon$ $S' \rightarrow eS$			$S' \rightarrow \varepsilon$
$E$		$E \rightarrow b$				

Figura 24: Tabela Sintática  $M$  para a Gramática  $S \rightarrow iEtSS' / a, S' \rightarrow eS | \varepsilon, E \rightarrow b$ .

Poderíamos tentar transformar uma gramática para  $LL(1)$  eliminando a *recursão à esquerda* e *fatorando à esquerda*. Entretanto, existem algumas gramáticas para as quais nenhuma alteração irá produzir uma gramática  $LL(1)$ , como é o caso do nosso exemplo.

Uma outra forma de construirmos a *tabela sintática* é feita através da numeração de cada uma das produções da gramática. Deste modo, para cada entrada  $M[X, a]$  deverá ser armazenado o número da produção a ser aplicada ou vazio. Este tipo de representação é mais indicado para a implementação de um programa de *análise descendente não recursivo* dirigido por uma *tabela sintática*, pois cada número representando uma produção pode ser tratado como um estado de um *autômato*.

A seguir encontram-se novamente listadas as produções para expressões aritméticas, com os respectivos números de identificação de cada produção.

1.  $E \rightarrow TE'$
2.  $E' \rightarrow +TE'$
3.  $E' \rightarrow \epsilon$
4.  $T \rightarrow FT'$
5.  $T' \rightarrow *FT'$
6.  $T' \rightarrow \epsilon$
7.  $F \rightarrow (E)$
8.  $F \rightarrow id$

A tabela sintática resultante encontra-se ilustrada na Figura 25.

Não Terminal	Símbolos de Entrada					
	<i>id</i>	+	*	(	)	\$
$E$	1			1		
$E'$		2			3	3
$T$	4			4		
$T'$		6	5		6	6
$F$	8			7		

Figura 25: Tabela Sintática  $M$  para Expressões Matemáticas com o Número da Produção a ser Aplicada.

## 4.6.5 Exercícios

1. Seja as produções de uma gramática G, indicadas a seguir.

$$S \rightarrow Sa \mid cA$$

$$A \rightarrow Ac \mid \epsilon$$

Remova as produções vazias, elimine a recursão a esquerda e fatore G, caso necessário. A seguir implemente em Java um analisador sintático para esta gramática. Considere que já exista um procedimento `Lexico()` que quando chamado retorna o último item léxico lido na variável pública `Lexema`.

2. Determine o conjunto *Primeiro* e *Seguinte* para a gramática a seguir, construa a sua tabela sintática e indique se ela é *LL(1)* e o porquê.

$S \rightarrow aSbS \mid bSaS \mid \epsilon$

3. Determine o conjunto *Primeiro* e *Seguinte* para a gramática a seguir, construa a sua tabela sintática e indique se ela é *LL(1)* e o porquê.

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

4. Determine o conjunto *Primeiro* e *Seguinte* para a gramática a seguir, construa a sua tabela sintática e indique se ela é *LL(1)* e o porquê.

$S \rightarrow \mathbf{id} \ A \bullet$

$A \rightarrow ( \ B \ ) \mid \epsilon$

$B \rightarrow \mathbf{id} \ C$

$C \rightarrow , \ B \mid \epsilon$

5. Determine o conjunto *Primeiro* e *Seguinte* para a gramática a seguir, construa a sua tabela sintática e indique se ela é *LL(1)* e o porquê.

$S \rightarrow \mathbf{id} \ B$

$B \rightarrow , \ S \mid : \ C$

$C \rightarrow \mathbf{int} \ D \mid \mathbf{real} \ D$

$D \rightarrow ; \ S \mid \epsilon$

## 4.7 Análise Sintática Ascendente

A **análise sintática ascendente (Bottom-Up)**, também denominada de **análise redutiva** (ou ainda **Shift-Reduce**), analisa uma sentença de entrada e tenta construir uma árvore de derivação, começando pelas **folhas** e prosseguindo para a **raiz**, produzindo uma **derivação mais à direita**, na ordem inversa. Caso seja obtida uma árvore cuja raiz tem por rótulo o símbolo inicial da gramática e na qual a sequência dos rótulos das folhas formam a sentença de entrada, então esta sentença pertence a linguagem gerada pela gramática e a árvore obtida é a sua árvore de derivação.

Podemos pensar na análise ascendente como o processo de *reduzir* uma sentença de entrada  $\alpha$  para o símbolo inicial  $S$  da gramática.

Por exemplo, seja a sentença  $abbcde$  e as produções gramaticais a seguir:

$$S \rightarrow aABe$$

$$A \rightarrow Abc \mid b$$

$$B \rightarrow d$$

Para reconhecer esta sentença devemos procurar por uma subcadeia que possa ser derivada por alguma das produções acima, e substituí-la pelo não-terminal do lado esquerdo da regra. O processo deve ser repetido até que a cadeia de entrada esteja reduzida ao símbolo inicial  $S$ .

Podemos demonstrar os passos de análise da cadeia acima através da tabela a abaixo.

Tabela 5: Passos para a análise ascendente da cadeia  $abbcde$ .

Passos	Entrada	Regra Aplicada	Saída
1.	<u><math>abbcde</math></u>	$A \rightarrow b$	$aAbcde$
2.	<u><math>aAbcde</math></u>	$A \rightarrow Abc$	$aAde$
3.	<u><math>aAde</math></u>	$B \rightarrow d$	$aABe$
4.	<u><math>aABe</math></u>	$S \rightarrow aABe$	$S$

Na tabela podemos observar que o processo de redução corresponde exatamente a seguinte derivação mais a direita, na ordem inversa:

$$S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$$

Também podemos observar que a análise da sentença de entrada é feita da esquerda para a direita, na tentativa de se encontrar uma subcadeia  $\beta$  em uma sentença  $\alpha\beta\gamma$ ,  $\gamma$  contendo apenas símbolos terminais, tal que exista uma produção  $\delta \rightarrow \beta$ , cuja substituição de  $\beta$  por  $\delta$  permita a redução de  $\alpha\delta\gamma$ , em zero ou mais passos, ao símbolo inicial. À forma sentencial  $\delta \rightarrow \beta$ , dá-se o nome de **handle**. Se a subcadeia  $\beta$  estiver bem definida em  $\alpha\beta\gamma$  e a produção  $\delta \rightarrow \beta$  for clara no contexto, então dizemos que  $\beta$  é um *handle* de  $\alpha\beta\gamma$ .

A tabela a seguir ilustra o *handle* correspondente a cada etapa de redução.

Tabela 6: Handle para cada etapa de redução.

Passos	Entrada	Handle	Regra Aplicada	Saída
5.	<i>abbcde</i>	<i>b</i>	$A \rightarrow b$	<i>aA<del>b</del>cde</i>
6.	<i>aAbcde</i>	<i>Abc</i>	$A \rightarrow Abc$	<i>aA<del>d</del>e</i>
7.	<i>aA<del>d</del>e</i>	<i>d</i>	$B \rightarrow d$	<i>aABe</i>
8.	<i>aABe</i>	<i>aABe</i>	$S \rightarrow aABe$	<i>S</i>

### 4.7.1 Construção de Analisadores Ascendentes

A construção de *analisadores ascendentes* é feita através da implementação de *autômatos de pilha*, cujos controles são dirigidos por *tabelas de análise sintática*. Para tal é utilizado uma pilha sintática para guardar os símbolos gramaticais de uma sentença de entrada a ser decomposta. Usamos  $\$$  para marcar o final da pilha e o final da sentença de entrada. Deste modo, no início do processo de análise deverá ser empilhado sobre a pilha o símbolo  $\$$ . A sentença a ser analisada também deverá ser seguida de  $\$$ . Assim se  $\alpha$  é a cadeia de entrada, então teremos a seguinte configuração:

Pilha	Sentença de Entrada
$\$$	$\alpha\$$

O processo de reconhecimento pelo analisador sintático, consiste em empilhar zero ou mais símbolos da sentença de entrada até que um *handle*  $\beta$  surja sobre o topo da pilha. Quando isto ocorre, reduz-se  $\beta$  para o não-terminal correspondente, empilhando-o. O processo se repete até que tenha sido detectado um erro ou que a pilha contenha em seu topo o não-terminal de partida, seguido do símbolo  $\$$ , e a entrada esteja vazia. Isto é:

Pilha	Sentença de Entrada
$\$S$	$\$$

Existem efetivamente quatro ações possíveis de serem realizadas pelo analisador sintático:

1. **Empilhar (Shift)**: esta ação é executada para se colocar o símbolo de entrada sobre a pilha;
2. **Reducir (Reduce)**: esta ação é executada quando um *handle* está sobre a pilha. Neste caso, o analisador o substitui pelo não terminal correspondente;
3. **Aceitar**: esta ação anuncia o término, com sucesso, do reconhecimento da sentença de entrada;
4. **Erro**: esta ação é executada quando um erro sintático é encontrado. Neste

caso o analisador sintático deverá chamar uma rotina de recuperação de erros.

Considere como exemplo as regras gramaticais:

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow (E)$$

$$E \rightarrow id$$

e a sentença  $id_1 + id_2 * id_3$ . O processo de reconhecimento é demonstrado pela

**Tabela 7.**

Observe pela tabela que para ocorrer o processo de redução o *handle* deverá sempre estar sobre o topo da pilha.

**Tabela 7:** Ação durante a análise ascendente para o reconhecimento da cadeia  $id_1 + id_2 * id_3$ .

Passo	Pilha	Entrada	Handle	Ação
1.	\$	$id_1 + id_2 * id_3 \$$		Empilhar
2.	$\$ id_1$	$+ id_2 * id_3 \$$	$id_1$	Reducir por $E \rightarrow id$
3.	$\$ E$	$+ id_2 * id_3 \$$		Empilhar
4.	$\$ E +$	$id_2 * id_3 \$$		Empilhar
5.	$\$ E + id_2$	$* id_3 \$$	$id_2$	Reducir por $E \rightarrow id$
6.	$\$ E + E$	$* id_3 \$$	$E + E$	Reducir por $E \rightarrow E + E$
7.	$\$ E$	$* id_3 \$$		Empilhar
8.	$\$ E *$	$id_3 \$$		Empilhar
9.	$\$ E * id_3$	\$	$id_3$	Reducir por $E \rightarrow id$
10.	$\$ E * E$	\$	$E * E$	Reducir por $E \rightarrow E + E$
11.	$\$ E$	\$		Aceitar

## 4.7.2 Análise de Precedência de Operadores

Os analisadores de precedência de operadores operam sobre a classe de gramáticas denominadas de **gramáticas de operadores**, que são aquelas que não contêm produções do tipo  $A \rightarrow \epsilon$  e no lado direito das produções os não terminais aparecem sempre separados por símbolos terminais, ou seja, não há dois ou mais não terminais adjacentes, tal como  $A \rightarrow \alpha BC\delta$ .

Por exemplo, seja a seguinte gramática:

$$E \rightarrow EOE | ( E ) | - E | id$$

$$O \rightarrow + | - | * | / | ^$$

Esta gramática não é de operadores, devido à produção  $E \rightarrow EOE$ , a qual possui três não terminais consecutivos. Entretanto, podemos transformá-la para uma gramática de operadores, substituindo o não terminal  $O$  pelos terminais por ele gerado, obtendo:

$$E \rightarrow E + E | E - E | E * E | E / E | E ^ E | ( E ) | - E | id$$

A análise de precedência de operadores é bastante eficiente e é aplicada, principalmente, no reconhecimento de expressões. Entretanto, este método apresenta algumas desvantagens, dentre elas a dificuldade de lidar com operadores iguais que tenham significados diferentes (como por exemplo, o operador “-”, o qual pode ser tanto binário como unário) e ser aplicado a apenas uma classe restrita de gramáticas.

### 4.7.2.1 Relações de Precedência de Operador

A partir de uma gramática de operadores podemos construir um analisador sintático utilizando-se das **relações de precedência de operadores** existentes entre os *tokens* a serem analisados.

Existem três relações de precedência de operadores:

1.  $a \lessdot b$ : *a* tem **precedência menor** que *b*;
2.  $a \dot> b$ : *a* tem **precedência maior** que *b*;
3.  $a \dot= b$ : *a* e *b* têm a mesma precedência.

A utilidade destas relações na análise de uma sentença é a identificação do *handle*:

- $\lessdot$  : identifica o limite esquerdo do *handle*;
- $\dot>$  : identifica o limite direito do *handle*;
- $\dot=$  : indica que os terminais pertencem ao mesmo *handle*.

Devemos ter cuidado ao interpretar esses operadores pois, diferentemente dos operadores maior, menor e igual da matemática, é perfeitamente possível termos  $a \triangleleft b$  e  $a \triangleright b$  simultaneamente.

Os analisadores sintáticos de precedência de operadores são dirigidos por uma **tabela de precedência**, cujas relações definem o movimento que o analisador deve fazer: *empilhar*, *reduzir*, *aceitar* ou *chamar uma rotina para tratamento de erros*. Esta tabela é uma matriz quadrada que relaciona todos os terminais da gramática mais o marcador \$. Os terminais nas *linhas* representam *terminais no topo da pilha*, e os terminais nas *colunas* representam *terminais sob a cabeça de leitura*.

Por exemplo, sejam as produções a seguir:

$$E \rightarrow E \vee E \mid E \wedge E \mid (E) \mid id$$

A tabela de precedência de operadores para esta gramática é indicada abaixo.

	<i>id</i>	$\vee$	$\wedge$	(	)	\$
<i>id</i>		$\triangleright$	$\triangleright$		$\triangleright$	$\triangleright$
$\vee$	$\triangleleft$	$\triangleright$	$\triangleleft$	$\triangleleft$	$\triangleright$	$\triangleright$
$\wedge$	$\triangleleft$	$\triangleright$	$\triangleright$	$\triangleleft$	$\triangleright$	$\triangleright$
(	$\triangleleft$	$\triangleleft$	$\triangleleft$	$\triangleleft$	$\triangleleft$	$\equiv$
)		$\triangleright$	$\triangleright$		$\triangleright$	$\triangleright$
\$	$\triangleleft$	$\triangleleft$	$\triangleleft$	$\triangleleft$		Aceita

Basicamente, um analisador de precedência funciona da seguinte forma. Seja *a* o terminal mais ao topo da pilha, desprezando-se qualquer não terminal que possa ocorrer, e *b* o terminal sob a cabeça de leitura:

1. Se  $a \triangleleft b$  ou  $a \equiv b$ , então *empilha*;
2. Se  $a \triangleright b$ , então procure um *handle* na pilha, o qual deverá estar delimitado pelas relações  $\triangleleft$  e  $\triangleright$ , e o substitua pelo não terminal correspondente.

Deve ser observado que o *handle* vai desde o topo da pilha até o primeiro terminal *a*, inclusive, que tem abaixo de si um terminal *b*, tal que  $b \triangleleft a$ .

A Figura 26 ilustra como funciona um analisador deste tipo. Nela podemos identificar, de acordo com a tabela de precedência definida anteriormente, que sobre o topo da pilha temos o *handle*  $E \wedge E$ , pois, neste caso,  $\wedge$  é o terminal mais ao topo da pilha,  $\$$  é o terminal sob a cabeça de leitura e  $\wedge \succ \wedge$ . Logo, o *handle* vai desde o topo da pilha até o não terminal  $E$  que antecede o terminal  $\vee$ , pois  $\vee \prec \wedge$ .

Para ilustrar o funcionamento de um analisador de precedência de operadores, suponha que se deseje reconhecer a sentença:  $id_1 \vee id_2 \wedge id_3$ ; considerando a tabela de precedência de operadores definida anteriormente. Os movimentos efetuados pelo analisador sintático são indicados abaixo.

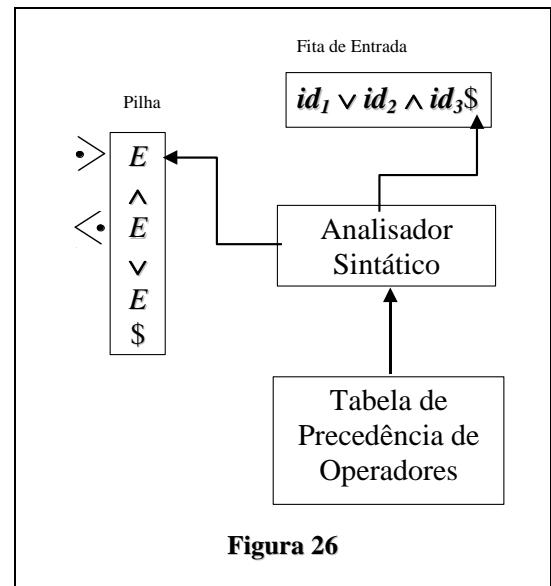


Figura 26

Passo	Pilha	Relação	Entrada	Handle	Ação
1.	\$		$id_1 \vee id_2 \wedge id_3 \$$		Empilhar
2.	\$ $id_1$		$\vee id_2 \wedge id_3 \$$	$id_1$	Reducir por $E \rightarrow id$
3.	\$ $E$		$\vee id_2 \wedge id_3 \$$		Empilhar
4.	\$ $E \vee$		$id_2 \wedge id_3 \$$		Empilhar
5.	\$ $E \vee id_2$		$\wedge id_3 \$$	$id_2$	Reducir por $E \rightarrow id$
6.	\$ $E \vee E$		$\wedge id_3 \$$		Empilhar
7.	\$ $E \vee E \wedge$		$id_3 \$$		Empilhar
8.	\$ $E \vee E \wedge id_3$		\$	$id_3$	Reducir por $E \rightarrow id$
9.	\$ $E \vee E \wedge E$		\$	$E \wedge E$	Reducir por $E \rightarrow E \wedge E$
10.	\$ $E \vee E$		\$	$E \vee E$	Reducir por $E \rightarrow E \vee E$
11.	\$ $E$		\$		Aceitar

Os movimentos de um analisador de precedência de operadores são efetuados de acordo com um algoritmo, o qual recebe como entrada uma tabela de precedência de operadores,  $t$ , e uma cadeia  $w$  a ser analisada. Este algoritmo é indicado a seguir.

```

Algoritmo PrecedênciaOperadores( $t, w$ )
  Início
    Seja  $w\$$  a cadeia de entrada
    Seja  $S$  o símbolo de partida
    Repita Sempre
      Se  $\$S$  está no topo da pilha
      e  $\$$  está sob a cabeça de leitura, então
      Aceite a cadeia de entrada e pare
    Senão
      Seja  $a$  o terminal ao topo da Pilha
      Seja  $b$  o terminal sob a cabeça de leitura
      Se  $a \triangleleft b$  ou  $a = b$ , então
        Empilhar  $b$ 
        Avançar a cabeça de leitura
      Senão
        Se  $a \triangleright b$ , então
          Repita
            Desempilhar
            Até encontrar a relação  $\triangleleft$  entre o terminal
            do topo da pilha e o último terminal
            desempilhado
            // Seja a produção  $X \rightarrow a$ , sendo  $a$  o
            // handle desempilhado
            // Neste caso,  $X$  deverá ser empilhado
            Empilhar o não terminal correspondente
        Senão
          Chamar a rotina de tratamento de erros
        Fim Se
      Fim Se
    Fim Repita
  Fim

```

### 4.7.2.2 Construção da Tabela de Precedência de Operadores

Para construir a tabela de precedência de operadores, podemos utilizar dois métodos, os quais nos permitem computar as relações de precedência entre os símbolos terminais de uma gramática de operadores: o método **intuitivo** e o **mecânico**.

#### Método Intuitivo:

Este método permite obter as relações de equivalência a partir do conhecimento prévio da associatividade e precedência dos operadores da gramática.

Sejam dois operadores  $\theta_1$  e  $\theta_2$ . O algoritmo funciona do seguinte modo, lembrando-se que o lado esquerdo da relação corresponde à linha da tabela (terminal mais ao topo da pilha) e o lado direito corresponde à coluna da tabela (terminal sob a cabeça de leitura):

1. Se o operador  $\theta_1$  tem maior precedência sobre o operador  $\theta_2$ , então fazemos  $\theta_1 \triangleright \theta_2$  e  $\theta_2 \triangleleft \theta_1$ .

2. Se os operadores  $\theta_1$  e  $\theta_2$  têm a mesma precedência, isto é, precedência igual, então:

2.1. Se são associativos à esquerda, então fazemos  $\theta_1 \cdot > \theta_2$  e  $\theta_2 \cdot > \theta_1$ ;

2.2. Se são associativos à direita, então fazemos  $\theta_1 \cdot < \theta_2$  e  $\theta_2 \cdot < \theta_1$ .

3. As relações entre os operadores e os demais tokens (operandos e delimitadores) são fixas. Para todo operador  $\theta$ , temos:

$$\begin{array}{ll} \theta \cdot < id & \text{e } id \cdot > \theta, \\ \theta \cdot < ( & \text{e } ( \cdot < \theta, \\ \theta \cdot > ) & \text{e } ) \cdot > \theta, \\ \theta \cdot > \$ & \text{e } \$ \cdot < \theta; \end{array}$$

4. As relações entre os tokens que não são operadores também são fixas:

$$\begin{array}{ll} ) \cdot > ) & \text{e } ( \cdot < (, \\ id \cdot > ) & \text{e } ( \cdot < id, \\ id \cdot > \$ & \text{e } \$ \cdot < id, \\ ) \cdot > \$ & \text{e } \$ \cdot < (, \\ (\cdot) & \text{e } (\cdot). \end{array}$$

Por exemplo, seja a gramática de operadores a seguir:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E ^ E \mid (E) \mid id$$

Têm-se as seguintes precedências e associatividade entre os operadores:

1.  $\wedge$ : tem maior precedência e é associativo à direita;
2.  $*$  e  $/$ : tem precedência intermediária e são associativos à esquerda;
3.  $+$  e  $-$ : tem menor precedência e são associativos à esquerda.

A tabela de precedência de operadores para esta gramática é indicada abaixo. As posições em branco são assinaladas como **erro**.

	<i>id</i>	$+$	$*$	$-$	$/$	$\wedge$	$($	$)$	$\$$
<i>id</i>		$\cdot >$		$\cdot >$	$\cdot >$				
$+$	$\cdot <$	$\cdot >$	$\cdot <$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot >$	$\cdot >$
$*$	$\cdot <$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot >$	$\cdot >$
$-$	$\cdot <$	$\cdot >$	$\cdot <$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot >$	$\cdot >$
$/$	$\cdot <$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot >$	$\cdot >$
$\wedge$	$\cdot <$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot >$	$\cdot >$
$($	$\cdot <$	$\cdot =$							
$)$		$\cdot >$		$\cdot >$	$\cdot >$				
$\$$	$\cdot <$		Aceita						

Observe que na gramática acima a produção:  $E \rightarrow - E$ ; foi eliminada, impedindo assim que o operador “-” seja utilizado como binário e unário. Para termos os dois significados para o mesmo operador, o analisador léxico deve distinguir entre os dois tipos, por exemplo, representando o operador unário por “-u”, o qual é o caso quando o operador é precedido de outro operador, de abre parênteses, de vírgula ou de um símbolo de atribuição.

## Método Mecânico:

Este método obtém as relações de precedência diretamente a partir da gramática de operadores, a qual não pode ser ambígua.

O algoritmo funciona do seguinte modo:

1. Defina  $leading(A)$ <sup>2</sup> para o não terminal  $A$  como sendo o conjunto dos terminais  $a$ , tal que  $a$  é o terminal **mais à esquerda** em alguma forma sentencial derivada de  $A$ ;
2. Defina  $trailing(A)$ <sup>3</sup> para o não terminal  $A$  como sendo o conjunto dos terminais  $a$ , tal que  $a$  é o terminal **mais à direita** em alguma forma sentencial derivada de  $A$ ;
3. Para cada dois terminais  $a$  e  $b$ , temos:
  - 3.1.  $a \stackrel{\bullet}{=} b$ , se existe um lado direito de produção da forma  $\alpha a \beta b \gamma$ , onde  $\beta$  é  $\epsilon$  ou é um não terminal e  $\alpha$  e  $\gamma$  são arbitrários;
  - 3.2.  $a \triangleleft b$ , se existe um lado direito de produção da forma  $\alpha a A \gamma$ , e  $b$  está em  $leading(A)$ ;
  - 3.3.  $a \triangleright b$ , se existe um lado direito de produção da forma  $\alpha A b \gamma$  e  $a$  está em  $trailing(A)$ ;
  - 3.4.  $\$ \triangleleft b$ , se  $S$  é o símbolo de partida da gramática e, para qualquer  $b$ ,  $b \in leading(S)$ ;
  - 3.5.  $a \triangleright \$$ , se  $S$  é o símbolo de partida da gramática e, para qualquer  $a$ ,  $a \in trailing(S)$ ;

Como exemplo de aplicação do algoritmo, suponha a gramática de operadores:

$$E \rightarrow E + E \mid E * E \mid E ^ E \mid ( E ) \mid id$$

Esta gramática é ambígua. Portanto, não podemos aplicar o algoritmo sem antes eliminarmos a ambiguidade. Isto pode ser feito transformando as produções para

<sup>2</sup> A tradução para **leading**, neste contexto, seria **à frente**. Portanto,  $leading(A)$  seria o conjunto formado por todos os símbolos terminais à frente de uma forma sentencial derivada de  $A$ , isto é, os primeiros símbolos terminais (não confundir com o conjunto  $Primeiro(A)$ ).

<sup>3</sup> A tradução para **trailing**, neste contexto, seria **finaliza**. Portanto,  $trailing(A)$  seria o conjunto formado por todos os símbolos terminais que finalizam uma forma sentencial derivada de  $A$ , isto é, os últimos símbolos terminais.

expressar a precedência e a associatividade dos operadores. Consegue-se isto introduzindo um símbolo não terminal para cada nível de precedência. No caso de associatividade à esquerda, a avaliação será feita da esquerda para a direita. Já para a associatividade à direita, a avaliação será feita da direita para a esquerda. Portanto, temos a seguinte gramática:

$E \rightarrow E + T \mid T$	// Operador de menor precedência e associativo à esquerda
$T \rightarrow T * F \mid F$	// Operador de precedência intermediária e associativo à esquerda
$F \rightarrow P ^ F \mid P$	// Operador de maior precedência e associativo à direita
$P \rightarrow (E) \mid id$	// Operandos

Os conjuntos *leading* e *trailing* são indicados pela tabela abaixo.

	<i>Leading</i>	<i>Trailing</i>
$E$	$+, *, ^, (, id$	$+, *, ^, ), id$
$T$	$*, ^, (, id$	$*, ^, ), id$
$F$	$^, (, id$	$^, ), id$
$P$	$(, id$	$), id$

Agora, calculemos as relações de precedência:

1.  $a \stackrel{\bullet}{=} b$ : examinar todos os lados direitos das produções procurando por formas  $\alpha a \beta b \gamma$ , onde  $\beta$  é  $\epsilon$  ou é um não terminal e  $\alpha$  e  $\gamma$  são arbitrários. Neste caso, a única produção que se enquadra é  $P \rightarrow (E)$ . Portanto, temos:

$(\stackrel{\bullet}{=})$

2.  $a \stackrel{\bullet}{<} b$ : examinar todos os lados direitos das produções procurando por formas  $\alpha a A \gamma$ , tal que  $b$  está em *leading(A)*. Neste caso, temos as produções:

$P \rightarrow E + T$  (par  $+ T$ ): o que nos dá a relação:  $+ \stackrel{\bullet}{<} \{ *, ^, (, id \}$

$T \rightarrow T * F$  (par  $* F$ ): o que nos dá a relação:  $* \stackrel{\bullet}{<} \{ ^, (, id \}$

$F \rightarrow P ^ F$  (par  $^ F$ ): o que nos dá a relação:  $^ \stackrel{\bullet}{<} \{ ^, (, id \}$

$F \rightarrow (E)$  (par ' $E$ ): o que nos dá a relação:  $( \stackrel{\bullet}{<} \{ +, *, ^, (, id \}$

3.  $a \stackrel{\bullet}{>} b$ : examinar todos os lados direitos das produções procurando por formas  $\alpha A b \gamma$ , tal que  $a$  está em *trailing(A)*. Neste caso, temos as produções:

$P \rightarrow E + T$  (par  $E +$ ): o que nos dá a relação:  $\{ +, *, ^, (, id \} \stackrel{\bullet}{>} +$

$T \rightarrow T * F$  (par  $T *$ ): o que nos dá a relação:  $\{ *, ^, (, id \} \stackrel{\bullet}{>} *$

$F \rightarrow P ^ F$  (par  $P ^$ ): o que nos dá a relação:  $\{ \}, id \} \stackrel{\bullet}{>} ^$

$F \rightarrow (E)$  (par  $F'$ ): o que nos dá a relação:  $\{+, *, ^, (), id\} \cdot > )$

4.  $\$ \leq^* b$ , se  $S$  é o símbolo de partida da gramática e, para qualquer  $b$ ,  $b \in leading(S)$ . Como  $\$$  tem precedência menor que qualquer terminal em  $leading(E)$ , então temos:

$\$ \leq^* \{+, *, ^, (), id\}$

5.  $a \cdot > \$$ , se  $S$  é o símbolo de partida da gramática e, para qualquer  $a$ ,  $a \in trailing(S)$ . Como qualquer terminal em  $trailing(E)$  possui precedência maior que  $\$$ , então temos:

$\{+, *, ^, (), id\} \cdot > \$$

A tabela de precedência obtida a partir deste algoritmo é indicada abaixo.

Tabela 8: Tabela de precedência de operadores, obtida pelo método mecânico.

	<i>id</i>	+	*	^	(	)	\$
<i>id</i>		$\cdot >$	$\cdot >$	$\cdot >$		$\cdot >$	$\cdot >$
+	$\leq^*$	$\cdot >$	$\leq^*$	$\leq^*$	$\leq^*$	$\cdot >$	$\cdot >$
*	$\leq^*$	$\cdot >$	$\cdot >$	$\leq^*$	$\leq^*$	$\cdot >$	$\cdot >$
^	$\leq^*$	$\cdot >$	$\cdot >$	$\leq^*$	$\leq^*$	$\cdot >$	$\cdot >$
(	$\leq^*$	$\leq^*$	$\leq^*$	$\leq^*$	$\leq^*$	$\equiv$	
)		$\cdot >$	$\cdot >$	$\cdot >$		$\cdot >$	$\cdot >$
\$	$\leq^*$	$\leq^*$	$\leq^*$	$\leq^*$	$\leq^*$		Aceita

### 4.7.2.3 Funções de Precedência

Podemos usar funções para indicar ao analisador a precedência de operadores, substituindo assim a tabela de precedência. As funções  $f$  e  $g$  mapeiam terminais em inteiros. Estes inteiros indicam a precedência. Sejam  $a$  e  $b$  terminais:

- $f(a) < g(b)$  sempre que  $a \leq^* b$ ;
- $f(a) = g(b)$  sempre que  $a \equiv b$ ;
- $f(a) > g(b)$  sempre que  $a \cdot > b$ .

O método apresenta algumas desvantagens, como não representar as entradas de erros. Além disto, nem sempre é possível obter as funções  $f$  e  $g$ .

O algoritmo para obter as funções de precedência é o seguinte:

1. Criar símbolos  $fa$  e  $ga$  para cada elemento de  $V_T$  e  $\$$ .

2. Distribuir os símbolos criados em grupos, tal que:
    - 2.1. Se  $a \stackrel{\bullet}{=} b$ , então  $fa$  e  $gb$  estão no mesmo grupo;
    - 2.2. Se  $a \stackrel{\bullet}{=} b$  e  $c \stackrel{\bullet}{=} b$ ,  $fa$  e  $fc$  deverão ficar no mesmo grupo que  $gb$ ;
    - 2.3. Se, ainda,  $c \stackrel{\bullet}{=} d$ , então  $fa$ ,  $fc$ ,  $gb$  e  $gd$  deverão ficar no mesmo grupo, mesmo que  $a \stackrel{\bullet}{=} d$  não ocorra.
  3. Gerar um grafo dirigido cujos **nós** são os grupos formados anteriormente. Para quaisquer  $a$  e  $b$ :
    - 3.1. Se  $a \stackrel{\bullet}{<} b$ , construir um arco do grupo  $fa$  para o grupo  $gb$ ; e
    - 3.2. Se  $a \stackrel{\bullet}{>} b$  fazer um arco de  $gb$  para  $fa$ .
  4. Se o grafo contiver ciclos, então as funções de precedência não existem. Se não houver ciclos, então  $f(a)$  é igual ao caminho mais longo iniciando em  $fa$  e  $g(a)$  é igual ao caminho mais longo iniciando em  $ga$ .

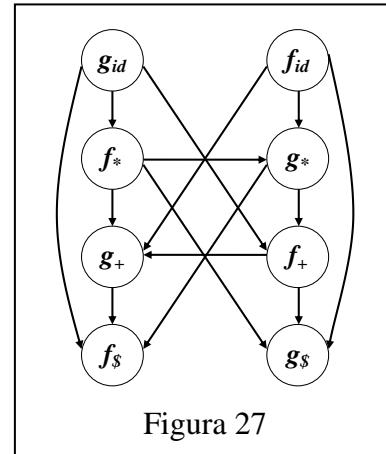
Por exemplo, seja a gramática:

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid id \end{array}$$

A tabela de precedência de operadores para esta gramática é indicada pela Tabela 9.

**Tabela 9: Tabela de precedência de operadores.**

	<i>id</i>	+	*	(	)	\$
<i>id</i>		•>	•>		•>	•>
+	<•	•>	<•	<•	•>	•>
*	<•	•>	•>	<•	•>	•>
(	<•	<•	<•	<•	•=	
)		•>	•>		•>	•>
\$	<•	<•	<•	<•		Aceita



Pelo algoritmo proposto temos o grafo indicado pela

Figura 27. As funções  $f$  e  $g$  encontram-se indicadas na Tabela 10.

**Tabela 10: Funções de precedência f e g.**

	+	*	(	)	id	\$
f	2	4	0	4	4	0
g	1	3	5	0	5	0

## 4.7.3 Análise LR( $k$ )

Analisa-se a seguir uma técnica eficiente de análise ascendente, a qual pode ser utilizada para decompor uma ampla classe de gramáticas livres de contexto. A técnica é chamada de análise  **$LR(k)$** , onde o  **$L$**  significa que a entrada é lida da esquerda para a direita (*left-to-right*), o  **$R$**  significa a construção da derivação mais à direita em reverso (*rightmost-derivation*), e o  **$k$**  é o número de símbolos de entrada que devem ser lidos (*lookahead*) para que o analisador possa tomar decisões. Dentre as vantagens destes analisadores destacam-se:

1. São capazes de reconhecer, praticamente, todas as estruturas sintáticas definidas por uma gramática livre de contexto;
2. É o método mais geral e eficiente que o de precedência de operadores e qualquer outro tipo de analisador ***shift-reduce***, podendo ser implementado com o mesmo grau de eficiência;
3. São capazes de descobrir erros sintáticos tão cedo quanto possível, numa varredura de entrada da esquerda para direita.

A principal desvantagem deste método está na dificuldade para se construir um analisador sintático  **$LR$**  manualmente, sendo normalmente, necessário a utilização de ferramentas especializadas, denominadas de geradores de analisadores (*parser-generator*), como por exemplo o YACC (*Yet Another Compiler-Compiler*) e o BISON.

Há basicamente três tipos de analisadores  **$LR$** :

1.  **$SLR$  (Simple  $LR$ ):** de fácil implementação, porém aplicáveis a uma classe restrita de gramáticas;
2.  **$LR$  Canônicos:** são os mais poderosos, podendo ser aplicados a um grande número de linguagens livres de contexto; e
3.  **$LALR$  (Look Ahead  $LR$ ):** de nível intermediário e implementação eficiente, que funciona para a maioria das linguagens de programação – é o método utilizado pelo YACC.

Basicamente, um analisador  **$LR$**  é composto por uma *fita de entrada*, uma *pilha*, um *programa* e uma *tabela sintática*, a qual é composta por duas partes: ***ação*** (*action*) e ***desvio*** (*goto*). O *programa* é o mesmo para os três tipos de analisadores  **$LR$** , apenas a *tabela sintática* muda de um para outro. A

Figura 28 ilustra a forma esquemática de um analisador  **$LR$** . A fita de entrada mostra a sentença a ser analisada, finalizada pelo marcador  $\$$ . A pilha armazena os símbolos  $X_i$  da gramática intercalados por estados  $s_m$  do analisador, formando o par  $X_m s_m$ . O símbolo na base da pilha é sempre  $S_0$ , ou seja, o estado inicial do analisador.

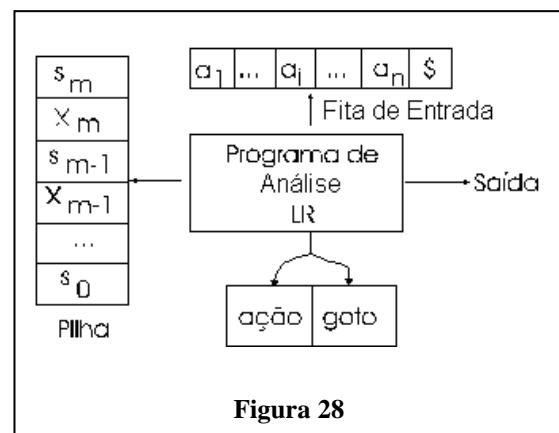


Figura 28

Na tabela sintática, a parte **ação** contém as seguintes ações, as quais estão associadas aos estados e aos símbolos terminais lidos da entrada:

1. Empilhar  $s$ , onde  $s$  é um estado;
2. Reduzir usando uma produção  $A \rightarrow \beta$ ;
3. Aceitar; ou
4. Erro.

A parte **desvio** contém transições de estados em relação aos símbolos não terminais da gramática.

Basicamente, o analisador funciona como se segue. Seja  $s_m$  o estado no topo da pilha e  $a_i$  o terminal sob a cabeça de leitura. O analisador consulta a tabela **Ação**[ $s_m, a_i$ ], a qual pode assumir um dos valores:

1. Empilha  $s$ : causa o empilhamento de  $a_i s$ ;
2. Reduzir  $A \rightarrow \beta$ : causa o desempilhamento de  $2r$  símbolos, onde  $r = |\beta|$ , e o empilhamento de  $A s_j$ , onde  $s_j$  resulta da consulta à tabela **Desvio**[ $s_{m-r}, A$ ];
3. Aceita: o analisador reconhece a sentença como válida; e
4. Erro: o analisador para a execução, identificando um erro sintático.

O funcionamento de um analisador LR pode ser entendido considerando as transformações que ocorrem na pilha e na fita de entrada, denominada de configuração do analisador, a qual é representada pelo par (*pilha, fita*). Portanto, a configuração inicial de um analisador LR é dada por  $(s_0, a_1 a_2 \dots a_n \$)$ .

Considerando a configuração  $(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$ , têm-se a seguinte configuração após a aplicação de cada ação:

1. Se  $\text{Ação}[s_m, a_i] = \text{empilhar } s$ , então tem-se a nova configuração:  $(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m a_i s, a_{i+1} \dots a_n \$)$ . Observe que a cabeça de leitura é avançada para o próximo símbolo na entrada;
2. Se  $\text{Ação}[s_m, a_i] = \text{reduzir } A \rightarrow \beta$ , então tem-se a nova configuração:  $(s_0 X_1 s_1 X_2 s_2 \dots X_{m-r} s_{m-r} A s, a_i a_{i+1} \dots a_n \$)$ ; onde  $s = \text{desvio}[s_{m-r}, A]$  e  $r = |\beta|$ . Nesse caso, o analisador desempilha  $2r$  símbolos da pilha, deixando no topo  $s_{m-r}$  e empilha  $A$  e  $s$ . Observe que a cabeça de leitura não é avançada, mantendo o mesmo símbolo à entrada;
3. Se  $\text{ação}[s_m, a_i] = \text{aceitar}$ , então o analisador conclui com sucesso; e
4. Se  $\text{ação}[s_m, a_i] = \text{erro}$ , então o analisador pára emitindo uma mensagem de erro, ou chama uma rotina de tratamento de erro.

Como exemplo, considere a gramática abaixo:

- |                          |                          |
|--------------------------|--------------------------|
| 1. $E \rightarrow E + T$ | 4. $T \rightarrow F$     |
| 2. $E \rightarrow T$     | 5. $F \rightarrow ( E )$ |
| 3. $T \rightarrow T * F$ | 6. $F \rightarrow id$    |

A tabela sintática  $LR$  para esta gramática é indicada a seguir, onde os símbolos  $si$  e  $ri$  significam, respectivamente, empilhar  $i$  (shift  $i$ ) e reduzir  $i$  (reduce  $i$ ). As entradas não definidas são de erro.

Estado	Ação						Desvio		
	<i>id</i>	+	*	(	)	\$	<i>E</i>	<i>T</i>	<i>F</i>
0	<i>s5</i>			<i>s4</i>			1	2	3
1		<i>s6</i>				<i>aceita</i>			
2		<i>r2</i>	<i>s7</i>		<i>r2</i>	<i>r2</i>			
3		<i>r4</i>	<i>r4</i>		<i>r4</i>	<i>r4</i>			
4	<i>s5</i>			<i>s4</i>			8	2	3
5		<i>r6</i>	<i>r6</i>		<i>r6</i>	<i>r6</i>			
6	<i>s5</i>			<i>s4</i>			9	3	
7	<i>s5</i>			<i>s4</i>					10
8		<i>s6</i>			<i>s11</i>				
9		<i>r1</i>	<i>s7</i>		<i>r1</i>	<i>r1</i>			
10		<i>r3</i>	<i>r3</i>		<i>r3</i>	<i>r3</i>			
11		<i>r5</i>	<i>r5</i>		<i>r5</i>	<i>r5</i>			

As configurações do analisador no reconhecimento da entrada:  $id_1 * id_2 + id_3$ ; são indicadas abaixo.

Passo	Pilha	Fita de Entrada	Ação
1.	0	$id_1 * id_2 + id_3 \$$	Empilhar
2.	0 $id_1$ 5	$* id_2 + id_3 \$$	Reducir $F \rightarrow id$
3.	0 $F$ 3	$* id_2 + id_3 \$$	Reducir $T \rightarrow F$
4.	0 $T$ 2	$* id_2 + id_3 \$$	Empilhar
5.	0 $T$ 2 * 7	$id_2 + id_3 \$$	Empilhar
6.	0 $T$ 2 * 7 $id_2$ 5	$+ id_3 \$$	Reducir $F \rightarrow id$
7.	0 $T$ 2 * 7 $F$ 10	$+ id_3 \$$	Reducir $T \rightarrow T * F$
8.	0 $T$ 2	$+ id_3 \$$	Reducir $E \rightarrow T$
9.	0 $E$ 1	$+ id_3 \$$	Empilhar
10.	0 $E$ 1 + 6	$id_3 \$$	Empilhar
11.	0 $E$ 1 + 6 $id_3$ 5	$\$$	Reducir $F \rightarrow id$
12.	0 $E$ 1 + 6 $F$ 3	$\$$	Reducir $T \rightarrow F$
13.	0 $E$ 1 + 6 $T$ 9	$\$$	Reducir $E \rightarrow E + T$
14.	0 $E$ 1	$\$$	Aceita

O algoritmo para um analisador LR é indicado a seguir, o qual recebe como entrada um tabela sintática  $t$  e a cadeia  $w$  a ser analisada.

```

Algoritmo AnaliseLR( $t, w$ )
Inicio
    Seja  $w\$$  a cadeia de entrada
    Empilhar  $s_0$ , o estado de partida, na pilha
    Faça  $ip$  apontar para o primeiro símbolo de  $w\$$ 
    Repita sempre
        Seja  $s$  o estado ao topo da pilha
        Seja  $a$  o símbolo apontado por  $ip$ 
        Se  $ação[s, a] = \text{empilhar } x$  então
            Empilhar  $a$ 
            Empilhar  $x$ 
            Avançar  $ip$  para o próximo símbolo da entrada
        Senão
            Se  $ação[s, a] = \text{reduzir } A \rightarrow \beta$  então
                Desempilhar  $2*|\beta|$  símbolos da pilha
                Seja  $x$  o estado ao topo da pilha
                Empilhar  $A$ 
                Empilhar  $\text{desvio}[x, A]$ 
                // Neste ponto executa-se qualquer ação
                // desejada, como por exemplo escrever
                // a redução  $A \rightarrow \beta$ 
            Senão
                Se  $ação[s, a] = \text{aceitar}$  então
                    Retornar
                Senão
                    Chamar rotina de tratamento de erro
                Fim Se
            Fim Se
        Fim Se
    Fim Repita
Fim

```

## 4.7.4 Construção de Analisadores Sintáticos SLR

Conforme citado anteriormente, o que varia em um método de análise **LR** é a sua tabela sintática. Por sua facilidade de implementação, apresentaremos agora o algoritmo para construção da tabela sintática para o método **SLR**.

A construção de analisadores SLR baseia-se no que se denomina **conjunto canônico de itens LR(0)**. Um **item LR(0)** (ou simplesmente **item**) de uma gramática  $G$  é uma produção de  $G$  com um ponto em alguma posição do lado direito da produção. Por exemplo, para a produção  $A \rightarrow X Y Z$  tem-se quatro itens:  $A \rightarrow \bullet X Y Z$ ,  $A \rightarrow X \bullet Y Z$ ,  $A \rightarrow X Y \bullet Z$ ,  $A \rightarrow X Y Z \bullet$ . A produção  $A \rightarrow \epsilon$  gera apenas um ítem:  $A \rightarrow \bullet$ .

Intuitivamente podemos dizer que o **item** indica o quanto de uma produção já foi visto até um determinado momento no processo de análise.

A ideia central no método **SLR** é de construir, a partir de uma gramática, um autômato finito determinístico que reconheça **prefixos viáveis**. **Prefixos viáveis** são formas sentenciais direitas que aparecem no topo da pilha de um analisador do tipo empilar-reduzir. Os **itens** são agrupados em conjuntos que dão origem aos estados do analisador **SLR**. São estes **estados** que irão reconhecer os prefixos viáveis.

A construção do **conjunto canônico de ítems LR(0)**, ou simplesmente **conjunto**

**canônico LR(0)**, para uma gramática  $G$ , requer duas operações e as funções **closure (fechamento)** e **goto (desvio)**:

1. Adicionar à gramática  $G$  a produção  $S' \rightarrow S$ , onde  $S$  é o símbolo inicial da gramática  $G$  e  $S'$  é um novo símbolo não terminal, gerando a gramática  $G'$ , denominada de **gramática aumentada**; e
2. Computar a função **closure** e **goto** para  $G'$ .

O objetivo de se acrescentar a produção  $S' \rightarrow S$  à  $G$  é indicar o momento em que o processo de análise acaba e aceita a sentença de entrada. Ou seja, isto ocorre quando o analisador está por reduzir  $S' \rightarrow S$ .

#### 4.7.4.1 Operação Closure

Se  $I$  é um **conjunto de itens LR(0)** de  $G$ , então **closure( $I$ )** é o **conjunto de itens** construído a partir de  $I$  pelas seguintes regras:

1. Todo **item** de  $I$  é adicionado em **closure( $I$ )**; e
2. Se  $A \rightarrow \alpha \cdot B \beta$  está em **closure( $I$ )** e  $B \rightarrow \gamma$  pertence a  $G$ , então adicione o **item**  $B \rightarrow \cdot \gamma$  a  $I$ , se ele já não estiver lá. Repita esta regra até que nenhum novo **item** possa ser adicionado.

Intuitivamente o item  $A \rightarrow \alpha \cdot B \beta$  em **closure( $I$ )** indica que, em algum ponto do processo de análise, esperamos ver uma cadeia derivável a partir de  $B\beta$ . Se  $B \rightarrow \gamma$  é uma produção de  $G$ , então também esperamos ver uma cadeia derivável de  $\gamma$  neste ponto. Por esta razão incluímos  $B \rightarrow \cdot \gamma$  em **closure( $I$ )**.

Por exemplo, seja a gramática aumentada  $G$ :

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow ( E ) \mid id \end{aligned}$$

Se  $I = \{E' \rightarrow \cdot E\}$ , então **closure( $I$ )** contém os seguintes itens:

$$\begin{aligned} E' &\rightarrow \cdot E \\ E &\rightarrow \cdot E + T \\ E &\rightarrow \cdot T \\ T &\rightarrow \cdot T * F \\ T &\rightarrow \cdot F \\ F &\rightarrow \cdot ( E ) \\ F &\rightarrow \cdot id \end{aligned}$$

## 4.7.4.2 Operação Goto

Informalmente,  $\text{goto}(\mathcal{I}, X)$ , avanço de  $I$  através de  $X$ , consiste em coletar as produções com ponto no lado esquerdo de  $X$ ,  $X$  terminal ou não terminal ( $X \in V_N \cup V_T$ ), avançar o ponto de uma posição e obter a função ***closure*** deste conjunto.

Formalmente, para  $X \in (V_N \cup V_T)$ , a função  $\text{goto}(\mathcal{I}, X)$  é a função ***closure*** do conjunto dos ***ítems***  $\{A \rightarrow \alpha X \cdot \beta\}$  tal que  $\{A \rightarrow \alpha \cdot X \beta\} \in \mathcal{I}$ .

Por exemplo, seja a gramática aumentada  $G$  usada anteriormente. Se  $\mathcal{I} = \{E' \rightarrow E \cdot, E \rightarrow E \cdot + T\}$ , então  $\text{goto}(\mathcal{I}, +) = \text{closure}(\{E \rightarrow E + \cdot T\})$ , que é:

$$\begin{aligned} E &\rightarrow E + \cdot T \\ T &\rightarrow \cdot T * F \\ T &\rightarrow \cdot F \\ F &\rightarrow \cdot ( E ) \\ F &\rightarrow \cdot id \end{aligned}$$

## 4.7.4.3 Algoritmo para a Construção do Conjunto de Ítems LR(0)

Para uma gramática  $G$ , o **conjunto canônico de ítems LR(0)**, referido por  $C$ , é obtido pelo algoritmo abaixo.

```
Algoritmo Closure(G)
  Inicio
    C <- I0 = closure({S' → •S})
    repita
      Para cada conjunto de ítems I em C e
      Para cada símbolo X de G, tal que
      goto(I, X) ≠ ∅ e goto(I, X) ∈ C Faça
        Adicione goto(I, X) em C
      Até que nenhum conjunto de ítems possa ser adicionado à C
  Fim
```

Por exemplo, seja a gramática aumentada  $G$  e os conjuntos Seguinte:

0.  $E' \rightarrow E$
1.  $E \rightarrow E + T$
2.  $E \rightarrow T$
3.  $T \rightarrow T * F$
4.  $T \rightarrow F$
5.  $F \rightarrow ( E )$
6.  $F \rightarrow id$

### Seguinte:

$$\begin{aligned} S(E') &= \{ \$ \} \\ S(E) &= \{ \$, +, ) \} \\ S(T) &= \{ \$, +, ), * \} \\ S(F) &= \{ \$, +, ), * \} \end{aligned}$$

O **conjunto canônico de itens LR(0)** para  $G$  é dado por:

1. Inicialização:  $C = \{I_0 = \text{closure}(\{E' \rightarrow^* E\})\}$

$$I_0 = \{E' \rightarrow^* E, E \rightarrow^* E + T, E \rightarrow^* T, T \rightarrow^* T * F, T \rightarrow^* F, F \rightarrow^* (E), F \rightarrow^* id\}$$

2. Repita: Para todo  $I \in C$  e  $X \in G$ , calcular  $\text{goto}(I, X)$  e adicionar a  $C$

$$I_1 = \text{goto}(I_0, E) = \text{closure}(\{E' \rightarrow E \bullet, E \rightarrow E \bullet + T\}) = \{E' \rightarrow E \bullet, E \rightarrow E \bullet + T\};$$

$$I_2 = \text{goto}(I_0, T) = \text{closure}(\{E \rightarrow T \bullet, T \rightarrow T \bullet * F\}) = \{E \rightarrow T \bullet, T \rightarrow T \bullet * F\};$$

$$I_3 = \text{goto}(I_0, F) = \text{closure}(\{T \rightarrow F \bullet\}) = \{T \rightarrow F \bullet\};$$

$$I_4 = \text{goto}(I_0, '(') = \text{closure}(\{F \rightarrow (( \bullet E))\}) = \{F \rightarrow (( \bullet E)), E \rightarrow \bullet E + T, E \rightarrow \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F, F \rightarrow \bullet (E), F \rightarrow \bullet id\}$$

$$I_5 = \text{goto}(I_0, id) = \text{closure}(\{F \rightarrow id \bullet\}) = \{F \rightarrow id \bullet\}$$

$$I_6 = \text{goto}(I_1, '+') = \text{closure}(\{E \rightarrow E + \bullet T\}) = \{E \rightarrow E + \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F, F \rightarrow \bullet (E), F \rightarrow \bullet id\}$$

$$I_7 = \text{goto}(I_2, '*') = \text{closure}(\{T \rightarrow T * \bullet F\}) = \{T \rightarrow T * \bullet F, F \rightarrow \bullet (E), F \rightarrow \bullet id\}$$

$$I_8 = \text{goto}(I_4, E) = \text{closure}(\{F \rightarrow (E \bullet), E \rightarrow E \bullet + T\}) = \{F \rightarrow (E \bullet), E \rightarrow E \bullet + T\}$$

$$\text{goto}(I_4, T) = \text{closure}(\{E \rightarrow T \bullet, T \rightarrow T \bullet * F\}) = I_2, \text{ o qual já foi incluído}$$

$$\text{goto}(I_4, F) = \text{closure}(\{T \rightarrow F \bullet\}) = I_3, \text{ o qual já foi incluído}$$

$$\text{goto}(I_4, '(') = \text{closure}(\{F \rightarrow (( \bullet E))\}) = I_4, \text{ o qual já foi incluído}$$

$$\text{goto}(I_4, id) = \text{closure}(\{F \rightarrow id \bullet\}) = I_5, \text{ o qual já foi incluído}$$

$$I_9 = \text{goto}(I_6, T) = \text{closure}(\{E \rightarrow E + T \bullet, T \rightarrow T \bullet * F\}) = \{E \rightarrow E + T \bullet, T \rightarrow T \bullet * F\}$$

$$\text{goto}(I_6, F) = \text{closure}(\{T \rightarrow F \bullet\}) = I_3, \text{ o qual já foi incluído}$$

$$\text{goto}(I_6, '(') = \text{closure}(\{F \rightarrow (( \bullet E))\}) = I_4, \text{ o qual já foi incluído}$$

$$\text{goto}(I_6, id) = \text{closure}(\{F \rightarrow id \bullet\}) = I_5, \text{ o qual já foi incluído}$$

$$I_{10} = \text{goto}(I_7, F) = \text{closure}(\{T \rightarrow T * F \bullet\}) = \{T \rightarrow T * F \bullet\}$$

$$\text{goto}(I_7, '(') = \text{closure}(\{F \rightarrow (( \bullet E))\}) = I_4, \text{ o qual já foi incluído}$$

$$\text{goto}(I_7, id) = \text{closure}(\{F \rightarrow id \bullet\}) = I_5, \text{ o qual já foi incluído}$$

$$I_{11} = \text{goto}(I_8, ')') = \text{closure}(\{F \rightarrow ((E) \bullet)\}) = \{F \rightarrow ((E) \bullet)\}$$

$$\text{goto}(I_8, '+') = \text{closure}(\{E \rightarrow E + \bullet T\}) = I_6, \text{ o qual já foi incluído}$$

$$\text{goto}(I_9, '**') = \text{closure}(\{T \rightarrow T * \bullet F\}) = I_7, \text{ o qual já foi incluído}$$

Como resultado do cálculo obtemos a Tabela 11.

Tabela 11: Tabela com os conjuntos canônicos de itens LR(0).

Conjunto Canônicos de Itens LR(0)	
Conjunto	Itens
$I_0$	$E' \rightarrow \bullet E, E \rightarrow \bullet E + T, E \rightarrow \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F, F \rightarrow \bullet (E), F \rightarrow \bullet id$
$I_1$	$E' \rightarrow E \bullet, E \rightarrow E \bullet + T$
$I_2$	$E \rightarrow T \bullet, T \rightarrow T \bullet * F$
$I_3$	$T \rightarrow F \bullet$
$I_4$	$F \rightarrow (\bullet E), E \rightarrow \bullet E + T, E \rightarrow \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F, F \rightarrow \bullet (E), F \rightarrow \bullet id$
$I_5$	$F \rightarrow id \bullet$
$I_6$	$E \rightarrow E + \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F, F \rightarrow \bullet (E), F \rightarrow \bullet id$
$I_7$	$T \rightarrow T * \bullet F, F \rightarrow \bullet (E), F \rightarrow \bullet id$
$I_8$	$F \rightarrow (E \bullet), E \rightarrow E \bullet + T$
$I_9$	$E \rightarrow E + T \bullet, T \rightarrow T \bullet * F$
$I_{10}$	$T \rightarrow T * F \bullet$
$I_{11}$	$F \rightarrow (E) \bullet$

A aplicação do algoritmo para a construção do conjunto canônico de itens LR(0) pode ser visualizada pelo diagrama de transição da Figura 29, correspondente a um autômato finito determinístico.

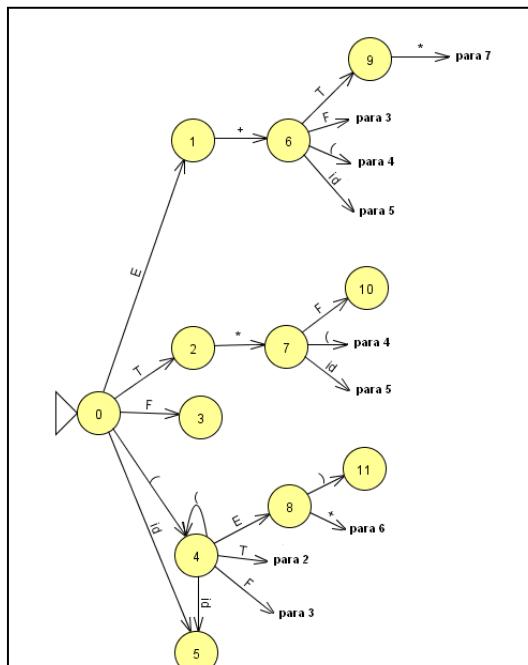


Figura 29: Diagrama de transição para o conjunto canônico de itens LR(0).

## 4.7.4.4 Construção da Tabela Sintática SLR

Dada uma gramática  $G$ , obtém-se  $G'$ , aumentando  $G$  com a produção  $S' \rightarrow S$ , onde  $S$  é o símbolo de partida de  $G$ . A partir de  $G'$ , determina-se o **conjunto canônico de itens LR(0)**. Finalmente, constrói-se as tabelas **ação** e **desvio**, conforme indicado pelo algoritmo abaixo.

1. Construa o conjunto canônico  $C = \{I_0, I_1, \dots, I_n\}$ ;
2. Cada linha da tabela corresponde a um estado  $i$  do analisador, o qual é construído a partir de  $I_i$ ,  $0 \leq i \leq n$ ;
3. A linha da tabela **ação** para o estado  $i$  é determinada pelas regras abaixo. Entretanto, se houver algum conflito na aplicação destas regras, então a gramática não é **SLR(1)**, e o algoritmo falha:
  - a) Se  $A \rightarrow \alpha \cdot a\beta$  está em  $I_i$  e  $\text{goto}(I_i, a) = I_j$ ,  $a \in V_T$ , então  $\text{ação}[i, a] = \text{empilhar } j$ ;
  - b) Se  $A \rightarrow \alpha \cdot$  está em  $I_i$ ,  $A \neq S'$ , então  $\text{ação}[i, a] = \text{reduzir } A \rightarrow \alpha$ , para todo  $a \in \text{Seguinte}(A)$ ;
  - c) Se  $S' \rightarrow S \cdot$  está em  $I_i$ , então defina  $\text{ação}[i, \$] = \text{aceita}$ .
4. A linha da tabela **desvio** para o estado  $i$  é determinada do seguinte modo:
  - a) Para todos os não terminais  $A$ , se  $\text{goto}(I_i, A) = I_j$ , então  $\text{desvio}[i, A] = j$ .
5. As entradas não definidas são erros;
6. O estado de partida é o estado  $0$ .

Pelo algoritmo, a tabela sintática para a gramática de expressões aumentada  $G$ , definida anteriormente, é:

Estado	id	Ação					Desvio		
		+	*	(	)	\$	E	T	F
0	s5			s4			1	2	3
1		s6				aceita			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4			9	3	
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

## 4.8 Geradores Automáticos de Analisadores Sintáticos

Conforme verificado, é possível escrever um analisador sintático descendente manualmente. Entretanto, escrever um analisador ascendente já é uma tarefa mais desgastante.

De qualquer modo, para ambos os casos existem geradores automáticos para analisadores sintáticos, tanto descendentes como ascendentes. Como exemplos de analisadores descendentes temos LLGen e Coco/R, dois geradores automáticos para gramáticas LL(1), e o ANTLR, o qual gera analisadores sintáticos para gramáticas ALL(\*). Já para os analisadores ascendentes, YACC e BISON são os geradores automáticos mais conhecidos e utilizados, trabalhando com gramáticas LALR(1). Um outro gerador extremamente interessante é o GOLD, também para gramáticas LALR(1). Enquanto o YACC e o BISON leem a especificação BNF da gramática e geram o código fonte para o analisador sintático, o GOLD, ao contrário, gera um arquivo contendo a tabela sintática. Esta tabela, por sua vez, é carregada em tempo de execução, por um módulo de *run-time*, permitindo assim que gramáticas diferentes possam ser processadas pelo mesmo programa.

Analisa-se a seguir a utilização do YACC/BISON e o ANTLR.

### 4.8.1 YACC / BISON

O YACC (*Yet Another Compiler-Compiler*) é um gerador automático de analisadores sintáticos (*parser-generator*), desenvolvido pela *Bell Laboratories*, em 1975. O YACC faz parte do pacote UNIX, assim como o LEX, ambos trabalhando de forma integrada.

Atualmente, o pacote FLEX/BISON é utilizado com vantagens em substituição ao LEX/YACC, dentre elas a disponibilização e a geração de código para diversas plataformas. O formato do arquivo de especificação para a gramática a ser analisada (denominado normalmente de programa YACC) é praticamente o mesmo, para ambos os casos, mantendo assim a compatibilidade. Caso uma extensão não seja especificada, a extensão padrão é '.y'.

Da mesma forma que o LEX, um programa YACC contém três seções, separadas pelo símbolo %:

- ❑ **Declarações:** Esta é a primeira seção a ser declarada. É nela que se encontram as declarações de variáveis, de constantes manifestas e todos os **tokens** a serem utilizados pelas regras de tradução. Os tokens deverão ser identificados pela palavra %token. Por exemplo, os tokens para os identificadores e para as palavras reservadas IF, THEN e ELSE poderiam ser:

```
%token IF THEN ELSE
```

```
%token ID
```

- **Regras de Tradução:** Esta é a segunda seção a ser declarada. Nela encontram-se especificadas as produções da gramática para a linguagem a ser reconhecida. Por exemplo, a produção:

```
<Expr> ::= <Expr> + <Termo> | <Termo>
```

deverá ser traduzida para:

```
expr : expr '+' termo | termo
```

Caso tenha sido definido o token PLUS para o operador de adição, então teríamos:

```
expr : expr PLUS termo | termo
```

Ações semânticas podem ser especificadas, desde que estejam envolvidas por chaves, conforme indicado a seguir:

```
expr : expr '+' termo { $$ = $1 + $3 } | termo
```

No exemplo podemos observar a utilização de \$\$, \$1 e \$3. Estes elementos correspondem aos atributos associados a produção. O \$\$ corresponde ao atributo associado à `expr`, o lado esquerdo da produção, o qual será retornado como resultado da avaliação desta produção. Já \$1 corresponde ao atributo do primeiro símbolo gramatical do lado direito da produção, neste caso `expr`, enquanto \$3 corresponde ao terceiro, o `termo`. Observe que o \$2 corresponde, portanto, ao atributo do símbolo '+'. Assim, se a expressão a ser analisada é 1 + 2, temos \$\$ = 3, \$1 = 1, \$2 = '+' e \$3 = 2.

- **Procedimentos Auxiliares:** Esta é a terceira e última seção do arquivo de especificação. Nela são colocadas as definições de procedimentos necessários para a realização das ações especificadas ou auxiliares ao analisador sintático.

O exemplo a seguir permite construir um analisador sintático para avaliar expressões aritméticas, simulando uma calculadora de mesa, a qual a cada ENTER pressionado avalia a expressão corrente. O arquivo de especificação para o YACC é indicado abaixo.

```
/* secao de declaracao */
%{
/* inclusao de arquivos de cabecalhos */
#include <cctype.h>
#include <stdio.h>
#include <math.h>
#define YYSTYPE double
int      yylex (void);
void    yyerror (char const *);
%}
```

```
/* declaracao dos tokens */
%token      NUM
%%
/* declaracao das regras gramaticais */
linha : /* vazio */
       | linha expr '\n' {printf("Resultado: %g\n", $2);}
       ;

expr   : expr '+' termo  {$$ = $1 + $3;}
       | expr '-' termo  {$$ = $1 - $3;}
       | termo
       ;
termo  : termo '*' fator {$$ = $1 * $3;}
       | termo '/' fator {$$ = $1 / $3;}
       | fator
       ;
fator  : NUM {$$ = $1;}
       | '(' expr ')' {$$ = $2;}
       ;
%%
/* procedimentos auxiliares */
/* Analisador Lexico */
int yylex()
{
    int c;
    /* Salta os espacos em branco */
    while ((c = getchar()) == ' ' || c == '\t') continue;

    /* Processa os numeros */
    if (c == '.' || isdigit (c))
    {
        ungetc (c, stdin);
        scanf ("%lf", &yyval);
        return NUM;
    }

    /* Retorna fim de arquivo (EOF) */
    if (c == EOF) return 0;

    /* Retorna um caracter simples */
    return c;
}

/* Tratamento de Erros. */
void yyerror (char const *s)
{
    fprintf (stderr, "%s\n", s);
}

/* Programa Principal */
void main()
{
    if (yyparse() != 0) { printf("Erro sintatico\n"); }
}
```

O exemplo anterior utilizou uma função `yylex()`, responsável pela análise léxica da entrada, construída no próprio arquivo de especificação. Entretanto, isto não é necessário. Na verdade, o LEX e o YACC foram projetados para trabalharem um com o outro. Portanto, apesar de podermos optar por implementar nosso próprio analisador léxico dentro do programa YACC, seria mais interessante fazê-lo com a ajuda do LEX.

O YACC espera, como analisador léxico a ser definido, a função `yylex()`, que tem o mesmo nome do analisador gerador pelo LEX. Desta forma, basta incluir ao arquivo de definição, na seção de funções, o arquivo ‘`lex.yy.c`’, o qual contém a implementação do analisador léxico gerado.

O arquivo de especificação abaixo permite a construção de uma calculadora de mesa um pouco mais avançada, a qual lida com variáveis. Detalhes adicionais sobre como construir o arquivo de especificação poderá ser obtido no manual do YACC/BISON.

```
/* seção de declaração */
{
    /* inclusao de arquivos de cabecalhos */
    /* inclusao de arquivos de cabecalhos */
    #include <ctype.h>
    #include <stdio.h>
    #include <math.h>
    #define YYSTYPE double
    int     yylex (void);
    void   yyerror (char const *);

    float   x[26];      // Tabela de variáveis
}

/* declaracao dos tokens */

%token <Real>      NUM          /* constantes numerica */
%token <Integer>    ID           /* variaveis */
%type  <Real>       expr         /* expressoes */
%left  '+' '-'        /* operadores */
%left  '*' '/' 
%right UMINUS

%token ILLEGAL        /* token ilegal */

%%
/* secao de regras de traducao */
input : /* derivação vazia - não faz nada */
        | input '\n'           { yyaccept; }
        | input expr '\n'       { printf("Resultado: %f", $2); }
        | input ID '=' expr '\n' { x[$2] := $4;
                                    printf("%c = %f", $2, $4); }
        | error '\n'           { yyerrok; }
```

```

;
expr  : expr '+' expr      { $$ := $1 + $3; }
      | expr '-' expr      { $$ := $1 - $3; }
      | expr '*' expr      { $$ := $1 * $3; }
      | expr '/' expr      { $$ := $1 / $3; }
      | '(' expr ')'
      | '-' expr
      %prec UMINUS
      | NUM                 { $$ := $1; }
      | ID                  { $$ := x[$1]; }
;

%%
/* procedimentos auxiliares */
/* inclusão do analisador léxico gerado pelo lexer */
#include yylex.y.c;

void main()
{
    for(int i = 0; i < 26; i++) { x[i] = 0; }
    if (yyparse != 0) { printf("Erro sintático\n"); }
}

```

## 4.8.2 ANTLR

O ANTLR<sup>4</sup> (ANother Tool for Language Recognition) é um gerador de analisadores sintáticos descendentes (*parser generator*) para gramáticas ALL(\*) – Adaptive LL(\*)<sup>5</sup>, desenvolvido em Java por Terence Parr, da University of San Francisco, sendo utilizado por empresas como a Google, Twiter, Mozilla Research e Oracle (PARR, 2012). Atualmente o ANTLR se encontra na versão 4, o qual permite gerar código para as linguagens Java (padrão), C#, Phyton, Go, C++, Swift e PHP. Na atual versão o ANTLR consegue lidar com gramáticas recursivas à esquerda (somente recursão direta) e com não determinismos, não sendo necessário fatorar à esquerda as produções.

Para poder executar o ANTLR você deverá baixar o `antlr-4.x-complete.jar` (onde `x` indica a release disponível), colocando-o em um diretório apropriado, e configurar adequadamente as variáveis ambientais `PATH` e `CLASSPATH`.

Considerando que a versão 4.8 do ANTLR tenha sido baixado para o diretório “`C:\ANTLR4`” e que o Java Development Kit (JDK) tenha sido instalado no diretório “`C:\Java`”, configure as variáveis ambientais como se segue:

<sup>4</sup> [www.antlr.org](http://www.antlr.org)

<sup>5</sup> ALL(\*) é uma extensão para LL(\*), sendo um método que permite analisar a gramática dinamicamente durante a execução, ao invés de estaticamente como é feito pelo YACC/Bison. Como os analisadores ALL (\*) têm acesso às sequências de entrada reais, eles podem sempre descobrir como reconhecê-las de modo mais adequado à gramática.

```
PATH=%PATH%;C:\Java\bin;C:\ANTLR4  
CLASSPATH=%CLASSPATH%;.;C:\ANTLR4\antlr-4.8-complete.jar
```

A seguir crie no diretório do ANTLR os seguintes arquivos de lote:

- `antlr4.bat`

```
java -cp C:\ANTLR4\antlr-4.8-complete.jar;%CLASSPATH% org.antlr.v4.Tool %*
```

- `grun.bat`

```
java org.antlr.v4.gui.TestRig %*
```

Uma vez criados os arquivos de lote, vejamos um exemplo de uma gramática a ser utilizada pelo ANTLR, a qual permite a avaliação de expressões aritméticas envolvendo os operadores de adição, subtração, multiplicação e divisão.

```
/** Expr.g4 */  
grammar Expr;  
  
/** Regras gramaticais */  
expr : expr ('+'|'-') term | term ;  
term : term ('*'|'/') fact | fact ;  
fact : INT | ID | '(' expr ')' ;  
  
/** Itens léxicos */  
ID   : [a-zA-Z] ([a-zA-Z] | [0-9])* ; // identificadores  
INT  : [0-9]+ ; // números inteiros  
WS   : [ \t\r\n]+ -> skip ; // despreza brancos
```

Crie o diretório “C:\ANTLR4\Exemplos\Expr” e salve a gramática nele como “Expr.g4”. No arquivo podemos verificar regras gramaticais (expr, term e fact), as quais devem estar em letras minúsculas, e expressões regulares para o reconhecimento de itens léxicos pelo analisador léxico (ID, INT e WS), as quais devem ser especificadas em letras maiúsculas. A seguir, abra o prompt de comando, vá para a pasta em questão e digite “antlr4 Expr.g4”. Como resultado da execução, o ANTLR irá criar os seguintes arquivos:

```
Expr.interp  
Expr.tokens  
ExprLexer.interp  
ExprLexer.tokens  
ExprLexer.java
```

```
ExprListener.java  
ExprBaseListener.java  
ExprParser.java
```

Os arquivos “`ExpLexer.java`” e “`ExpParser.java`” correspondem ao analisador léxico e sintático, respectivamente. Os arquivos “`ExprListner.java`” e “`ExprBaseListner.java`” contêm as interfaces para se caminhar na árvore sintática gerada durante a execução e serão tratados posteriormente. Já os demais arquivos são auxiliares.

Para compilar os arquivos digite “`javac *.java`”.

Para testar o funcionamento do analisador sintático utilizaremos o “`org.antlr.v4.gui.TestRig`”, chamado pelo “`grun.bat`”. Este programa permite verificarmos o fluxo de tokens contidos na entrada e visualizar a árvore sintática correspondente a uma expressão.

Para verificarmos o fluxo de tokens, basta que digitemos:

```
grun Expr expr -tokens  
45+32*67  
^Z
```

onde `Expr` é o arquivo contendo o analisador sintático; `expr` é a regra gramatical principal (ou seja, o não terminal de partida da gramática); `-tokens` é uma diretiva para análise do fluxo de tokens; `45+32*67` é a sentença de entrada; `^Z` (CONTROL-Z) é o marcador de fim de arquivo.

O resultado da execução é:

```
[@0,0:1='45',<INT>,1:0]  
[@1,2:2='+',<+'>,1:2]  
[@2,3:4='32',<INT>,1:3]  
[@3,5:5='*',<'*>,1:5]  
[@4,6:7='67',<INT>,1:6]  
[@5,10:9='<EOF>',<EOF>,2:0]
```

Cada linha apresentada corresponde a um token da entrada. As informações contidas em cada token encontram-se separadas por vírgula. Por exemplo, na terceira linha, “`@2`” indica que se trata do terceiro token (iniciando-se a contagem em 0); “`3:4='32'`” indica que o texto vai da posição 3 até a 4, sendo ele igual (‘=’) a “`32`”; o `<INT>` indica que se trata de um `INT`; e o “`1:3`” indica que o item está na linha 1, começando na posição 3 (iniciando-se a contagem em 0).

Para ver a árvore gramatical gerada, de acordo com a representação adotada pelo LISP, digite:

```
grun Expr expr -tree
1+2*3
^Z
```

O resultado da execução é:

```
(expr (expr (term (fact 1))) + (term (term (fact 2)) * (fact 3)))
```

Entretanto, a forma mais fácil de visualizar a árvore gramatical gerada é digitando:

```
grun Expr expr -gui
1+2*3
^Z
```

O resultado da execução do grun, utilizando a diretiva `-gui`, se encontra ilustrado na Figura 30.

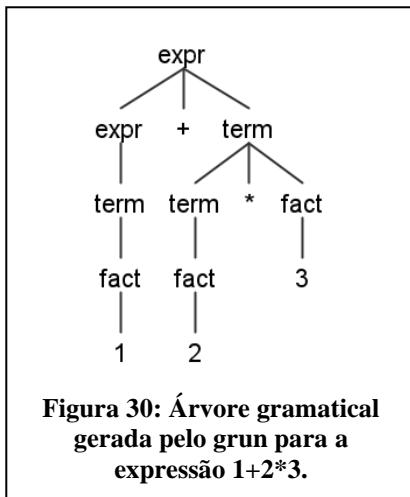


Figura 30: Árvore gramatical gerada pelo grun para a expressão  $1+2*3$ .

Agora que você já viu como testar o ANTLR, vejamos como um *parser* pode ser gerado.

Existem duas formas de gerar um *parser* com o ANTLR. A primeira delas é através da utilização de *Parse-Tree Listeners* e *Visitors*, enquanto que a segunda é através da especificação de ações na própria gramática. A vantagem do primeiro método em relação ao segundo é que podemos deixar a gramática livre, permitindo que a reutilizemos para outros fins, o que não é possível com a abordagem do segundo método.

Durante o processo de análise, o *parser* gerado pelo ANTLR cria automaticamente uma árvore sintática (*parser-tree*). Podemos interagir com a árvore sintática gerada através de *listeners*<sup>6</sup> criados, bastando para isto conhecer as estruturas de dados e o nome das classes utilizadas para o seu processamento.

---

<sup>6</sup> Listeners são métodos chamados automaticamente quando algum evento ocorre, como por exemplo o click de um botão.

Vamos começar examinando as estruturas de dados e os nomes de classe que o ANTLR usa para reconhecimento e análise da *parse-tree*, conforme definido por Terence Parr (2012). Para isto, considere a seguinte gramática:

```
/** Stat.g4 */
grammar Stat;

/** Regras gramaticais */
stat      : assign          // Sentença de Atribuição
           | ifstat         // Sentenças if
           | whilestat      // Senteças while
           ;
assign    : ID '=' expr ';' 
           ;
ifstat   : ... ;
whilestat: ... ;
expr     : ID
           | INT
           ;
/** Itens léxicos */
ID       : [a-zA-Z] ([a-zA-Z] | [0-9])* ; // identificadores
INT      : [0-9]+ ;                      // números inteiros
WS       : [\t\r\n]+ -> skip ;          // despreza brancos
```

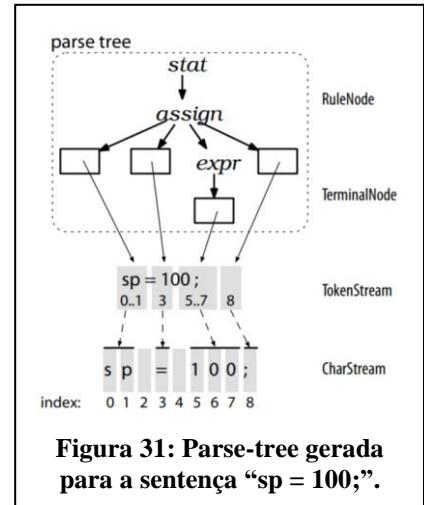


Figura 31: Parse-tree gerada para a sentença “sp = 100;”.

Durante a compilação da gramática pelo ANTLR, as seguintes classes serão geradas: CharStream, Lexer, Token, Parser e ParseTree. A classe CharStream corresponde a sequência de caracteres a ser lido pelo analisador léxico (classe Lexer), gerando uma lista de tokens, o TokenStream, o qual será o elo de comunicação entre o analisador léxico e o sintático (classe Parser), de onde os tokens serão retirados (classe Token). Ao chamarmos o parser para analisar a expressão “sp = 100;” a *parse-tree* da Figura 31 será gerada. Nela podemos verificar duas outras estruturas de dados, as classes RuleNode e TerminalNode, as quais correspondem as raízes e as folhas das subárvore, respectivamente.

Para dar um melhor suporte ao acesso dos elementos dentro dos nós, o ANTLR gera subclasses de RuleNode para cada regra. A Figura 32 apresenta as classes StatContext, AssignContex e ExprContext, correspondentes às raízes das subárvore geradas para a sentença de atribuição “sp = 100;”. Estas classes são chamadas de *objetos de contexto* devido a elas armazenarem tudo que é necessário para o reconhecimento da sentença. Por exemplo, AssignContex provê métodos

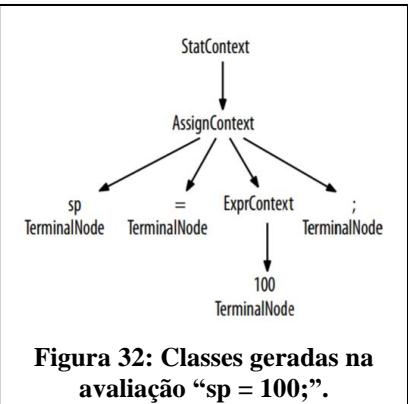


Figura 32: Classes geradas na avaliação “sp = 100;”.

ID() e expr() para acessar o identificador e o nó correspondente a subárvore da expressão. Vejamos um exemplo. Abra o arquivo “StatParser.java” e procure pelas definições abaixo:

```
public static class StatContext extends ParserRuleContext {...}
public static class AssignContext extends ParserRuleContext {
    public TerminalNode ID() { ... }
    public ExprContext expr() { ... }
}
public static class IfstatContext extends ParserRuleContext { ... }
public static class WhilestatContext extends ParserRuleContext { ... }
public static class ExprContext extends ParserRuleContext { ... }
```

Como vimos, o ANTLR gera uma *parse-tree* correspondente a análise realizada. Para podemos percorrer esta árvore o ANTLR gera *listeners* correspondentes a cada regra gramatical. São gerados dois tipos de *listeners* correspondentes a uma produção: um para entrada de um nó (*enter*) e outro para a saída do nó (*exit*). O que devemos fazer é sobrescrever os métodos correspondentes e executarmos as ações desejadas. Por exemplo, considerando a gramática anterior, teríamos os seguintes métodos para a produção `assign`:

```
@Override public void enterAssign(StatParser.AssignContext ctx) { }
@Override public void exitAssign(StatParser.AssignContext ctx) { }
```

Estas especificações encontram-se no arquivo “StatBaseListener.java”, as quais devemos sobreescrivê-las. Entretanto, a interface encontra-se definida no arquivo “StatListener.java”.

Vejamos agora um exemplo de aplicação. Suponha a seguinte gramática, nomeada por “ExprLabel.g4”:

```
/** ExprLabel.g4 */
grammar ExprLabel;

/** Regras gramaticais */
expr  : expr ADD term  # LabelAdd
      | expr SUB term  # LabelSub
      | term          # Labelterm
      ;
term  : term MUL fact  # LabelMul
      | term DIV fact  # LabelDiv
      | fact           # Labelfact
      ;
fact   : INT             # LabelInt
```

```

| ID           # LabelId
| '(' expr ')' # Labelexpr
;

/** Itens léxicos      */
ADD  : '+' ;          // Operador de adição
SUB  : '-' ;          // Operador de subtração
MUL  : '*' ;          // Operador de multiplicação
DIV  : '/' ;          // Operador de divisão
ID   : [a-zA-Z] ([a-zA-Z] | [0-9])* ; // identificadores
INT  : [0-9]+ ;        // números inteiros
WS   : [ \t\r\n]+ -> skip ;          // despreza brancos

```

Na gramática acima encontramos rótulos para cada produção a ser aplicada, os quais estão no final da produção, identificado pelo marcador `#` seguido de um nome, como em “`# LabelAdd`”. Isto é importante, pois permite isolar cada produção, o que força o ANTLR a gerar um método para entrada e outro para saída para cada rótulo, como pode ser visto no exemplo a seguir, o qual apresenta os métodos “`enterLabelAdd`” e “`exitLabelAdd`”, gerados para o rótulo “`LabelAdd`”:

```

@Override public void enterLabelAdd(ExprLabelParser.LabelAddContext ctx) {}
@Override public void exitLabelAdd(ExprLabelParser.LabelAddContext ctx) {}

```

Nosso objetivo agora é gerar instruções para avaliar uma expressão em uma máquina baseada em pilha. As instruções geradas são ADD (somar dois valores contidos no topo da pilha e empilhar o resultado), SUB (subtrair dois valores contidos no topo da pilha e empilhar o resultado), MUL (multiplicar dois valores contidos no topo da pilha e empilhar o resultado), DIV (dividir dois valores contidos no topo da pilha e empilhar o resultado), PUSH valor (empilhar um valor inteiro), PUSH (end) (empilhar o conteúdo apontado pelo endereço end). Para realizar esta tarefa, basta que sobrescrevamos os métodos `exit`, contidos no arquivo “`ExprLabelBaseListener.java`”. Isto é feito no arquivo “`ExpLabelAssembly.java`”, conforme o exemplo a seguir:

```

/**
 * Gera código assembly para uma máquina baseada em pilha
 */
public class ExprLabelAssembly extends ExprLabelBaseListener {
    /** Saída para o operação de Adição */
    @Override
    public void exitLabelAdd(ExprLabelParser.LabelAddContext ctx) {
        System.out.println("ADD");
    }

    /** Saída para o operação de Subtração */
    @Override
    public void exitLabelSub(ExprLabelParser.LabelSubContext ctx) {

```

```
        System.out.println("SUB");
    }

    /** Saída para o operação de Multiplicação */
    @Override
    public void exitLabelMul(ExprLabelParser.LabelMulContext ctx) {
        System.out.println("MUL");
    }

    /** Saída para o operação de Divisão */
    @Override
    public void exitLabelDiv(ExprLabelParser.LabelDivContext ctx) {
        System.out.println("DIV");
    }

    /** Saída para o operação de reconhecimento de um inteiro */
    @Override
    public void exitLabelInt(ExprLabelParser.LabelIntContext ctx) {
        System.out.println("PUSH " + ctx.INT().getText());
    }

    /** Saída para o operação de reconhecimento de um identificador */
    @Override
    public void exitLabelId(ExprLabelParser.LabelIdContext ctx) {
        System.out.println("PUSH (" + ctx.ID().getText() + ")");
    }
}
```

Note pelo exemplo as chamadas dos métodos “ctx.INT()” e “ctx.ID()”, contidos nos métodos “exitLabelInt()” e “exitLabelId()”. Note que ambos acessam o contexto “ctx” para descobrir qual é o valor correspondente ao inteiro e o nome correspondente ao identificador, respectivamente.

Para executar o programa precisamos criar o método “main()”. Isto é feito no arquivo “TestExprLabel.java”, o qual encontra-se transcrito abaixo:

```
/*
 * Exemplo de integração com o ANTLR
 */
// import das bibliotecas de runtime do ANTLR
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

public class TestExprLabel {
    public static void main(String[] args) throws Exception {
        // Cria um CharStream que lê a partir da entrada padrão
        ANTLRInputStream input = new ANTLRInputStream(System.in);

        // Cria um lexer para processar a entrada
        ExprLabelLexer lexer = new ExprLabelLexer(input);

        // Cria um buffer de tokens gerados pelo lexer
        CommonTokenStream tokens = new CommonTokenStream(lexer);
```

```
// Cria um parser para processar os tokens
ExprLabelParser parser = new ExprLabelParser(tokens);

// Chame a regra principal, expr, para a qual gerará
// uma árvore sintática
ParseTree tree = parser.expr();

// Cria um parse tree walker para gerenciar as chamadas
// dos callbacks
ParseTreeWalker walker = new ParseTreeWalker();

// Caminha na árvore criada durante a análise sintática e
// chama os callbacks
walker.walk(new ExprLabelAssembly(), tree);

System.out.println();
}
```

No exemplo podemos verificar a criação do léxico (lexer), do *stream* de *tokens* (CommonTokenStream), o *parser* (parser), a *parse-tree* (tree) e uma classe para podermos caminhar na árvore gerada (walker). Note que para avaliarmos a árvore gerada, basta que passemos o arquivo contendo os métodos sobrescritos, contidos em “ExprLabelAssembly”, e a árvore a ser avaliada, neste caso tree.

Para executar o arquivo, primeiro rode o ANTLR, com “antlr4 ExprLabel.g4”, compile os arquivos gerados com “javac \*.java” e rode o programa com “java TestExprLabel”. Vejamos um exemplo, considerando a expressão “1+2\*a”:

```
antlr4 ExprLabel.g4
javac *.java
java TestExprLabel
1+2*a
^z
PUSH 1
PUSH 2
PUSH (a)
MUL
ADD
```

Como vimos, os exemplos anteriores mostraram como utilizar o ANTLR com *listeners*. Vejamos agora como utilizar *visitors*.

O ANTLR oferece uma forma de você gerenciar a *parse-tree*, gerada automaticamente, de modo manual. Isto é, você é quem deverá efetuar as chamadas para visitar os nós da árvore quando necessário. Vejamos um exemplo de

como realizar esta tarefa analisando uma calculadora, conforme definida por Terence Parr (2012). Crie o arquivo “LabeledExpr.g4” definido a seguir:

```
/** LabeledExpr.g4 */
grammar LabeledExpr ;

prog   :   stat+ ;

stat   :   expr NEWLINE          # printExpr
        |   ID '=' expr NEWLINE    # assign
        |   NEWLINE                 # blank
        ;
expr   :   expr op=('*'|'/') expr  # MulDiv
        |   expr op=('+|-') expr   # AddSub
        |   INT                      # int
        |   ID                       # id
        |   '(' expr ')'            # parens
        ;
MUL    :   '*' ;                // Operador de multiplicação
DIV    :   '/' ;                // Operador de divisão
ADD    :   '+' ;                // Operador de divisão
SUB    :   '-' ;                // Operador de subtração
ID     :   [a-zA-Z]+ ;           // Identificador
INT    :   [0-9]+ ;              // Números inteiros
NEWLINE: '\r'? '\n' ;           // Newline (no Windos CR+LF)
WS     :   [ \t]+ -> skip ;    // Consome brancos
```

Observe a diferença desta gramática com as anteriores. Note que nela foi colocado um “op=(‘+’ | ‘-’)” e “op=(‘\*’ | ‘/’)”, apesar de termos definidos também os itens léxicos MUL, DIV, ADD e SUB. O motivo do “op” é para podermos consultar qual foi o operador utilizado. Poderíamos ter declarado “op=(ADD|SUB)”, que o resultado seria o mesmo. Entretanto, é importante a declaração dos operadores como itens léxicos, pois o ANTLR irá gerar constantes que poderão ser referenciadas pelo nome, o que não ocorre se declararmos apenas o caractere “+” ou “-”.

Outro detalhe a observar é que a produção inicial “prog” permite que definamos um ou mais “stat” (observe o uso do “+”), sendo cada “stat” separado do outro por um “NEWLINE” (um ENTER). Assim, as seguintes entradas para “prog” são válidas:

```
193
a = 5
b = 6
a+b*2
(1+2)*3
```

Salve estas entradas no arquivo “t.expr” e execute os comandos a seguir:

```
antlr4 LabeledExpr.g4
javac *.java
grun LabeledExpr prog t.expr -gui
```

Como resultado teremos a árvore da Figura 33. Note na figura as diferentes “stat” e os marcadores de finalização das linhas.

Agora que já vimos que a gramática está correta, vamos codificar a nossa calculadora. Primeiramente, vejamos como ficaria o programa principal necessário para manipular o arquivo “t.expr”:

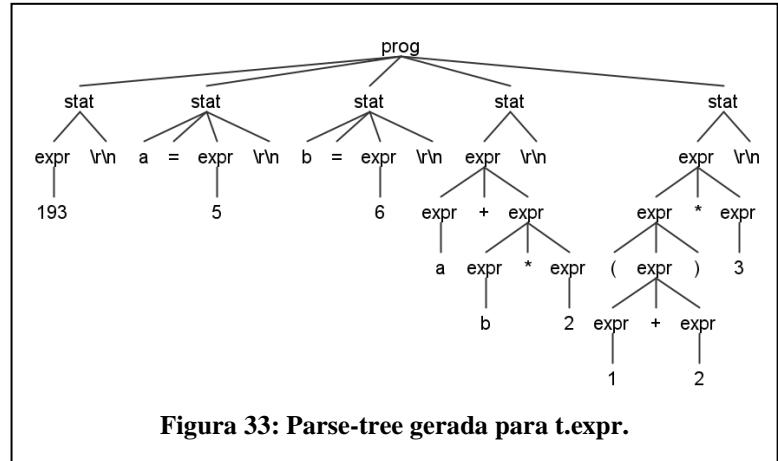


Figura 33: Parse-tree gerada para t.expr.

```
/*
 * Calc.java
 */
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.ParseTree;

import java.io.FileInputStream;
import java.io.InputStream;

public class Calc {
    public static void main(String[] args) throws Exception {
        String inputFile = null;
        if ( args.length > 0 ) inputFile = args[0];
        InputStream is = System.in;
        if ( inputFile != null ) is = new FileInputStream(inputFile);
        ANTLRInputStream input = new ANTLRInputStream(is);
        LabeledExprLexer lexer = new LabeledExprLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        LabeledExprParser parser = new LabeledExprParser(tokens);
        ParseTree tree = parser.prog(); // parse

        EvalVisitor eval = new EvalVisitor();
        eval.visit(tree);
    }
}
```

Note que agora não estamos mais utilizando um *walker* e sim um *visitor*, ou seja, uma instância da classe `EvalVisitor`, que possui o método `visit()`, o qual permitirá que caminhemos pela *parse-tree* gerada. Esta classe será uma subclasse da classe `LabeledExprBaseVisitor`, a qual é criada automaticamente pelo ANTLR utilizando a diretiva “`-visitor`”, assim como outras. Para isto devemos chamar o ANTLR da seguinte forma:

```
antlr4 -no-listener -visitor LabeledExpr.g4
```

Como resultado serão gerados os seguintes arquivos:

```
LabeledExpr.g4
LabeledExpr.interp
LabeledExpr.tokens
LabeledExprLexer.interp
LabeledExprLexer.tokens
LabeledExprLexer.java
LabeledExprParser.java
LabeledExprVisitor.java
LabeledExprBaseVisitor.java
```

Olhando o arquivo “`LabeledExprBaseVisitor.java`” podemos observar que foi criada a classe genérica “`LabeledExprBaseVisitor<T>`” e métodos para acessar cada nó da *parse-tree*, os quais estão identificadas pelos respectivos rótulos das produções da gramática:

```
public class LabeledExprBaseVisitor<T>
    extends AbstractParseTreeVisitor<T>
    implements LabeledExprVisitor<T> {

    @Override public T visitProg(LabeledExprParser.ProgContext ctx)
    { return visitChildren(ctx); }

    @Override public T visitPrintExpr(LabeledExprParser.PrintExprContext ctx)
    { return visitChildren(ctx); }

    @Override public T visitAssign(LabeledExprParser.AssignContext ctx)
    { return visitChildren(ctx); }

    @Override public T visitBlank(LabeledExprParser.BankContext ctx)
    { return visitChildren(ctx); }

    @Override public T visitParens(LabeledExprParser.ParensContext ctx)
    { return visitChildren(ctx); }

    @Override public T visitMulDiv(LabeledExprParser.MulDivContext ctx)
    { return visitChildren(ctx); }

    @Override public T visitAddSub(LabeledExprParser.AddSubContext ctx)
    { return visitChildren(ctx); }

    @Override public T visitId(LabeledExprParser.IdContext ctx)
```

```
    { return visitChildren(ctx); }

    @Override public T visitInt(LabeledExprParser.IntContext ctx)
    { return visitChildren(ctx); }
}
```

Observe que cada *visitor* visita seus filhos, tendo como retorno o valor de retorno daquele filho, ou seja, “return visitChildren(ctx)”. O que devemos fazer para modificarmos o comportamento padrão dos métodos é sobrescrevê-los para adicionarmos as instruções necessárias a nossos propósitos.

Vejamos, desejamos efetuar cálculos assim que passarmos pelos nós da árvore contendo as operações de adição, subtração, multiplicação e divisão. Mas também desejamos atribuir o resultado da avaliação de uma expressão para um identificador, conforme pode ser visto em “ID '=' expr NEWLINE”. Como estamos utilizando somente valores inteiros, então o resultado da avaliação de uma expressão será um valor inteiro. Portanto, o que devemos fazer é criar classe “EvalVisitor” como uma subclasse de “LabeledExprBaseVisitor”, sendo o seu tipo de retorno um “Integer”, e sobreescrver os métodos necessários a tarefa desejada. Vejamos como tudo isto ficaria:

```
/*
 * EvalVisitor.java
 */
import java.util.HashMap;
import java.util.Map;

public class EvalVisitor extends LabeledExprBaseVisitor<Integer> {
    /** Memória para armazenarmos os pares (variável, valor) */
    Map<String, Integer> memory = new HashMap<String, Integer>();

    /** ID '=' expr NEWLINE */
    @Override
    public Integer visitAssign(LabeledExprParser.AssignContext ctx) {
        String id = ctx.ID().getText(); // Id a esquerda do '='
        int value = visit(ctx.expr()); // Calcula o valor da expressão
        memory.put(id, value); // Armazena na memória
        return value; // Retorna o valor calculado
    }

    /** expr NEWLINE */
    @Override
    public Integer visitPrintExpr(LabeledExprParser.PrintExprContext ctx) {
        Integer value = visit(ctx.expr()); // Calcula o valor da expressão
        System.out.println(value); // Imprime o resultado
        return 0; // Nenhum valor a retornar
    }

    /** INT */
    @Override
    public Integer visitInt(LabeledExprParser.IntContext ctx) {
        return Integer.valueOf(ctx.INT().getText()); // Retorna o valor
    }
}
```

```

}

/** ID */
@Override
public Integer visitId(LabeledExprParser.IdContext ctx) {
    String id = ctx.ID().getText(); // Qual é o identificador
    if ( memory.containsKey(id) ) // Se ele já está na memória
        return memory.get(id); // Devolve o seu valor
    return 0; // Senão o valor é zero
}

/** expr op=('*'|'/') expr */
@Override
public Integer visitMulDiv(LabeledExprParser.MulDivContext ctx) {
    int left = visit(ctx.expr(0)); // Avalia a expressão a esquerda
    int right = visit(ctx.expr(1)); // Avalia a expressão a direita
    if ( ctx.op.getType() == LabeledExprParser.MUL ) // Qual operação?
        return left * right; // Multiplicação
    return left / right; // Divisão
}

/** expr op=('+'|'-') expr */
@Override
public Integer visitAddSub(LabeledExprParser.AddSubContext ctx) {
    int left = visit(ctx.expr(0)); // Avalia a expressão a esquerda
    int right = visit(ctx.expr(1)); // Avalia a expressão a esquerda
    if ( ctx.op.getType() == LabeledExprParser.ADD ) // Qual operação?
        return left + right; // Adição
    return left - right; // Subtração
}

/** '(' expr ')' */
@Override
public Integer visitParens(LabeledExprParser.ParensContext ctx) {
    return visit(ctx.expr()); // Avalia a expressão e retorna o
                            // resultado
}

```

Note pelo exemplo acima que estamos utilizando o tipo genérico `Map` como tabela de símbolos, criando a variável `memory`, onde armazenaremos o identificador com o valor de atribuição associado a ele. Veja também que o que é feito pelos métodos `visitMulDiv()` e `visitAddSub()` é efetuar a operação associada ao operador `op`, com os valores retornados pela expressão contida no lado esquerdo e direito do operador. Além disso, se o valor da expressão é um `INT`, então apenas devolvemos o seu valor correspondente, conforme pode ser visto em `visitInt()`; e se for um `ID` acessamos `memory` e capturamos o valor associado a ele, conforme indicado por `visitId()`.

Para ver o programa em execução, execute o ANTLR, compile os arquivos gerados e execute o programa “Calc” passando “t.expr” como entrada:

```
antlr4 -no-listener -visitor LabeledExpr.g4
javac *.java
java Calc t.expr
193
17
9
```

Uma das coisas que você pode estar se perguntando agora é: “Mas a gramática é recursiva à esquerda e ambígua. Como o ANTLR consegue lidar com isto?”. Bom! Para fazer isto precisamos entender os mecanismos de precedência e associatividade, o qual veremos mais a frente. Por enquanto, desconsideramos este fato e o ANTLR resolveu a ambiguidade seguindo a ordem da declaração das produções. Se tivéssemos mudado a ordem das produções de multiplicação e divisão com as de adição e subtração em `expr`, conforme o exemplo abaixo, teríamos problemas com o resultado da avaliação da expressão:

```
expr      : expr op=('*'|'/') expr # MulDiv
          | expr op=('+'|'-') expr # AddSub
          ...
;
```

Conforme comentado anteriormente, os métodos descritos até o momento deixam a gramática livre para reutilização. Esta abordagem permite a reutilização da gramática para, por exemplo, ser utilizada para a geração de código para diversas linguagens a partir de um mesmo arquivo fonte. Entretanto, todos eles necessitam que a *parse-tree* seja construída para somente depois podermos executar alguma ação. Mas, em alguns casos desejamos executar alguma ação durante a etapa de análise sintática e não somente após ela ter sido concluída. Pensando nisso o ANTLR permite que você especifique ações diretamente na gramática, as quais serão executadas durante o processo de análise e não após ele ter sido concluído. Além do mais, a inclusão de ações diretamente na gramática pode parecer uma tarefa mais simples. Vejamos como isto funciona.

A inclusão de ações em uma gramática ANTLR é feita colocando-a entre { e }. Por exemplo, a gramática a seguir permite gerar código para avaliação de uma expressão pela máquina de pilha visto anteriormente.

```
/** AcoesExpr.g4 */
grammar AcoesExpr ;

expr : expr op=('*'|'/') expr { System.out.println($op.type == MUL ? "MUL" : "DIV"); }
      | expr op=('+'|'-') expr { System.out.println($op.type == ADD ? "ADD" : "SUB"); }
      | INT
      | ID
      | '(' expr ')' ;
```

```

;
MUL : '*' ;           // Operador de multiplicação
DIV : '/' ;           // Operador de divisão
ADD : '+' ;           // Operador de divisão
SUB : '-' ;           // Operador de subtração
ID  : [a-zA-Z]+ ;     // Identificador
INT : [0-9]+ ;         // Números inteiros
WS  : [ \t\r\n]+ -> skip ; // Consome brancos

```

Note pelo exemplo que podemos acessar um elemento da gramática usando o prefixo \$, como em “\$op.type”. O atributo “text”, em “\$INT.text” e “\$ID.text” é similar ao `getText()` do java, podendo-se utilizá-lo também.

O programa java “VM.java” a seguir é necessário para ativar a análise da expressão:

```

/*
 * VM.java
 */
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.ParseTree;

import java.io.FileInputStream;
import java.io.InputStream;

public class VM {
    public static void main(String[] args) throws Exception {
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        AcoesExprLexer lexer = new AcoesExprLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        AcoesExprParser parser = new AcoesExprParser(tokens);
        ParseTree tree = parser.expr();
    }
}

```

Execute o ANTLR, compile todos arquivos gerados e execute o “VM.class”:

```

antlr4 -no-listener AcoesExpr.g4
javac *.java
java VM
1+2*a
^Z

```

O resultado será igual ao obtido utilizando *listeners*:

```

PUSH 1
PUSH 2

```

```
PUSH (a)
MUL
ADD
```

Terence Parr (2012), apresenta um exemplo ilustrando uma calculadora, onde ao se digitar uma expressão e pressionar a tecla ENTER o resultado da avaliação é mostrado na tela. Vejamos primeiramente a gramática:

```
/** EvalExpr.g4 */
grammar EvalExpr;

/** Inclusão dos imports no Parser */
@header {
    import java.util.*;
}

/** Inclusão de campos na classe Parser */
@parser::members {
    /** Memória para a calculadora, armazena o par (id, valor) */
    Map<String, Integer> memory = new HashMap<String, Integer>();

    /** Método para avaliação da expressão */
    int eval(int left, int op, int right) {
        switch ( op ) {
            case MUL : return left * right;
            case DIV : return left / right;
            case ADD : return left + right;
            case SUB : return left - right;
        }
        return 0;
    }
}

stat : expr NL          {System.out.println($expr.val);}
     | ID '=' expr NL   {memory.put($ID.text, $expr.val);}
     | NL
     ;

expr returns [int val]
: left=expr op=('*'|'/') right=expr  {$val = eval($left.val, $op.type, $right.val);}
| left=expr op=('+'|'-') right=expr  {$val = eval($left.val, $op.type, $right.val);}
| INT
| ID
{
    String id = $ID.text;
    $val = memory.containsKey(id) ? memory.get(id) : 0;
}
| '(' expr ')'
;

MUL : '*' ;           // Operador de multiplicação
DIV : '/' ;           // Operador de divisão
ADD : '+' ;           // Operador de divisão
SUB : '-' ;           // Operador de subtração
ID  : [a-zA-Z]+ ;     // Identificador
INT : [0-9]+ ;         // Números inteiros
NL  : '\r'? '\n' ;     // Marcador de fim de linha
WS  : [ \t]+ -> skip ; // Consome brancos
```

Analisemos esta gramática em detalhes. Primeiramente vejamos algumas diretivas do ANTLR:

- ❑ `@header`: esta diretiva solicita ao ANTLR que coloque no cabeçalho dos arquivos gerados tudo o que estiver entre os delimitadores `{ e }`.
- ❑ `@lexer`: solicita que informações sejam incluídas no analisador léxico, ou seja, no `lexer`, que se encontra no arquivo de sufixo “`Lexer.java`”.
- ❑ `@parser`: solicita que informações sejam incluídas no analisador sintático, ou seja, no `parser`, que se encontra no arquivo de sufixo “`Parser.java`”.
- ❑ `members`: solicita que as informações contidas entre os delimitadores `{ e }` sejam incluídas como campos membros da classe `Parser` (`@parser::members`) ou da classe `Lexer` (`@lexer::members`).
- ❑ `returns`: lista de valores a serem retornados da regra de produção; cada retorno deverá estar associado a uma variável, como “`val`” na gramática.

Olhando a gramática também vemos a presença do prefixo `$` antes do nome dos rótulos de regras, operadores (`$left`, `$right` e `$op`) e variável de retorno (`$val`). Isto indica ao ANTLR que desejamos acessar ou modificar o conteúdo dos atributos destes rótulos. Por exemplo, em `$left.val` estamos acessando o conteúdo da variável de retorno `val` da expressão a esquerda do operador `op`. Já em `$INT.int` estamos acessando o valor inteiro associado ao token `INT`, enquanto `$ID.text` acessa o texto do token `ID`. Estes elementos, presentes na produção `expr`, são gerados pelo ANTLR como elementos do contexto de `expr` e são encontrados na classe `ExprContext`, que é derivada a partir da classe `ParserRuleContext`, podendo ser parcialmente visualizada a seguir:

```
public static class ExprContext extends ParserRuleContext {  
    public int val;  
    public ExprContext left;  
    public Token INT;  
    public Token ID;  
    public ExprContext expr;  
    public Token op;  
    public ExprContext right;  
    ...  
}
```

O método `eval()` é utilizado para o cálculo do resultado da operação associada a um operador. Assim, em “`{$val = eval($left.val, $op.type, $right.val); }`” estamos calculando o valor correspondente ao operador, o qual é dado por `$op.type`, e retornando o resultado em `$val`, que é a variável de retorno associada a produção `expr`.

É interessante também analisarmos o token `NL`. No Linux, todo final de linha em um arquivo texto é marcado pelo caractere ASCII 10 (o Line Feed, ou LF), o que é

representado por “\n”. Já no Windows, o final de linha é marcado pela sequência Carriage Return (ou CR), que é o código ASCII 13, seguido pelo LF. Logo, o que a expressão regular “NL : '\r'? '\n'” faz é identificar se a sequência de caracteres sendo analisada possui zero ou um CR, para o caso de o texto estar no Windows, e um LF. Neste caso, esta sequência é considerada como um avanço de linha, ou seja, NL.

Para podermos fazer que o nosso *parser* funcione, precisamos construir o programa de ativação. Ele é dado a seguir:

```
1. /**
2.  * Calc.java
3. */
4. import org.antlr.v4.runtime.ANTLRInputStream;
5. import org.antlr.v4.runtime.CommonTokenStream;
6.
7. import java.io.BufferedReader;
8. import java.io.FileInputStream;
9. import java.io.InputStream;
10. import java.io.InputStreamReader;
11.
12. public class Calc {
13.     public static void main(String[] args) throws Exception {
14.         String inputFile = null;
15.         if ( args.length>0 ) inputFile = args[0];
16.         InputStream is = System.in;
17.         if ( inputFile!=null ) {
18.             is = new FileInputStream(inputFile);
19.         }
20.
21.         BufferedReader br = new BufferedReader(new InputStreamReader(is));
22.         String expr = br.readLine(); // Lê a primeira linha
23.         int line = 1; // Linha inicial
24.
25.         EvalExprParser parser = new EvalExprParser(null); // Declara uma instância do parser
26.         parser.setBuildParseTree(false); // Não criará a parse-tree
27.
28.         while ( expr != null ) { // Enquanto tiver expressões
29.             // Cria um novo lexer e um token stream para cada linha (expressão)
30.             ANTLRInputStream input = new ANTLRInputStream(expr + "\n");
31.             EvalExprLexer lexer = new EvalExprLexer(input);
32.             lexer.setLine(line); // Notificar o lexer da posição da linha
33.             lexer.setCharPositionInLine(0); // Inicializa a posição inicial da linha
34.             CommonTokenStream tokens = new CommonTokenStream(lexer);
35.             parser.setInputStream(tokens); // Informa ao parser a token stream
36.             parser.stat(); // Ativa o parser
37.             expr = br.readLine(); // Pega a próxima linha, se existir
38.             line++;
39.         }
40.     }
41. }
```

Observe pelo programa, na linha 25, que na criação do *parser* não é mais informado a *token stream*, enquanto que na linha 26 é solicitado a ele para não gerar mais a *parse-tree*, pois ela é desnecessária. A linha 30 cria uma *input stream* composta apenas por uma linha da sentença de entrada, ao qual é passada para o *lexer*, o qual é criado na linha 31. As linhas 32 e 33 especificam o contador de linhas e a posição inicial do caractere da linha de entrada para o *lexer*. A linha 34 cria a *token stream*, a qual é informada ao parser na linha 35, sendo o mesmo ativado na linha

36, chamando a produção inicial `stat`. Já a linha 37 pega uma nova linha da entrada enquanto que a linha 38 incrementa o contador de linhas.

Para executar o programa siga os seguintes passos, onde o marcador > indica apenas a entrada a ser informada:

```
antlr4 -no-listener EvalExpr.g4
javac *.java
java Calc
> x = 1
> x
1
> x+2*3
7
^Z
```

Agora, considerando que o arquivo “`t.expr`” contenha as entradas a seguir:

```
193
a = 5
b = 6
x = a+b*2
x
(1+2)*3
```

Ao executarmos novamente a calculadora teremos os seguintes resultados:

```
java Calc t.expr
193
17
9
```

## 5 Tradução Dirigida pela Sintaxe

A *Tradução Dirigida pela Sintaxe* (*TDS*) é uma técnica de especificação de compiladores que permite associarmos *regras semânticas* às produções gramaticais, as quais serão avaliadas quando tais produções forem utilizadas no reconhecimento de uma sentença. A avaliação destas regras pode gerar ou interpretar código, armazenar informações na tabela de símbolos, emitir mensagens de erro ou realizar qualquer outra tarefa necessária.

Existem duas notações para associarmos regras semânticas às produções:

- *Definições Dirigidas pela Sintaxe (DDS)*: são especificações de alto nível, escondendo muitos detalhes de implementação, libertando o usuário de especificar a ordem exata onde as traduções ocorrem; e
- *Esquemas de Tradução (ET)*: possibilitam que o usuário evidencie alguns detalhes de implementação, indicando a ordem na qual as traduções devem ocorrer.

As *DDS* e os *ET* são generalizações de gramáticas livres de contexto, nas quais a cada símbolo da gramática pode ser associado um conjunto de atributos, divididos em dois subconjuntos: *Atributos Sintetizados* e *Atributos Herdados*.

Basicamente, podemos pensar em um *atributo* como sendo uma variável a qual é associada a um símbolo gramatical, podendo representar qualquer informação necessária, como uma cadeia de caracteres, um tipo, uma posição de memória, etc. Por exemplo, suponha a produção  $A \rightarrow \alpha$ . Podemos associar o atributo *val* à  $A$  para armazenar o resultado da derivação de  $\alpha$ , o que seria representado por  $A.val$ .

O valor de cada atributo associado a um nó da árvore de derivação é definido pela *regra semântica* associada à produção usada naquele nó. O valor de um *atributo sintetizado* em um nó é calculado a partir dos valores dos atributos dos filhos daquele nó na árvore. O valor de um *atributo herdado* é calculado a partir dos valores dos atributos dos irmãos e pai daquele nó.

As *regras semânticas* estabelecem dependências entre os atributos, as quais serão representadas por um *grafo de dependências*. A partir do *grafo de dependências*, derivamos uma ordem de avaliação para as regras semânticas. A avaliação destas regras define os valores dos atributos nos nós da árvore de derivação para uma dada cadeia de entrada, podendo também produzir efeitos colaterais, como, por exemplo, atualizar ou imprimir o valor de uma variável.

Uma árvore de derivação que mostra os valores dos atributos associados a cada nó é chamada de *árvore de derivação anotada*. O processo para calcular os valores dos atributos de cada nó é chamado de *anotação* ou *decoração* da árvore de derivação.

Conceitualmente, através das duas notações podemos analisar sintaticamente o fluxo de *tokens* presentes na entrada, construindo a árvore gramatical e, em seguida, percorrê-la da forma necessária, avaliando as regras semânticas associadas a cada nó. Casos especiais de *DDS* e *ET* podem ser implementados em um único passo através da avaliação das regras semânticas durante a análise

sintática, sem construir explicitamente a árvore de derivação ou o grafo de dependência entre os atributos.

### 5.1 Definição Dirigida pela Sintaxe

Numa DDS, cada produção gramatical  $A \rightarrow \alpha$  tem associada a si um conjunto de regras semânticas da forma  $b := f(c_1, \dots, c_k)$ , onde  $f$  é uma função e:

1.  $b$  é um *atributo sintetizado* de  $A$  e  $c_1, \dots, c_k$  são atributos pertencentes as símbolos gramaticais da produção; ou
2.  $b$  é um *atributo herdado* pertencente a um dos símbolos gramaticais do lado direito da produção e  $c_1, \dots, c_k$  são atributos pertencentes as símbolos gramaticais da produção.

Em ambos os casos, dizemos que  $b$  depende dos atributos  $c_1, \dots, c_k$ .

Uma *gramática de atributos* é uma DDS na qual as funções nas *regras semânticas* não produzem efeitos colaterais.

Normalmente, as funções nas regras gramaticais serão frequentemente escritas como expressões. Aquelas que causarem efeitos colaterais serão escritas como chamadas de procedimentos ou como fragmentos de programas.

Como exemplo, considere a DDS da Figura 34, a qual é utilizada para especificar um programa para uma calculadora de mesa.

Produções	Regras Semânticas
$L \rightarrow E =$	$imprimir(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow digito$	$F.val := digito.lexval$

Figura 34: DDS de uma calculadora de mesa.

Esta definição associa um atributo sintetizado, chamado *val*, a cada símbolo não-terminal  $E$ ,  $T$  e  $F$ . Para cada produção  $E$ ,  $T$  e  $F$ , a *regra semântica* calcula o atributo *val* para o não-terminal do lado esquerdo a partir dos valores de *val* para os não-terminais do lado direito. A utilização do não terminal  $E_1$  na figura se justifica apenas para podemos diferenciar o não terminal  $E$  do lado esquerdo e do lado direito da regra de produção.

O token *digito* tem um atributo sintetizado *lexval* cujo valor é fornecido pelo *analisador léxico*. A regra semântica associada a produção  $L \rightarrow E =$  é uma rotina que imprime o valor da expressão aritmética gerada por  $E$ .

Em uma *DDS*, símbolos terminais têm apenas atributos sintetizados. Além disso, valores para os atributos de terminais são normalmente fornecidos pelo analisador léxico.

## 5.2 Atributos Sintetizados

O *atributo sintetizado* de um nó é aquele cujo valor depende dos atributos associados aos seus nós *filhos*.

Uma *DDS* que usa extensivamente atributos sintetizados é chamada de *S-atribuída*. Uma árvore de derivação para uma *DDS S-atribuída* pode ser *anotada* através da avaliação ascendente das regras semânticas para os atributos de cada nó.

A *DDS* da Figura 34 é *S-atribuída*, a qual especifica uma calculadora de mesa que lê uma linha contendo uma expressão aritmética formada por dígitos, parênteses, operadores + e \*, seguida do símbolo =, e imprime o valor da expressão. Por exemplo, dada a expressão  $3*5+4=$ , o programa imprimirá o valor 19.

A árvore de derivação anotada para a expressão de entrada  $3*5+4=$ , é dada pela Figura 35. Observe que o resultado impresso na raiz da árvore é o valor *E.val* do filho à esquerda da raiz.

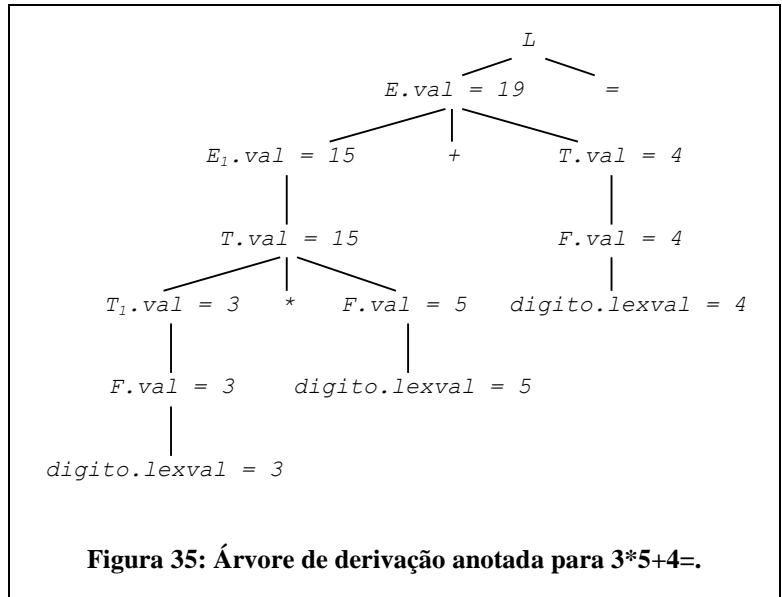


Figura 35: Árvore de derivação anotada para  $3*5+4=$ .

## 5.3 Atributos Herdados

Um *atributo herdado* de um nó é aquele cujo valor é definido em função dos atributos do nó pai e/ou dos nós irmãos. Atributos herdados são convenientes para expressar a dependência de uma construção de linguagem do contexto onde a construção aparece. Por exemplo, pode-se usar um atributo herdado para registrar se um identificador aparece à esquerda ou à direita de uma atribuição a fim de decidir se é necessário salvar o endereço ou o valor do identificador.

Apesar de ser sempre possível rescrever uma *DDS* com apenas atributos sintetizados, é mais natural usar *DDS* com atributos herdados.

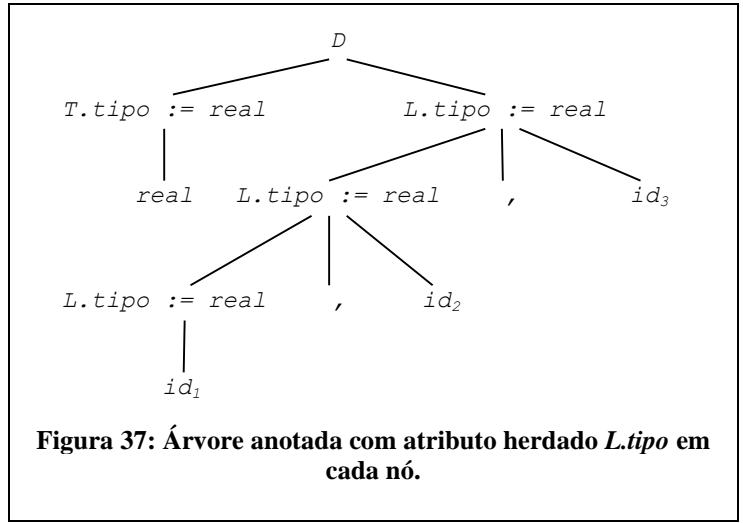
O exemplo Figura 36 ilustra o uso de uma *DDS* com atributos herdados na qual permite distribuir informações de tipo para vários identificadores em uma declaração.

Produções	Regras Semânticas
$D \rightarrow T \ L$	$L.tipo := T.tipo$
$T \rightarrow int$	$T.tipo := inteiro$
$T \rightarrow real$	$T.tipo := real$
$L \rightarrow L_1 , id$	$L_1.tipo := L.tipo$ $IncTipo(id.indice, L.tipo)$
$L \rightarrow id$	$IncTipo(id.indice, L.tipo)$

Figura 36: DDS tendo  $L.tipo$  como atributo herdade.

O não-terminal  $T$  tem um atributo sintetizado  $tipo$ , cujo valor é determinado a partir das produções  $T \rightarrow int$  ou  $T \rightarrow real$ .

A regra semântica  $L.tipo := T.tipo$ , associada à produção  $D \rightarrow TL$ , armazena no atributo herdado  $L.tipo$  o tipo especificado na declaração. As regras semânticas passam este tipo para baixo na árvore de derivação através do atributo  $L.tipo$ . Regras associadas com as produções que definem  $L$  chamam a rotina  $IncTipo()$  para incluir o tipo de cada identificador, apontado pelo atributo  $indice$ , na tabela de símbolos.



A Figura 37 ilustra a árvore de derivação anotada para a sentença:  $real \ id_1, \ id_2, \ id_3$ .

Observe que o valor de  $L.tipo$  determina o tipo da lista de identificadores  $id_1, id_2, id_3$ . Estes valores são determinados calculando-se o valor do atributo  $T.tipo$  junto ao filho à esquerda da raiz e avaliando-se  $L.tipo$  de modo descendente, na árvore à direita da raiz. A cada nó  $L$ , a rotina  $IncTipo()$  é chamada para inserir na tabela de símbolos o fato de que o identificador do filho mais à esquerda deste nó é do tipo  $real$ .

## 5.4 Grafos de Dependência

Se um atributo  $b$  de um nó da árvore de derivação depender de um atributo  $c$ , a regra semântica para  $c$  deve ser avaliada antes da que define  $b$ . As interdependências entre os atributos herdados e sintetizados dos nós de uma árvore de derivação podem ser representados em um grafo dirigido chamado *grafo de dependência*.

Antes de se construir um *grafo de dependência*, deve-se modificar cada regra

semântica para a forma  $b := f(c_1, \dots, c_k)$ , introduzindo um *atributo sintetizado* fictício  $b$  para cada regra semântica que consista em uma chamada de procedimento. O grafo deve possuir um *nó* para cada atributo e um *arco* a partir de  $c$  em direção à  $b$ , evidenciando a relação de dependência.

Para se construir um grafo de dependência basta seguir o seguinte algoritmo:

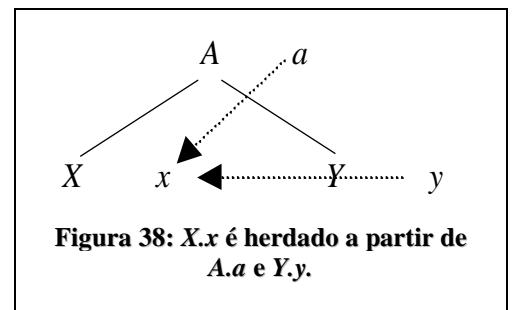
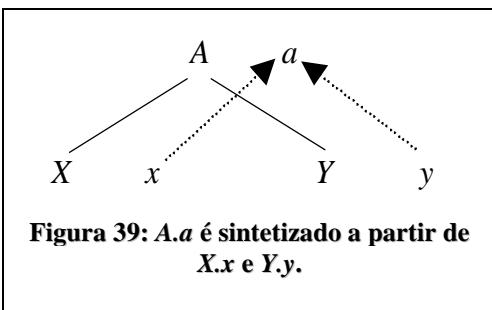
```

Para cada nó n da árvore de derivação faça
    Para cada atributo a do símbolo gramatical em n faça
        Construir um nó no grafo de dependências para a
    Fim Para
Fim Para
Para cada nó n da árvore de derivação faça
    Para cada regra semântica b := f(c1, ..., ck) associada à
        produção usada em n faça
            Para i := 1 to k faça
                Construir um arco a partir de ci até o nó b
            Fim Para
        Fim Para
    Fim Para

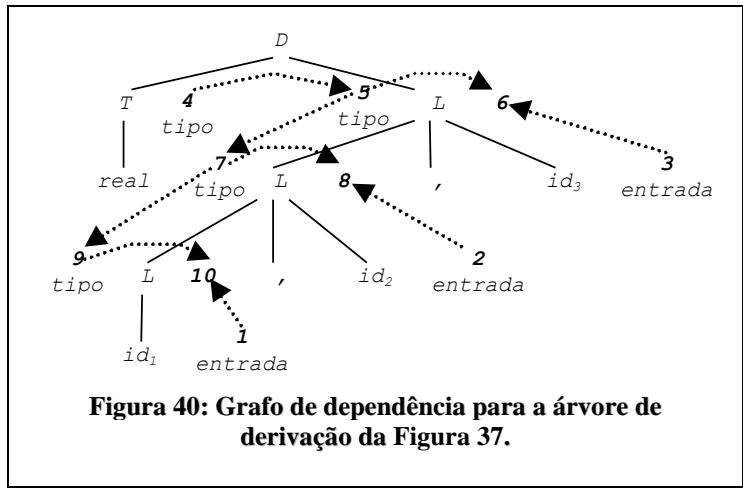
```

Por exemplo, suponha que  $A.a := f(X.x, Y.y)$  seja uma regra semântica para a produção  $A \rightarrow XY$ . Esta regra define o *atributo sintetizado*  $A.a$  que depende dos atributos  $X.x$  e  $Y.y$ . Logo teremos o grafo de dependências ilustrado pela Figura 39.

Se a produção  $A \rightarrow XY$  possuir a regra semântica  $X.x := f(A.a, Y.y)$  associada a si, então  $X.x$  é um *atributo herdado* de  $A.a$  e  $Y.y$ . Portanto, teremos o grafo de dependências ilustrado pela Figura 38.



A Figura 40 ilustra o grafo de dependências para a árvore de derivação da Figura 37. Os nós estão marcados por números os quais indicam a ordem de dependência entre os atributos. Por exemplo, o arco 4–5 indica a dependência entre os atributos  $L.tipo$  e  $T.tipo$ , de acordo com a regra semântica  $L.tipo := T.tipo$ , associada à produção  $D \rightarrow T L$ . Cada uma das regras semânticas  $IncTipo(id.indice, L.tipo)$  associadas às produções  $L$  leva à criação de um atributo fictício. Os nós 6, 8 e 10 são construídos para esses atributos fictícios.



**Figura 40:** Grafo de dependência para a árvore de derivação da Figura 37.

Observe que a avaliação de um atributo  $b$  de um nó do grafo de dependências só pode ocorrer quando os atributos  $c_1, \dots, c_k$ , dos quais ele depende, já tiverem sido sintetizados.

Qualquer *ordenação topológica* de um grafo de dependências provê uma ordem de avaliação válida para as regras semânticas associadas aos nós da árvore de derivação. Isto é, numa ordenação topológica, os atributos  $c_1, \dots, c_k$  de uma regra semântica  $b := f(c_1, \dots, c_k)$  são avaliados antes que  $f$  seja avaliada.

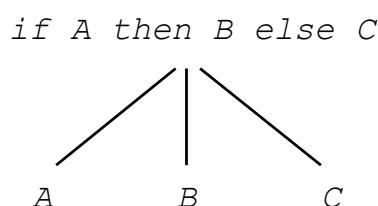
Uma ordenação topológica de um *grafo orientado acíclico* é qualquer ordenação  $m_1, \dots, m_k$  dos nós do grafo, tal que o sentido dos arcos ocorre dos nós que aparecem mais cedo para os nós que aparecem mais tarde na ordenação. Isto é, se  $m_i \rightarrow m_j$  é um arco, então  $m_i$  ocorre antes de  $m_j$  na ordenação. Para o grafo da Figura 40, uma ordenação topológica pode ser obtida listando-se os nós em ordem crescente de numeração.

## 5.5 Construção de Árvores Sintáticas

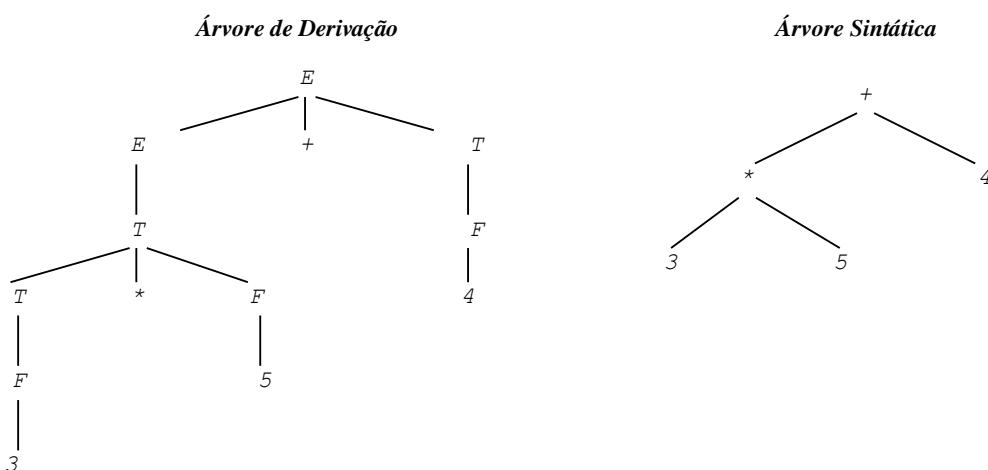
Uma árvore *sintática abstrata* (ou simplesmente *árvore sintática*) é uma forma condensada da árvore de derivação, útil para a representação das construções da linguagem. Nela, os operadores e as palavras chaves não figuram como folhas, mas sim como nós interiores.

Por exemplo, a produção  $S \rightarrow \text{if } A \text{ then } B \text{ else } C$  poderia aparecer como a árvore sintática indicada pela Figura 41.

Outra simplificação encontrada nas árvores sintáticas é a eliminação das cadeias de produção simples, como  $A \rightarrow B$  e  $B \rightarrow C$ . Por exemplo, considerando as produções  $E$ ,  $T$  e  $F$  da Figura 34, a expressão  $3*5+4$ , teria as árvores de derivação e sintática indicadas pela Figura 42.



**Figura 41:** Árvore sintática para  $S \rightarrow \text{if } A \text{ then } B \text{ else } C$ .



**Figura 42:** Árvore de derivação e sintática para a cadeia  $3*5+4$ , de acordo com as produções  $E$ ,  $T$  e  $F$  da gramática da figura 1.

O uso de árvores sintáticas como forma de *representação intermediária* permite que a tradução seja feita de modo independente da análise sintática, evitando assim algumas restrições.

Uma tradução *dirigida por sintaxe* pode basear-se em qualquer das duas formas: árvores de derivação ou árvores sintáticas. Em ambos os casos, os atributos são associados aos nós da árvore.

Em uma árvore sintática para a representação de expressões, construímos subárvores para cada uma das subexpressões, através da criação de um *nó* para cada operador e operandos. Cada nó é composto por três campos: o primeiro campo identifica o operador (comumente chamado de *rótulo*), e os demais contêm apontadores para os demais operandos. Podem existir também campos adicionais para representar os valores dos atributos associados àquele nó.

Usaremos as seguintes funções para criar os nós das árvores sintáticas para as expressões com operadores binários. Cada função retorna um apontador para o nó recém-criado.

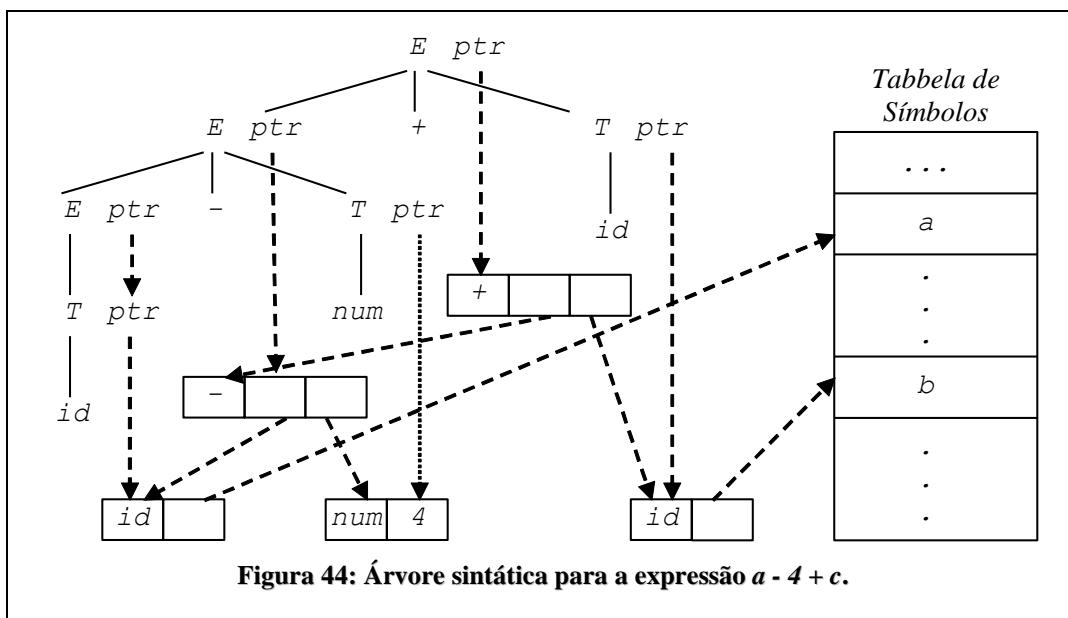
1. *CriaNo(op, esquerdo, direito)*: cria um nó de operador com rótulo *op* e dois campos contendo apontadores para os operandos *esquerdo* e *direito*;
2. *CriaFolha(id, indice)*: cria um nó de identificador, com rótulo *id* e um campo contendo um apontador (*indice*), para a entrada de *id* na tabela de símbolos;
3. *CriaFolha(num, val)*: cria um nó para um número, com rótulo *num* e um campo *val* contendo o valor deste número.

Produções	Regras Semânticas
$E \rightarrow E_1 + T$	$E.ptr := CriaNo('+', E_1.ptr, T.ptr)$
$E \rightarrow E_1 - T$	$E.ptr := CriaNo('-', E_1.ptr, T.ptr)$
$E \rightarrow T$	$E.ptr := T.ptr$
$T \rightarrow (E)$	$T.ptr := E.ptr$
$T \rightarrow id$	$T.ptr := CriaFolha(id, id.indice)$
$T \rightarrow num$	$T.ptr := CriaFolha(num, num.val)$

Figura 43: DDS para a construção da árvore sintática de uma expressão simplificada.

A Figura 44, indicada a seguir, ilustra a utilização destas funções para criação da árvore sintática para a expressão  $a - 4 + c$ , utilizando a DDS da Figura 43.

A árvore de derivação foi construída de modo ascendente. Os nós *E* e *T* da árvore de derivação utilizam o atributo sintetizado *ptr* como apontador para a árvore sintática, a qual é realmente construída. A árvore de derivação pode não ser explicitamente implementada, existindo apenas de forma imaginária.



## 5.6 Avaliação Ascendente de Definições S-Atribuídas

Existem classes de *DDS* para as quais é fácil construir tradutores. Uma destas classes é as *DDS S-Atribuídas*, isto é, definições dirigidas pela sintaxe que contêm apenas *atributos sintetizados*.

*Atributos sintetizados* podem ser avaliados por um *analisador ascendente* a medida que a cadeia de entrada é reconhecida. O analisador consegue manter os valores dos *atributos sintetizados*, associados aos símbolos da gramática, em sua própria *pilha*. Sempre que ocorre uma redução, os valores dos novos *atributos sintetizados* são computados, a partir dos *atributos* que estão na *pilha*, associados aos símbolos do lado direito da produção que está sendo reduzida.

Para se armazenar os valores dos *atributos sintetizados*, são adicionados novos campos na *pilha do analisador*. Por exemplo, suponha a *DDS* a seguir e a pilha sintática cuja situação é indicada pela Figura 45.

Produção	Regra Semântica
$A \rightarrow XYZ$	$A.a := f(X.x, Y.y, Z.z)$

É importante observar que a *pilha sintática* pode ser implementada como um *array* de registros, cada qual contendo o par (*Símbolo*, *Val*).

Antes que *XYZ* seja reduzido para *A*, o valor dos atributos *Z.z*, *Y.y* e *X.x* estão armazenados em *Val[Topo]*, *Val[Topo - 1]* e *Val[Topo - 2]*, respectivamente. Se o símbolo não possui *atributos*, então a entrada *Val* da pilha é indefinida. Após a redução, o *Topo* é decrementado de 2, *A* é armazenado em *Símbolo[Topo]* e o valor do atributo sintetizado *A.a* é colocado em *Val[Topo]*.

Considere agora, a *DDS* da calculadora de mesa, representada pela Figura 34. Os atributos sintetizados na árvore da Figura 35 podem ser avaliados por um analisador ascendente cuja entrada seja  $3 * 5 + 4 =$ . Para tal, devemos modificar o analisador para executar as instruções da figura antes de fazer as reduções apropriadas. O código foi obtido a partir das regras semânticas substituindo-se cada atributo por uma posição correspondente ao campo *Val*, na pilha sintática.

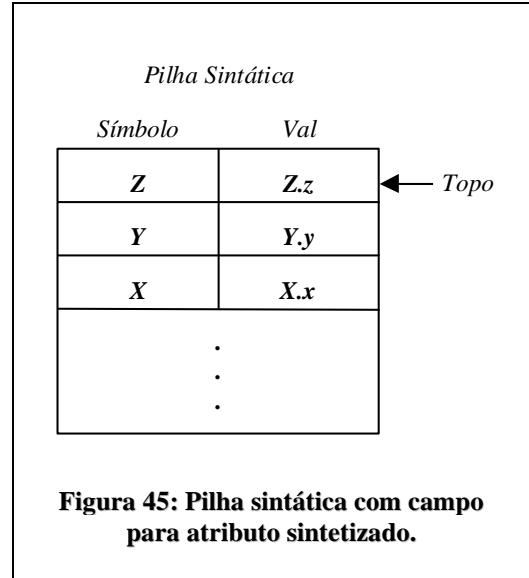


Figura 45: Pilha sintática com campo para atributo sintetizado.

Produções	Regras Semânticas
$L \rightarrow E =$	$imprimir(Val[Topo - 1])$
$E \rightarrow E_1 + T$	$Val[nTopo] := Val[Topo - 2] + Val[Topo]$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$Val[nTopo] := Val[Topo - 2] * Val[Topo]$
$T \rightarrow F$	
$F \rightarrow (E)$	$Val[nTopo] := Val[Topo - 1]$
$F \rightarrow digito$	$Val[Topo] := digito.lexval$

Figura 46: DDS para uma calculadora de mesa para um analisador sintático ascendente.

Quando o analisador sintático reconhece um dígito, o token *digito* é empilhado em *Símbolo[Topo]* e seu atributo é armazenado em *Val[Topo]*.

Observe que o código não mostra como as variáveis *Topo* e *nTopo* são gerenciadas. Entretanto, quando uma produção com *r* símbolos à direita é reduzida, o valor de *nTopo* é feito igual à (*Topo* – *r* + 1). Após a execução de cada instrução, *Topo* é igualado a *nTopo*.

A Figura 47 abaixo mostra a sequência de movimentos realizada para a entrada  $3 * 5 + 4 =$ .

Passo	Entrada	Pilha		Produção Usada
		Símbolo	Val	
1.	$3 * 5 + 4 =$	\$	-	
2.	$* 5 + 4 =$	\$ 3	- 3	
3.	$* 5 + 4 =$	\$ F	- 3	$F \rightarrow digito$
4.	$* 5 + 4 =$	\$ T	- 3	$T \rightarrow F$
5.	$5 + 4$	\$ T *	- 3 -	
6.	$+ 4$	\$ T * 5	- 3 - 5	
7.	$+ 4$	\$ T * F	- 3 - 5	$F \rightarrow digito$
8.	$+ 4$	\$ T	- 15	$T \rightarrow T * F$
9.	$+ 4$	\$ E	- 15	$E \rightarrow T$
10.	$4$	\$ E +	- 15 -	
11.	$=$	\$ E + 4	- 15 - 4	
12.	$=$	\$ E + F	- 15 - 4	$F \rightarrow digito$
13.	$=$	\$ E + T	- 15 - 4	$T \rightarrow F$
14.	$=$	\$ E	- 19	$E \rightarrow E + T$
15.		\$ E =	- 19 -	
16.		\$ L	--	$L \rightarrow E =$

Figura 47: Movimentos realizados pelo analisador sintático para a entrada  $.3 * 5 + 4 =$ .

Observe pela Figura 47 que as ações semânticas são sempre executadas antes de uma redução. Os *Esquemas de Tradução*, considerados na próxima seção, provêm uma notação para se intercalar ações com a análise sintática.

## 5.7 Definições L-Atribuídas

Quando a *tradução* é realizada durante a *análise sintática*, a *ordem de avaliação dos atributos* é feita de acordo com à *ordem de criação dos nós da árvore de derivação*. Uma ordem natural, que caracteriza muitos métodos de análise descendente e ascendente, é aquela obtida através da aplicação do método de *pesquisa em profundidade (depth-first)* para a avaliação dos atributos associados a cada nó da árvore de derivação, o qual é definido pelo seguinte procedimento:

```

Procedimento DepthFirst(n: nó)
  Início
    Para cada filho m de n, da esquerda para a direita, faça
    Início
      Avaliar os atributos herdados de m
      DepthFirst(m)
    Fim
    Avaliar os atributos sintetizados de n
  Fim

```

Os atributos de uma definição *L-Atribuída* podem sempre serem avaliados numa ordem de *pesquisa em profundidade*, visto que a informação dos atributos sempre aparecem no fluxo da esquerda para a direita.

Dizemos que uma *DDS* é *L-Atribuída* se cada atributo herdado de  $X_i$ ,  $1 \leq i \leq n$ , do lado direito de  $A \rightarrow X_1X_2\dots X_n$ , depender somente:

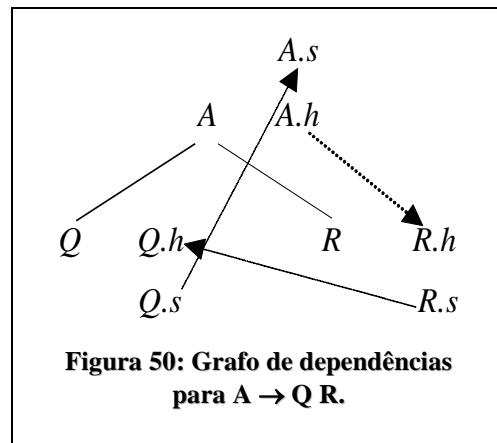
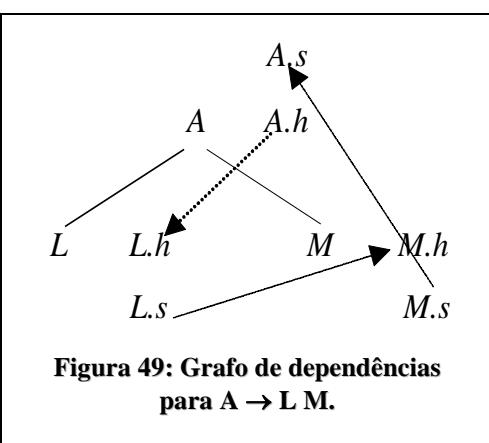
1. Dos atributos dos símbolos  $X_1X_2\dots X_{i-1}$  à esquerda de  $X_i$  na produção e
2. Dos atributos herdados de  $A$ .

Toda definição *S-Atribuída* é *L-Atribuída*, visto que as restrições (1) e (2) se aplicam somente aos *atributos herdados*.

Por exemplo, a *DDS* a seguir não é *L-Atribuída*, pois o atributo herdado  $Q.h$  do símbolo gramatical  $Q$  depende do atributo sintetizado  $R.s$  do símbolo gramatical à sua direita, conforme indicado pela Figura 50.

Produções	Regras Semânticas
$A \rightarrow LM$	$L.h := l(A.h)$ $M.h := m(L.s)$ $A.s := f(M.s)$
$A \rightarrow QR$	$R.h := r(A.h)$ $Q.h := q(R.s)$ $A.s := f(Q.s)$

Figura 48: Exemplo de *DDS* que não é *L-Atribuída*.



## 5.8 Esquemas de Tradução

Um *Esquema de Tradução* (*ET*) é uma gramática livre de contexto na qual os atributos são associados aos símbolos gramaticais e as ações são envolvidas entre chaves e inseridas ao lado direito das produções, podendo ter *atributos sintetizados* e *herdados*. *Esquemas de Tradução* constituem notações apropriadas para a especificação de tradução durante a sintaxe.

Por exemplo, o *ET* da Figura 51 mapeia expressões infixas para pós-fixas.

Produções
$E \rightarrow T\ R$
$R \rightarrow O\ T \quad \{ \text{imprimir}(O.\text{Lexema}) \} \ R_1$
$R \rightarrow \epsilon$
$T \rightarrow \text{num} \quad \{ \text{imprimir}(\text{num}.val) \}$
$O \rightarrow + \mid - \quad \{ O.\text{Lexema} = \text{Lexico}.Lexema \}$

Figura 51: Esquema de Tradução para converter expressões in-fixas para pós-fixas.

A Figura 52 ilustra a árvore de derivação para a expressão  $9-5+2$  utilizando-se do *ET* da Figura 51. As ações aparecem associadas aos nós apropriados. Ações são tratadas como símbolos terminais. Quando executadas em uma pesquisa em profundidade, imprimem a expressão  $95-2+$ .

Na definição de um *ET* deve-se assegurar que os valores dos atributos estejam disponíveis sempre que as ações se referirem a eles.

No caso mais simples, quando são necessários apenas *atributos sintetizados*, pode-se construir o *ET* criando uma ação que consiste de uma atribuição para cada regra semântica, e colocando esta ação no final do lado direito da produção associada.

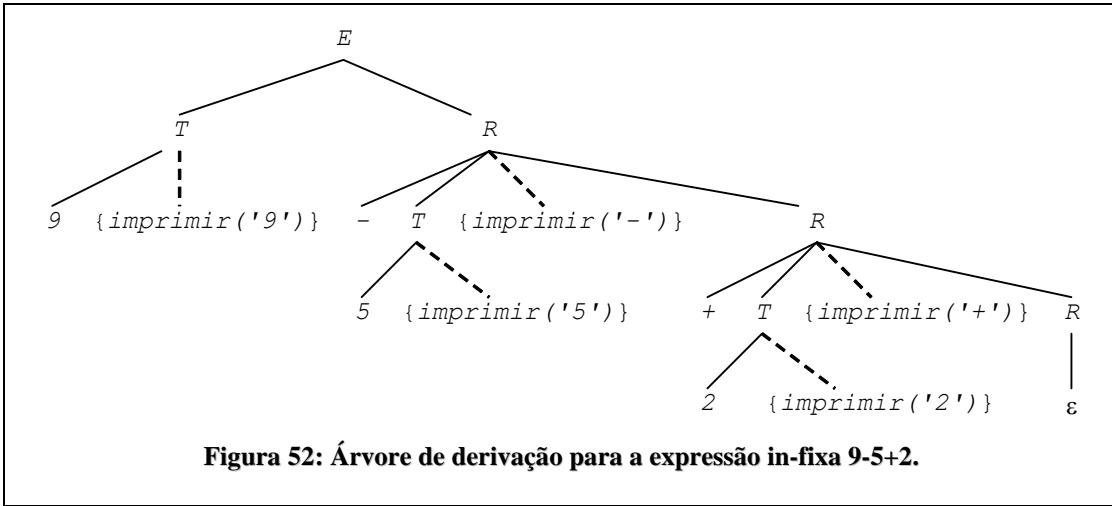


Figura 52: Árvore de derivação para a expressão in-fixa 9-5+2.

Se existirem ambos, atributos sintetizados e herdados, deve-se proceder da seguinte forma:

1. Um atributo herdado para cada símbolo no lado direito de uma produção deve ser calculado em uma ação antes deste símbolo;
2. Uma ação não deve se referir a um atributo sintetizado de um símbolo à direita desta ação;
3. Um atributo sintetizado do não-terminal à esquerda pode ser calculado somente depois que todos os atributos que ele referencia tenham sido calculados. A ação que calcula tais atributos pode, em geral, ser especificada no final da produção.

O  $ET$  abaixo não satisfaz o primeiro dos requisitos acima:

$$S \rightarrow A_1 A_2 \{A_1.h := 1; A_2.h := 2\}$$

$$A \rightarrow a \{imprimir(A.h)\}$$

O atributo herdado  $A.h$ , na segunda produção, ainda não está definido quando uma tentativa de imprimir seu valor é feita, durante uma travessia em uma *pesquisa em profundidade* da árvore de derivação, para a entrada  $a$ . A travessia inicia em  $S$  e visita as sub-árvores  $A_1$  e  $A_2$  antes que os valores de  $A_1.h$  e  $A_2.h$  sejam atribuídos.

Sempre é possível iniciar com uma  $DDS$  e construir um  $ET$  que satisfaça os três requisitos acima, conforme analisado na próxima seção.

## 5.9 Tradução Descendente (Top-Down)

Nesta seção,  $ET$  serão implementados durante a análise preditiva. Trabalharemos com  $ET$  ao invés de  $DDS$  de modo que poderemos explicitar a ordem na qual as ações e as avaliações de atributos serão executadas. A transformação se aplica à  $ET$  com atributos sintetizados.

Para adaptar  $ET$  com produções recursivas à esquerda para análise preditiva, deve-se eliminar a recursividade à esquerda e fazer a transformação correspondente à avaliação de atributos.

Por exemplo, suponha a  $DDS$  da Figura 53. Podemos transformá-la para o  $ET$  indicado pela Figura 54.

Produções	Regras Semânticas
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow E_1 - T$	$E.val := E_1.val - T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow (E)$	$T.val := E.val$
$T \rightarrow num$	$T.val := num.val$

Figura 53: DDS para expressões aritméticas simples.

Produções
$E \rightarrow E_1 + T \{E.val := E_1.val + T.val\}$
$E \rightarrow E_1 - T \{E.val := E_1.val - T.val\}$
$E \rightarrow T \{E.val := T.val\}$
$T \rightarrow (E) \{T.val := E.val\}$
$T \rightarrow num \{T.val := num.val\}$

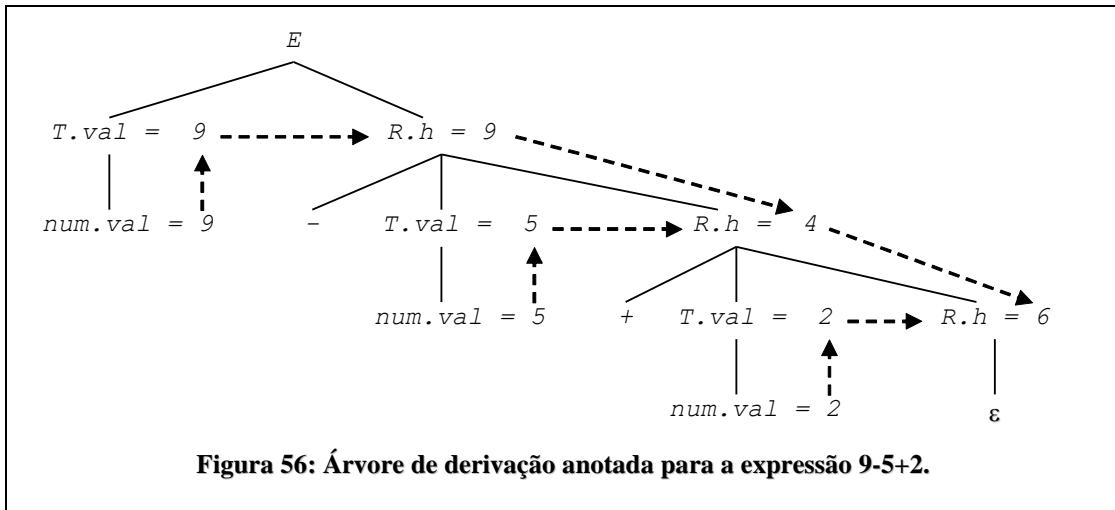
Figura 54: Esquema de Tradução para expressões aritméticas simples.

Eliminando-se a recursividade à esquerda e realizando as transformações correspondentes para a avaliação de atributos, obtém-se o *Esquema de Tradução* da figura a baixo.

Produções
$E \rightarrow T \{R.h := T.val\}$
$R \{E.val := R.s\}$
$R \rightarrow +$
$T \{R_1.h := R.h + T.val\}$
$R_1 \{R.s := R_1.s\}$
$R \rightarrow -$
$T \{R_1.h := R.h - T.val\}$
$R_1 \{R.s := R_1.s\}$
$R \rightarrow \epsilon \{R.s := R.h\}$
$T \rightarrow (E) \{T.val := E.val\}$
$T \rightarrow num \{T.val := num.val\}$

Figura 55: Esquema de Tradução para expressões aritméticas simples.

Pelo Esquema de Tradução acima, a árvore de derivação anotada para a expressão  $9-5+2$  é construída conforme indicado pela Figura 56. As setas mostram o caminho de determinação da expressão.



Para a análise descendente assume-se que uma ação é executada no momento em que um símbolo na mesma posição é expandido. Assim, na segunda produção ( $R \rightarrow + T R_I$ ), a atribuição  $R_I.h := R.h + T.val$  é executada após  $T$  ter sido totalmente expandido para terminais, enquanto que a segunda ação ( $R.s := R_I.s$ ) é executada após  $R_I$  ter sido totalmente expandido. No caso de definições *L-Atribuídas*, um *atributo herdado* de um símbolo deve ser calculado por uma ação que aparece antes do símbolo, e um atributo sintetizado do não-terminal à esquerda deve ser calculado após todos os atributos dos quais ele depende tenham sido calculados.

A Figura 57 ilustra a transformação da DDS da Figura 43, para criação de árvores sintáticas, para *ET*. A eliminação da recursão à esquerda deste esquema produz o *ET* representado pela Figura 59. A árvore sintática construída para a expressão  $a-4+c$  é indicada pela Figura 58. Nela, os *atributos sintetizados* são mostrados à direita de cada nó e os *atributos herdados* são mostrados à esquerda.

Produções
$E \rightarrow E_1 + T \{E.ptr := CriaNo('+', E_1.ptr, T.ptr)\}$
$E \rightarrow E_1 - T \{E.ptr := CriaNo('-', E_1.ptr, T.ptr)\}$
$E \rightarrow T \{E.ptr := T.ptr\}$
$T \rightarrow (E) \{T.ptr := E.ptr\}$
$T \rightarrow id \{T.ptr := CriaFolha(id, id.indice)\}$
$T \rightarrow num \{T.ptr := CriaFolha(num, num.val)\}$

**Figura 57:** Esquema de Tradução para a construção da árvore sintática de uma expressão simplificada.

Produções	
$E \rightarrow T$	{ $R.h := T.ptr$ }
$R$	{ $E.ptr := R.s$ }
$R \rightarrow +$	
$T$	{ $R_1.h := CriaNo('+', R.h, T.ptr)$ }
$R_1$	{ $R.s := R_1.s$ }
$R \rightarrow -$	
$T$	{ $R_1.h := CriaNo('-', R.h, T.ptr)$ }
$R$	{ $R.s := R_1.s$ }
$R \rightarrow \epsilon$	{ $R.s := R.h$ }
$T \rightarrow (E)$	{ $T.ptr := E.ptr$ }
$T \rightarrow id$	{ $T.ptr := CriaFolha(id, id.indice)$ }
$T \rightarrow num$	{ $T.ptr := CriaFolha(num, num.val)$ }

Figura 59: Esquema de Tradução sem recursão a esquerda para a construção da árvore sintática de uma expressão simplificada.

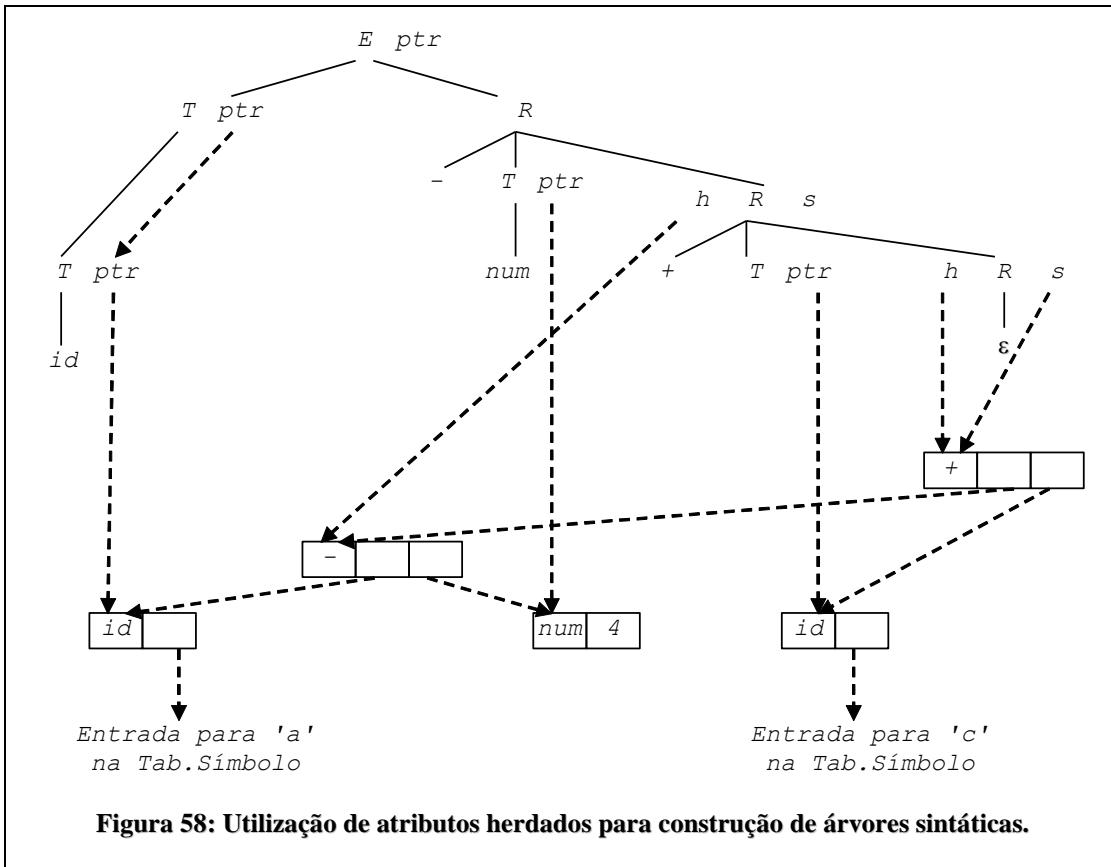


Figura 58: Utilização de atributos herdados para construção de árvores sintáticas.

### 5.10 Projeto de um Tradutor Preditivo

Para construirmos um *analisador sintático preditivo* o qual implementa um *ET* baseado em uma gramática adequada à análise descendente, alguns critérios devem ser respeitados:

1. Para cada produção não-terminal  $A$ , devemos construir uma função que contenha um *parâmetro formal* para cada *atributo herdado* de  $A$  e que retorne os valores dos *atributos sintetizados* de  $A$ . A função para  $A$  deve possuir uma variável local para cada atributo de cada símbolo gramatical que figure em uma produção para  $A$ .
2. O código para o não-terminal  $A$  decide que produção usar, baseado no símbolo corrente de entrada.
3. Considerando os *tokens*, os não-terminais e as ações no lado direito da produção, da esquerda para a direita, o código associado a cada produção realiza:
  - a) Para o *token*  $X$  com atributo sintetizado  $x$ , salvar o valor de  $x$  na variável declarada para  $X.x$ . Em seguida, gerar uma chamada para reconhecer o *token*  $X$  e avançar a entrada.
  - b) Para o não-terminal  $B$ , gerar uma atribuição  $c := B(b_1, b_2, \dots, b_k)$  com uma chamada de função do lado direito, onde  $b_1, b_2, \dots, b_k$  são as variáveis para os *atributos herdados* de  $B$  e  $c$  é a variável para o *atributo sintetizado* de  $B$ .
  - c) Para uma ação, copiar o código dentro do analisador sintático, substituindo cada referência a um atributo pela variável para aquele atributo.

O exemplo a seguir ilustra como proceder para a construção de um analisador sintático descendente preditivo que implementa o *ET* da Figura 59. A partir dos atributos dos não terminais da gramática, obtemos os seguintes tipos de argumentos e resultados para as funções de  $E$ ,  $R$  e  $T$ :

**Função  $E()$  como  $\uparrow$  nó\_arvore\_sintatica**  
**Função  $R(h$  como  $\uparrow$  nó\_arvore\_sintatica) como  $\uparrow$  nó\_arvore\_sintatica**  
**Função  $T()$  como  $\uparrow$  nó\_arvore\_sintatica**

Observe que uma vez que  $E$  e  $T$  não possuem *atributos herdados*, então não possuem argumentos. Entretanto, todos possuem um valor de retorno, o qual corresponde ao nó da árvore sintática criado, isto é, o atributo sintetizado  $ptr$ .

Podemos ainda combinar duas produções  $R$ , da Figura 59, para tornarmos o tradutor menor. As novas traduções usam o *token* *OpAdi* para representar os operadores + e -, obtendo as produções da Figura 60.

Produções	
$E \rightarrow T$	{ $R.h := T.ptr$ }
$R$	{ $E.ptr := R.s$ }
$R \rightarrow OpAdi$	
$T$	{ $R1.h := CriaNo(OpAdi.Lexema, R.h, T.ptr)$ }
$R_1$	{ $R.s := R1.s$ }
$R \rightarrow \epsilon$	{ $R.s := R.h$ }
$T \rightarrow (E)$	{ $T.ptr := E.ptr$ }
$T \rightarrow id$	{ $T.ptr := CriaFolha(id, id.indice)$ }
$T \rightarrow num$	{ $T.ptr := CriaFolha(num, num.val)$ }

Figura 60: Esquema de Tradução apropriado para um analisador sintático preditivo para a construção da árvore sintática de uma expressão simplificada.

O código para a produção  $R$ , construído de acordo com os critérios definidos anteriormente, é indicado a seguir.

```

Função R(Rh como  $\uparrow$  nó_árvore_sintática) como  $\uparrow$  nó_árvore_sintática
Declare
    Tptr, R1h, Rs, R1s como  $\uparrow$  nó_árvore_sintática
    op como caracter
Início
    Se Token = OpAdi então           // Produção R → OpAdi T R
        op <- OpAdi.lexval
        Léxico()
        Tptr <- T()
        R1h <- CriaNo(op, Rh, Tptr)
        R1s <- R(R1h)
        Rs <- R1s
    Senão                               // Produção R → ε
        Rs := Rh
    Fim Se
    Retornar Rs
Fim

```

No código acima, a variável  $R1h$  corresponde ao *atributo herdado*  $R1.h$  e  $R1s$  corresponde ao *atributo sintetizado*  $R1.s$ .

O código para as produções  $E$  e  $T$  podem ser obtidos de modo semelhante.

## 6 Geração de Código Intermediário

Após a fase de análise, um compilador pode proceder a geração de código objeto para uma máquina real ou abstrata, sendo essa última uma representação intermediária. Nesse caso, dizemos que o código gerado é intermediário.

A geração de código intermediário apresenta algumas vantagens:

1. O código gerado é independente de máquina, livrando o programador de detalhes da máquina alvo;
2. Possibilita a sua otimização, independentemente da máquina alvo, obtendo um código mais eficiente;
3. Simplifica a tarefa de implementação do compilador, visto que as dificuldades de tradução do código fonte para objeto podem ser resolvidas gradativamente;
4. Permite a tradução do código intermediário para diversas máquinas alvo.

A desvantagem da geração de código intermediário é a introdução de mais um passo ao compilador, tornando a compilação um pouco mais lenta.

### 6.1 Representações Intermediárias

Existem vários tipos de representações intermediários, mas todas elas recaem em um dos três tipos:

1. Representações gráficas: *árvore sintáticas* e *grafos sintáticos*;
2. Notações *pós-fixa* ou *pré-fixa*;
3. Código de três endereços: *quádruplas* e *triplas*.

Analisamos a seguir os três tipos.

#### 6.1.1 Árvores e Grafos Sintáticos

Uma árvore sintática permite representar a estrutura hierárquica natural do programa fonte sendo compilado. Trata-se de uma forma condensada da árvore de derivação na qual somente os operandos aparecem como folhas da árvore. Os operadores constituem nós interiores da árvore. Por exemplo, a Figura 61 ilustra uma árvore sintática para o comando de atribuição PASCAL:

$$a := (b + c) * (b + c);$$

A DDS responsável por construir a árvore da Figura 61 é indicada pela Figura 64. O resultado da aplicação dessa DDS é indicado na Figura 63.

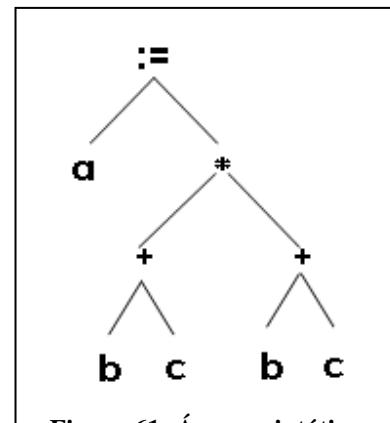


Figura 61: Árvore sintática para  $a := (b + c) * (b + c)$ .

Um grafo sintático fornece as mesmas informações de uma árvore sintática, mas de uma forma mais compacta, pois identifica e efetua a fatoração de subexpressões comuns. A Figura 62 ilustra um grafo sintático para a instrução de atribuição descrita anteriormente.

Para a construção do grafo sintático, podemos utilizar a DDS da Figura 64. Entretanto, devemos modificar as funções `CriaFolha()` e `CriaNo()` para que elas verifiquem, antes de construir um nó, se já existe algum nó identifico a ele. Caso exista, um ponteiro para esse nó é retornado, senão um novo é criado, sendo o seu ponteiro retornado.

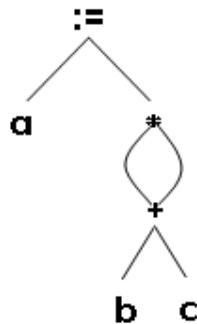


Figura 62: Grafo para  $a := (b + c) * (b + c)$ .

Produções	Ações Semânticas
$S \rightarrow id := E$	$t := CriaFolha(id, id.indice)$ $S.ptr := CriaNo(':=', t, E.ptr)$
$E \rightarrow E_1 + E_2$	$E.ptr := CriaNo('+', E_1.ptr, E_2.ptr)$
$E \rightarrow E_1 * E_2$	$E.ptr := CriaNo('*', E_1.ptr, E_2.ptr)$
$E \rightarrow ( E_1 )$	$E.ptr := E_1.ptr$
$E \rightarrow id$	$E.ptr := CriaFolha(id, id.indice)$

Figura 64: DDS para a construção da árvore sintática para  $a := (b + c) * (b + c)$ .

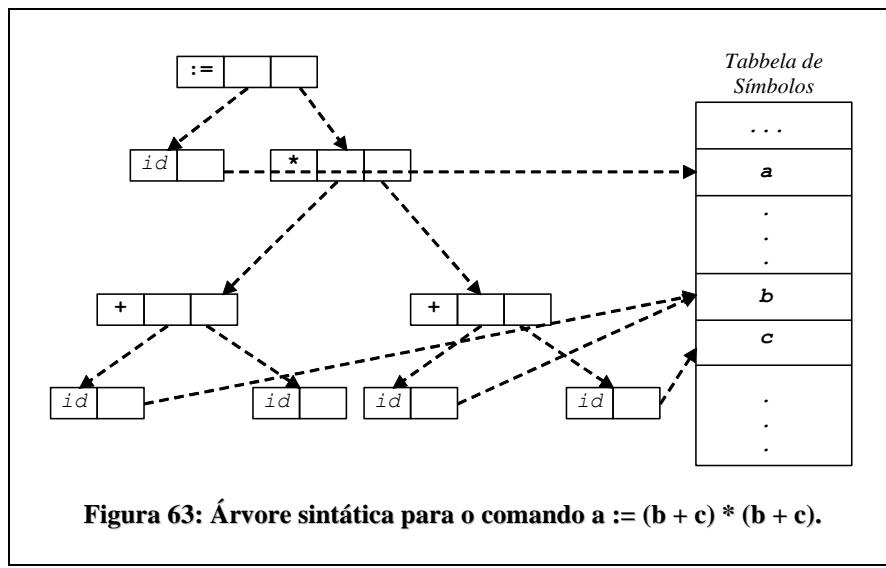


Figura 63: Árvore sintática para o comando  $a := (b + c) * (b + c)$ .

## 6.1.2 Notações Infixa, Pós-fixada e Pré-fixada

Podemos definir a notação infixa, para expressões aritméticas, por recorrência, do seguinte modo:

1. Caso básico:

Todo **identificador** e **número** está na forma infixa;

2. Passo recorrente:

a)  $(E)$  está na forma infixa se, e somente se,  $E$  está na forma infixa;

b)  $E_1 \ op \ E_2$  está na forma infixa se, e somente se,  $E_1$  e  $E_2$  estão na forma infixa e  $op$  é um operador binário.

Raciocínio semelhante pode ser adotado para outros tipos de construções. De acordo com a definição, a expressão:  $(\text{Valor} * \text{Percentual}) / 100$ ; está na forma infixa, mas a expressão:  $\text{Valor} \ \text{Percentual} * 100 /$ ; não. Esta última está na forma pós-fixada. Podemos definir a forma pós-fixada por:

1. Caso básico:

Todo **identificador** e **número** está na forma pós-fixada;

2. Passo recorrente:

$E_1 \ E_2 \ op$  está na forma pós-fixada se, e somente se,  $E_1$  e  $E_2$  estão na forma pós-fixada e  $op$  é um operador binário.

Observe que para a notação pós-fixada o uso do parêntese é desnecessário, pois a ordem de avaliação é feita lendo-se a expressão da esquerda para a direita, respeitando-se a ordem dos argumentos. A avaliação de expressões desse tipo pode ser feita utilizando-se uma pilha, sendo o resultado da operação deixado no topo da pilha, conforme é indicado pelo algoritmo abaixo.

```
Enquanto a expressão não tiver sido totalmente percorrida, da esquerda
para a direita, faça
    Leia o próximo elemento da expressão
    Se o elemento lido é um operando, então
        Empilhe o elemento
    Senão
        Retire os dois operandos sobre o topo da pilha, aplique-lhes a
        operação indicada pelo operador e empilhe o resultado
    Fim se
Fim do enquanto
```

A notação pré-fixada pode ser obtida de forma análoga a da pós-fixada, substituindo no passo recorrente a expressão  $E_1 \ E_2 \ op$  por  $op \ E_1 \ E_2$ .

A figura ilustra algumas expressões representadas em notação infixa, pré-fixada e pós-fixada.

Infixada	Pré-fixada	Pós-fixada
$a + b$	$+ a b$	$a b +$
$a + b * c$	$+ a * b c$	$a b c * +$
$(a + b) * c$	$* + a b c$	$a b + c *$
$(a + b) * (a - c)$	$* + a b - a c$	$a b + a c - *$
$(a + b - c) * d / e$	$/ * - + a b c d e$	$a b + c - d * e /$

Figura 65: Notações infixadas, pré-fixadas e pós-fixadas.

A DDS da Figura 66 permite transformar uma expressão aritmética na forma infixada para suas correspondentes pré-fixada e pós-fixada. Nela, a operação  $a \parallel b$  significa a concatenação do atributo  $a$  com o atributo  $b$ . A expressão pré-fixada resultante do processo de conversão se encontra armazenada no atributo  $E.\text{pre}$ , enquanto a pós-fixada se encontra no atributo  $E.\text{pos}$ .

Produções	Ações Semânticas
$E \rightarrow E_1 + T$	$E.\text{pre} := '+' \parallel E_1.\text{pre} \parallel T.\text{pre}$ $E.\text{pos} := E_1.\text{pos} \parallel T.\text{pos} \parallel '+'$
$E \rightarrow T$	$E.\text{pre} := T.\text{pre}$ $E.\text{pos} := T.\text{pos}$
$T \rightarrow T_1 * F$	$T.\text{pre} := '*' \parallel T_1.\text{pre} \parallel F.\text{pre}$ $T.\text{pos} := T_1.\text{pos} \parallel F.\text{pos} \parallel '*'$
$T \rightarrow F$	$T.\text{pre} := F.\text{pre}$ $T.\text{pos} := F.\text{pos}$
$F \rightarrow ( E )$	$F.\text{pre} := E.\text{pre}$ $F.\text{pos} := E.\text{pos}$
$F \rightarrow \text{id}$	$F.\text{pre} := \text{id}.\text{LexVal}$ $F.\text{pos} := \text{id}.\text{LexVal}$
$F \rightarrow \text{num}$	$F.\text{pre} := \text{num}.\text{LexVal}$ $F.\text{pos} := \text{num}.\text{LexVal}$

Figura 66: DDS para calcular a notação pré-fixada e pós-fixada.

## 6.1.3 Código de Três Endereços

O código intermediário de três endereços permite que cada instrução faça referência a no máximo três endereços de memória – daí o seu nome. Trata-se de um formato de grande utilidade na geração de código, principalmente para arquiteturas baseadas em registradores.

Basicamente, todas arquiteturas de computadores suportam quatro tipos básicos de instruções: operações unárias e binárias, movimentação de dados, desvio incondicional e condicional. Além dessas, outras são adicionadas para oferecer uma maior facilidade ao programador, como chamada e retorno de procedimentos, manipulação de variáveis indexadas, atribuição de endereços e de apontadores. São exatamente estes conjuntos de instruções suportadas pelo código de três endereços. Nelas, as letras A, X, R e p representam endereços de memória, B e C representam endereços de memória, constantes ou literais, n representa um literal, op denota um operador unário ou binário e oprel representa um operador relacional:

1. Operações aritméticas, lógicas ou algum outro tipo de operação:
  - a. Unárias: **A := op B** – aplica o operador unário op a B e armazena o resultado em A. Por exemplo:  $A := -u B$  ou  $A := \text{NOT } B$ .
  - b. Binárias: **A := B op C** – aplica o operador binário op a A e B, armazenando o resultado em A. Por exemplo:  $A := B + C$  ou  $A := B \text{ AND } C$ .
2. Movimentação de dados: **A := B** – atribui o valor de B a A.
3. Desvio incondicional: **goto R** – desvia o fluxo de execução para o endereço R.
4. Desvio condicional: **if B oprel C goto R** – efetua a operação relacional definida por oprel entre B e C, desviando o fluxo de execução para R, caso a operação seja bem sucedida.
5. Manipulação de procedimento:
  - a. Passagem de parâmetros: **param B** – permite a passagem do parâmetro B.
  - b. Chamada de procedimento: **call p, n** – permite a chamada do procedimento p, onde n é a quantidade de argumentos.
  - c. Retorno de procedimento: **return C** – retorna de um procedimento, onde C é o valor retornado, caso exista.
6. Atribuições indexadas:
  - a. Movimentação de dados de um vetor: **X := A[i]** – armazena em X o valor contido no endereço de memória deslocado em i unidades a partir do endereço de A.

- b. Movimentação de dados para um vetor:  $\text{A}[i] := \text{B}$  – armazena no endereço de memória deslocado em  $i$  unidades a partir de  $\text{A}$  o valor de  $\text{X}$ .
7. Atribuição de endereços e apontadores:
- a. Atribuição de endereço:  $\text{x} := \&\text{A}$  – armazena em  $\text{x}$  o endereço de memória correspondente a localização de  $\text{A}$ .
  - b. Atribuição de valor apontado:  $\text{x} := * \text{A}$  – armazena em  $\text{x}$  o valor contido no endereço de memória apontado por  $\text{A}$ . Isto é, o valor armazenado em  $\text{A}$  se refere a um endereço de memória onde o dado realmente se encontra armazenado. Trata-se de um endereçamento indireto.
  - c. Atribuição a um apontador:  $*\text{x} := \text{B}$  – armazena no endereço de memória apontado por  $\text{x}$  o valor de  $\text{B}$ . Isto é, o valor armazenado em  $\text{x}$  se refere a um endereço de memória onde o dado deverá ser armazenado. Novamente, trata-se de um endereçamento indireto.

A utilização do código de três endereços exige o uso de variáveis temporárias, que são criadas, quando necessário, para armazenamento temporário dos resultados das operações intermediárias, devendo serem destruídas após sua utilização. Elas armazenam os resultados obtidos pela execução das ações semânticas contidas em cada nó interior da árvore sintática.

Por exemplo, suponha a expressão de atribuição:  $a := (b + c) * (b + c)$ . Esta expressão gera a árvore da Figura 68, conforme visto anteriormente. O código de três endereços correspondente a compilação desta expressão é:

```
$1 := b + c
$2 := b + c
$3 := $1 * $2
a := $3
```

onde  $\$1$ ,  $\$2$  e  $\$3$  são variáveis temporárias.

Para o GDA da Figura 62 este código é menor, conforme pode ser observado pela Figura 67:

```
$1 := b + c
$2 := $1 * $1
a := $2
```

As demais construções serão analisadas posteriormente.

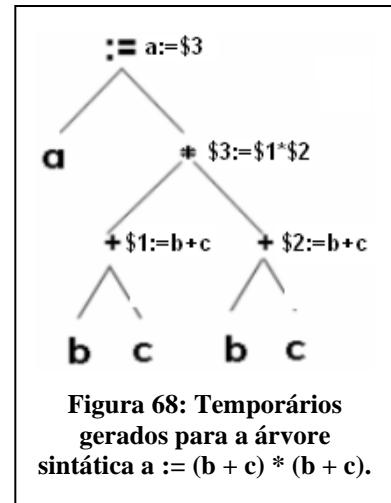


Figura 68: Temporários gerados para a árvore sintática  $a := (b + c) * (b + c)$ .

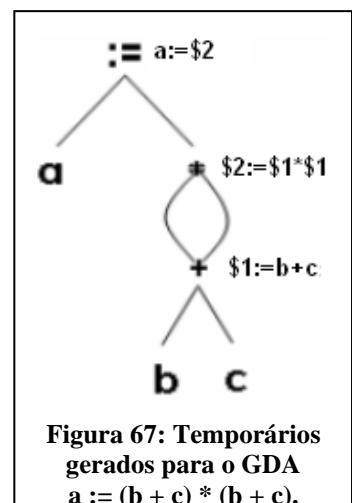


Figura 67: Temporários gerados para o GDA  $a := (b + c) * (b + c)$ .

### 6.1.4 Quádruplas e Triplas

Quando geramos um código de três endereços, durante o processo de compilação, podemos armazená-lo em uma estrutura de dados do tipo vetor contendo campos para o armazenamento do operador e operandos, denominadas quádruplas ou triplas.

As quádruplas são estruturas de dados contendo quatro campos: um para o operador, dois para os operandos e um para o resultado. Dessa forma, a compilação da expressão de atribuição  $a := -b + c * d$ , seria aquela indicada pela Tabela 12.

Tabela 12: Quádrupla para a expressão  $a := -b + c * d$ .

Endereço	Operador	Operando 1	Operando 2	Resultado
1	$-u$	b		\$1
2	*	c	d	\$2
3	+	\$1	\$2	\$3
4	$:=$	\$3		a

As triplas, por outro lado, não oferecem o campo para armazenamento do resultado. Na verdade, esta representação utiliza ponteiros para a própria estrutura onde os temporários foram calculados, evitando a nomeação explícita de temporários e, consequentemente, a sua instalação na tabela de símbolos. Por exemplo, a compilação em triplas da expressão de atribuição  $a := -b + c * d$ , seria aquela indicada pela Tabela 13.

Tabela 13: Triplas para a expressão  $a := -b + c * d$ .

Endereço	Operador	Operando 1	Operando 2
1	$-u$	B	
2	*	C	d
3	+	(1)	(2)
4	$:=$	(3)	

De acordo com a Tabela 13, (1), (2) e (3) representam os endereços dos temporários 1, 2 e 3, correspondendo aos endereços na estrutura onde foram calculados.

A figura especifica uma DDS que gera código de três endereços para comandos de atribuição, envolvendo expressões aritméticas simplificadas, e a condicional if-then-else. Deve ser observado que na implementação da condicional são utilizados rótulos, que são endereços simbólicos calculados pela função

`GeraRotulo()`, a qual ao ser chamada retorna sempre um novo nome simbólico. A função `GeraTemporario()`, gera e retorna um novo nome de temporário. O atributo `code` armazena o código gerado, enquanto `nome` armazena o nome de um identificador ou de um temporário. Na compilação do `if-then-else`, nesse exemplo, é assumido que **FALSO** = 0 e **VERDADE** é qualquer valor diferente de 0. Portanto, em `GeraCode('if' E.nome '= 0 then goto' R1)` estamos gerando uma instrução para testar se o resultado da expressão `E` é falso, desviando para o rótulo `R1`, caso afirmativo. Senão, após da execução do bloco `then`, desviamos para o rótulo `R2`. A execução dessas instruções pode ser melhor entendida pela Figura 70.

Produções	Ações Semânticas
$S \rightarrow id := E$	<code>S.code := E.code    GeraCode(id.nome ':=' E.nome)</code>
$S \rightarrow if E$ $\quad then S_1$ $\quad else S_2$	<code>R1 := GeraRotulo()</code> <code>R2 := GeraRotulo()</code> <code>S.code := E.code   </code> <code>  GeraCode('if' E.nome '= 0 then goto' R1)   </code> <code>  S1.code   </code> <code>  GeraCode('goto' R2)   </code> <code>  GeraCode(R1 ':')   </code> <code>  S2.code   </code> <code>  GeraCode(R2 ':')</code>
$E \rightarrow - E_1$	<code>E.nome := GeraTemporario()</code> <code>E.code := E1.code   </code> <code>  GeraCode(E.nome ':=' '-u' E1.nome)</code>
$E \rightarrow E_1 + E_2$	<code>E.nome := GeraTemporario()</code> <code>E.code := E1.code    E2.code   </code> <code>  GeraCode(E.nome ':=' E1.nome '+' E2.nome)</code>
$E \rightarrow E_1 * E_2$	<code>E.nome := GeraTemporario()</code> <code>E.code := E1.code    E2.code   </code> <code>  GeraCode(E.nome ':=' E1.nome '*' E2.nome)</code>
$E \rightarrow ( E_1 )$	<code>E.nome := E1.nome</code> <code>E.code := E1.code</code>
$E \rightarrow id$	<code>E.nome := id.nome</code> <code>E.code := ''</code>

Figura 69: DDS para gerar código de três endereços para comandos de atribuição e if-then-else.

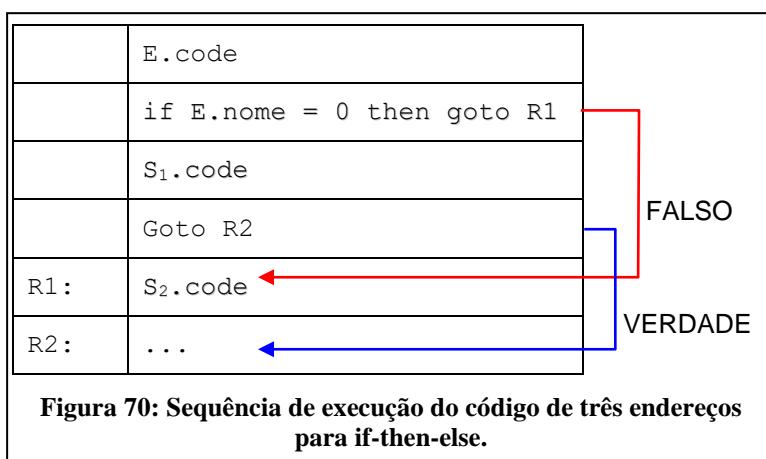


Figura 70: Sequência de execução do código de três endereços para if-then-else.

### 6.1.5 Reutilização de temporários

Podemos efetuar a reutilização dos temporários, evitando assim a sobrecarga da tabela de símbolos. Isto pode ser feito através da seguinte estratégia:

1. Antes da compilação criamos um contador, denominado C, e o inicializamos com o valor 1;
2. Ao criarmos um novo temporário devemos fazê-lo por intermédio de '\$' + C e, a seguir, incrementarmos C de 1;
3. Toda vez que utilizarmos um temporário como operando, devemos decrementar C de 1.

Utilizando essa estratégia, o código de três endereços para a expressão  $x := -a + b * d$  seria:

```
$1 := -u a
$2 := b * d
$1 := $1 + $2
a := $1
```

Observe que ao utilizarmos \$1 e \$2 como operandos, em  $\$1 := \$1 + \$2$ , C foi decrementado duas vezes. Logo, o próximo temporário criado foi \$1.

## 6.2 Construção de Tabelas de Símbolos

Durante o processo de compilação, ao encontrarmos uma declaração de variável, devemos armazenar as informações relativas ao seu tipo na tabela de símbolos, juntamente com a quantidade de memória requisitada. Para programas formados por apenas um bloco, ou seja, monolíticos, a DDS da Figura 72 permite reconhecer e adicionar informações correspondentes a declaração de variáveis na tabela de símbolos. Nela observados a presença de offset, o qual corresponde a uma variável pública, necessária para o cálculo do próximo endereço de memória disponível. A função `InstTabSim(nome, tipo, offset)` insere as informações de nome, tipo e deslocamento da variável sendo declarada. O tipo `array(num, tipo)` especifica que a variável é do tipo vetor, contendo num elementos do tipo tipo. Já o tipo `apontador(tipo)` indica que a variável é um ponteiro para o tipo tipo. O atributo `length` especifica a quantidade de bytes requisitados pelo tipo.

De acordo com essa DDS, a compilação da declaração abaixo geraria a tabela de símbolos correspondente a Figura 71. Nela podemos observar o campo offset, o qual corresponde a quantos bytes a variável está deslocada a partir

Nome	Tipo	Offset
i	Inteiro	0
r	Real	4
a	array(5, inteiro)	12
p	apontador(inteiro)	32

Figura 71: Tabela de símbolos para as variáveis i, r, a e p.

do início do bloco de memória reservado às variáveis. Desta forma, se a memória destinada às variáveis inicia-se em A5F0<sub>16</sub>, então os endereços de memória para i, r, a e p, são, respectivamente: A5F0<sub>16</sub>, A5F4<sub>16</sub>, A5FC<sub>16</sub> e A610<sub>16</sub>.

```
i : integer;
r : real;
a : array[5] of integer;
p : ^integer;
```

Produções	Ações Semânticas
S → M D	
M → ε	offset := 0
D → D ; D	
D → id : T	InstTabSim(id.nome, T.tipo, offset) offset := offset + T.length
T → integer	T.tipo := inteiro T.length := 4
T → real	T.tipo := real T.length := 8
T → array[num] of T <sub>1</sub>	T.tipo := array(num.lexval, T <sub>1</sub> .tipo) T.length := num.lexval * T <sub>1</sub> .length
T → ^T <sub>1</sub>	T.tipo := apontador(T <sub>1</sub> .tipo) T.length := 4

Figura 72: DDS para inserir declarações de variáveis na tabela de símbolos.

A DDS definida anteriormente não permite a compilação de programas bloco estruturados. Para compilarmos programas deste tipo é necessário controlarmos o escopo das informações. A Figura 73 apresenta uma DDS que estende a definição da Figura 72 para tratar a declaração aninhada de procedimentos. Nela o escopo é controlado através da criação de uma nova tabela de símbolos na entrada do procedimento sendo compilado. O aninhamento é controlado pelo uso de duas pilhas LIFO. A primeira, denominada por pTabSim, armazena ponteiros para as tabelas de símbolos criadas. A segunda, denominada pOffset, contém o próximo endereço de memória disponível, relativo à área de dados do procedimento. As seguintes funções são necessárias:

1. **Topo(ptr)**: retorna o valor contido no topo da pilha apontada por ptr, sem removê-lo;
2. **Push(val, ptr)**: empilha um literal ou um ponteiro, armazenado em val, na pilha apontada por ptr;
3. **Pop(ptr)**: desempilha o valor contido no topo da pilha apontada por ptr;
4. **CriaTabSim(ptr)**: cria uma nova tabela de símbolos, filha daquela apontada por ptr, retornando o ponteiro para a mesma;
5. **SetLength(ptr, length)**: armazena, na tabela de símbolos apontada por ptr, o comprimento length total da área de dados local do procedimento

correspondente;

6. **AddProc(ptr, nome, newptr)**: insere, na tabela de símbolos apontada por `ptr`, o nome do procedimento e o ponteiro `newptr` para a tabela de símbolos deste procedimento;
7. **AddSimb(ptr, nome, tipo, offset)**: adiciona, na tabela de símbolos apontada por `ptr`, o novo símbolo `nome` do tipo `tipo` e deslocamento `offset`.

Produções	Ações Semânticas
$S \rightarrow M\ D$	<code>SetLength(Topo(pTabSim), Topo(pOffset))</code> <code>Pop(pTabSim)</code> <code>Pop(pOffset)</code>
$M \rightarrow \epsilon$	<code>New := CriaTabSim(nil)</code> <code>Push(New, pTabSim)</code> <code>Push(0, pOffset)</code>
$D \rightarrow D ; D$	
$D \rightarrow id : T$	<code>AddSimb(Topo(pTabSim), id.nome, T.tipo, Topo(pOffset))</code> <code>Topo(pOffset) := Topo(pOffset) + T.length</code>
$D \rightarrow procedure\ id ; N\ D ; S$	<code>Tmp := Topo(pTabSim)</code> <code>SetLength(Tmp, Topo(pOffset))</code> <code>Pop(pTabSim)</code> <code>Pop(pOffset)</code> <code>AddProc(Topo(pTabSim), id.nome, Tmp)</code>
$N \rightarrow \epsilon$	<code>New := CriaTabSim(Topo(pTabSim))</code> <code>Push(New, pTabSim)</code> <code>Push(0, pOffset)</code>
$T \rightarrow integer$	<code>T.tipo := inteiro</code> <code>T.length := 4</code>
$T \rightarrow real$	<code>T.tipo := real</code> <code>T.length := 8</code>
$T \rightarrow array[num] of T_1$	<code>T.tipo := array(num.lexval, T1.tipo)</code> <code>T.length := num.lexval * T1.length</code>
$T \rightarrow ^T_1$	<code>T.tipo := apontador(T1.tipo)</code> <code>T.length := 4</code>

Figura 73: DDS para a criação e o gerenciamento de tabelas de símbolos, respeitando-se o escopo.

Observe pelo esquema de tradução da figura que na produção:

$$D \rightarrow \text{procedure}\ id ; N\ D ; S$$

o não terminal `S` corresponde a regra responsável pela geração da sequência de comandos correspondentes ao corpo do procedimento, a qual não foi definida por questões de economia de espaço. A Figura 74 ilustra as tabelas de símbolos criadas durante a compilação do programa bloco estruturado abaixo, no qual as marcas “...” indicam o código gerado durante a avaliação da produção `S`:

```

n : integer;
e : real;
procedure Exp;
  r,a,b : real;
  procedure Mult;
    r, x, y : integer;
    procedure Add;
      r : real;
      a, b : integer;
      ...
      ...
      ...
procedure Show;
  h : integer;
  m : array[10] of integer;
  p : ^array[10] of integer;
  ...
  ...
  ...

```

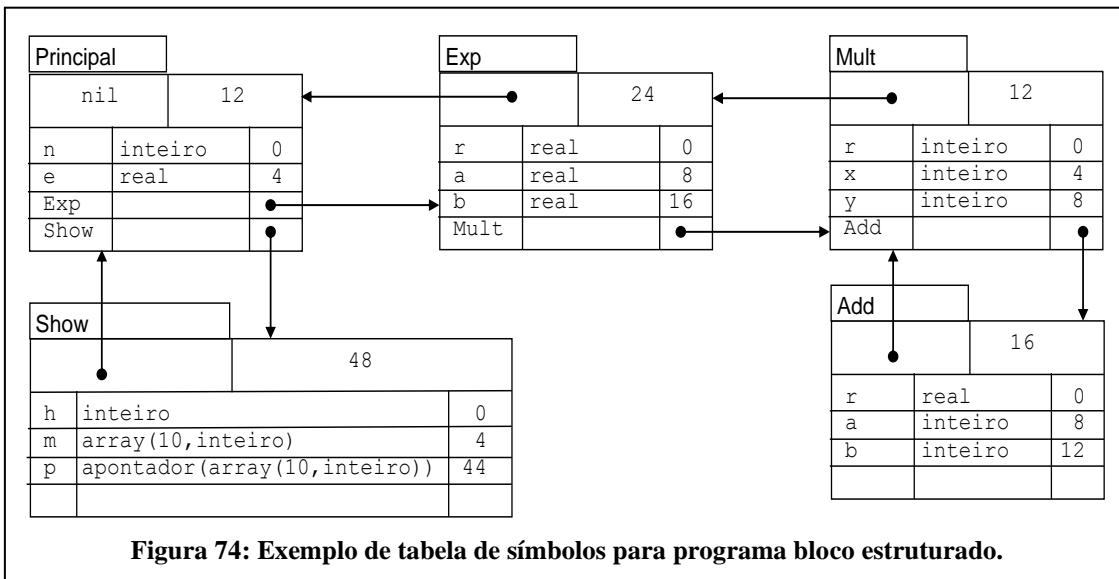


Figura 74: Exemplo de tabela de símbolos para programa bloco estruturado.

Podemos ainda modificar a DDS da Figura 73 para permitir a declaração do tipo registro, além daqueles já definidos. Isto é feito através da introdução da DDS indicada pela Figura 75.

Observe que na declaração de um registro, uma nova tabela de símbolos é criada para os nomes dos campos, sendo seu ponteiro empilhado em `pTabSim`. Desta forma, ao executarmos a ação associada a

$$D \rightarrow id : T$$

o ponteiro associado ao registro, o qual é dado pelo atributo `T.tipo`, é instalado na tabela de símbolos corrente, a qual é apontada por `Topo(pTabSim)`, na entrada criada para `id.nome`. Esta abordagem permite limitar o escopo dos nomes de campos ao registro declarado.

T → record L D end	T.tipo := registro(Topo(pTabSim)) T.length := Topo(pOffset) Pop(pTabSim) Pop(pOffset)
L → ε	t := CriaTabSim(nil) Push(t, pTabSim) Push(0, pOffset)

Figura 75: DDS para a criação e o gerenciamento do tipo registro.

Apesar da produção na Figura 75 permitir a declaração de procedimentos dentro de registros, este fato é desprezado. A Figura 76 ilustra as tabelas de símbolos criadas para a declaração:

```
a : integer;
b : real;
r : record a : integer; b : real end ;
s : record a : ^integer ; b : ^real end;
...

```

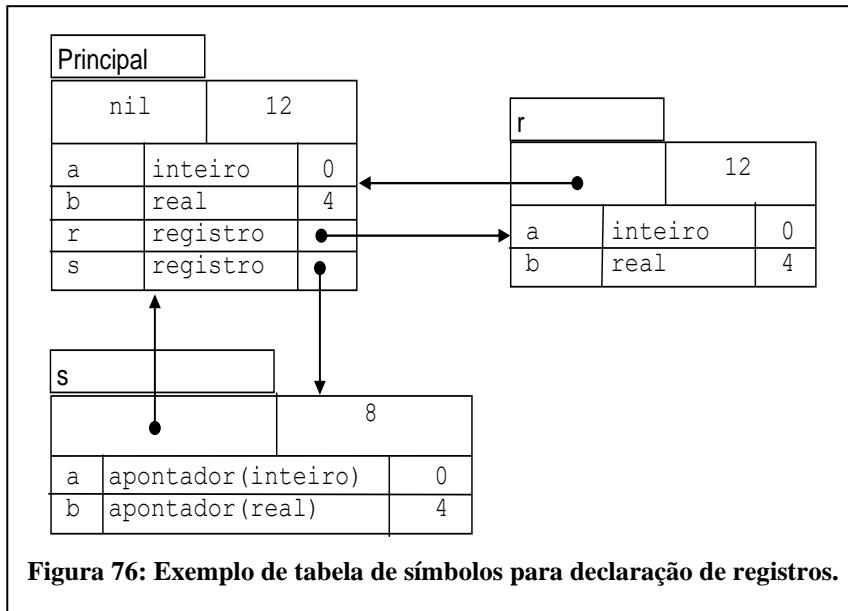


Figura 76: Exemplo de tabela de símbolos para declaração de registros.

## 6.3 Compilação de Atribuições

Em uma atribuição, a expressão pode ter diferentes tipos, como inteiro, real, apontador, array, etc.. Desta forma, acessar a região de memória apontada pelos operandos, verificar a compatibilidade de tipos, efetuar a conversão de tipos, quando necessária, e armazenar o resultado na locação especificada, são tarefas necessárias .

## 6.3.1 Atribuições

Em uma atribuição devemos identificar dois elementos:

```
valor-L := valor-R
```

O `valor-L` corresponde ao endereço de memória onde o conteúdo do `valor-R` será armazenado. Note portanto que `valor-R` corresponde a um conteúdo. Desta forma, se considerarmos a atribuição `x := a`, o `valor-L` corresponde ao endereço de `x`, enquanto o `valor-R` corresponde ao valor armazenado no endereço de memória referenciado por `a`. Já na atribuição `^p := a + 5`, o `valor-L` corresponde a um endereço de memória, o qual se encontra armazenado no endereço de memória de `p`, enquanto o `valor-R` é o resultado da avaliação da expressão `a + 5`.

A Figura 77 especifica uma DDS para a compilação de atribuições. Note a presença da função `LookUp(nome)`, a qual verifica se um `nome` está na tabela de símbolos corrente, ou seja, aquela apontada por `Topo(pTabSim)`, devolvendo um ponteiro para o mesmo. Caso o nome não seja encontrado, então a tabela de símbolos do procedimento envolvente deve ser pesquisada. Caso todas as tabelas apontadas tenham sido pesquisadas e o nome não seja encontrado em nenhuma delas, então `LookUp()` retorna `nil`. Além disso, deve ser observado também a presença da função `Erro()`, a qual efetua notifica e efetua o tratamento do erro ocorrido.

Dois comentários ainda são necessários. A função `GeraCode()` agora concatena e grava seus argumentos em um arquivo de saída, logo a gramática a seguir não é de atributos. Além disso, visto que estamos lidando com ponteiros, a função `GeraTemporario()` foi adaptada para gerar um novo temporário e instalá-lo na tabela de símbolos, retornando um ponteiro para o mesmo.

Produções	Ações Semânticas
$S \rightarrow id := E$	$p := \text{LoopUp}(id.nome)$ Se $p <> \text{nil}$ then $\text{GeraCode}(p ':= E.ptr)$ Senão $\text{Erro}()$ Fim se
$E \rightarrow - E_1$	$E.ptr := \text{GeraTemporario}()$ $\text{GeraCode}(E.ptr ':= '- u' E_1.ptr)$
$E \rightarrow E_1 + E_2$	$E.ptr := \text{GeraTemporario}()$ $\text{GeraCode}(E.ptr ':= E_1.ptr '+' E_2.ptr)$
$E \rightarrow E_1 * E_2$	$E.ptr := \text{GeraTemporario}()$ $\text{GeraCode}(E.ptr ':= E_1.ptr '*' E_2.ptr)$
$E \rightarrow ( E_1 )$	$E.ptr := E_1.ptr$
$E \rightarrow id$	$P := \text{LoopkUp}(id.nome)$ Se $p <> \text{nil}$ então $E.ptr := p$ Senão $\text{Erro}()$ Fim se

Figura 77: DDS para tratamento de comandos de atribuição.

## 6.3.2 Verificação e Conversão de Tipos

Em uma linguagem tipada, é necessário efetuarmos a verificação e a conversão, quando possível, entre tipos diferentes. Por exemplo, para expressões aritméticas, o tipo da operação é determinado pelos tipos dos operandos. Para exemplificar este fato, é Figura 78 ilustrada uma DDS para operações de adição. As demais operações podem ser deduzidas da mesma forma. Na Figura 78 podemos observar um novo tipo, o tipoerro, que indica a ocorrência de um erro na conversão de tipos.

Assim, para o comando de atribuição:  $X := A + B * C$ , seria gerado o seguinte código de três endereços, considerando que  $X$  e  $A$  são do tipo real, e  $B$  e  $C$  são do tipo inteiro:

```
$1 := B *int C
$3 := inttoreal $1
$2 := A +real $3
X := $2
```

Pelo exemplo observamos os comandos `*int`, que realiza a multiplicação inteira, `inttoreal`, que efetua a conversão de um valor inteiro para real e `+real`, que realiza a adição entre reais.

Produções	Ações Semânticas
$E \rightarrow E_1 + E_2$	<pre> E.nome := GeraTemporario() Se (E<sub>1</sub>.tipo = inteiro) e (E<sub>2</sub>.tipo = inteiro) então     GeraCode(E.nome ':=&gt; E<sub>1</sub>.nome '+int' E<sub>2</sub>.nome)     E.tipo := inteiro Senão     Se (E<sub>1</sub>.tipo = real) e (E<sub>2</sub>.tipo = real) então         GeraCode(E.nome ':=&gt; E<sub>1</sub>.nome '+real' E<sub>2</sub>.nome)         E.tipo := real     Senão         Se (E<sub>1</sub>.tipo = inteiro) e (E<sub>2</sub>.tipo = real) então             t := GeraTemporario()             GeraCode(t ':=&gt; inttoreal' E<sub>1</sub>.nome)             GeraCode(E.nome ':=&gt; t '+real' E<sub>2</sub>.nome)             E.tipo := real         Senão             Se (E<sub>1</sub>.tipo = real) e (E<sub>2</sub>.tipo = inteiro) então                 t := GeraTemporario()                 GeraCode(t ':=&gt; inttoreal' E<sub>2</sub>.nome)                 GeraCode(E.nome ':=&gt; E<sub>1</sub>.nome '+real' t)                 E.tipo := real             Senão                 E.tipo := tipoerro         Fim se     </pre>

Figura 78: DDS para verificação e conversão de tipos em expressões aritméticas.

### 6.3.3 Endereçando Elementos de Matrizes

Quando um vetor é declarado, os seus elementos podem ser rapidamente acessados se forem armazenados em um bloco de localizações consecutivas na memória. Assim, se considerarmos que cada elemento do vetor tem uma largura  $w$ , que o primeiro e último elemento do vetor são definidos por  $LInf$  e  $LSup$  (Limite Inferior e Superior do vetor), então o  $i$ -ésimo elemento do vetor  $A$  está localizado na posição de memória

$$\text{Base} + (i - LInf) * w$$

onde  $\text{Base}$  é o endereço relativo de memória alocada para  $A$ , ou seja, trata-se do endereço do elemento  $A[LInf]$ .

A expressão acima pode ser avaliada em tempo de compilação se for traduzida para

$$i * w + (\text{Base} - LInf * w)$$

Nesta expressão, o termo  $i$  é conhecido somente em tempo de execução. Entretanto, o termo  $c = \text{Base} - LInf * w$  pode ser calculado na declaração do vetor. Assim, se salvarmos  $c$  na tabela de símbolos, juntamente com a declaração de  $A$ , para obtermos o elemento  $A[i]$  é só calcularmos  $i * w + c$ .

Assim, se considerarmos a declaração do vetor

```
A : array[3] of integer;
```

e considerando que  $A$  seja alocado a partir do endereço relativo de memória 12, então os elementos  $A[1]$ ,  $A[2]$  e  $A[3]$  são alocados a partir nos endereços 12, 16 e 20, respectivamente, pois  $c = 12 - 1 * 4 = 8$ , e logo:

$$\text{endereço de } A[1] = 1 * 4 + 8 = 12$$

$$\text{endereço de } A[2] = 2 * 4 + 8 = 16$$

$$\text{endereço de } A[3] = 3 * 4 + 8 = 20$$

O mesmo raciocínio acima pode ser adotado para matrizes bidimensionais. Entretanto, neste caso, primeiramente devemos decidir se os elementos serão armazenados em linha ou em coluna. A Figura 79 ilustra os dois tipos de armazenamento. O Pascal, por exemplo, utiliza a organização em linha, já o Fortran utiliza a organização em coluna.

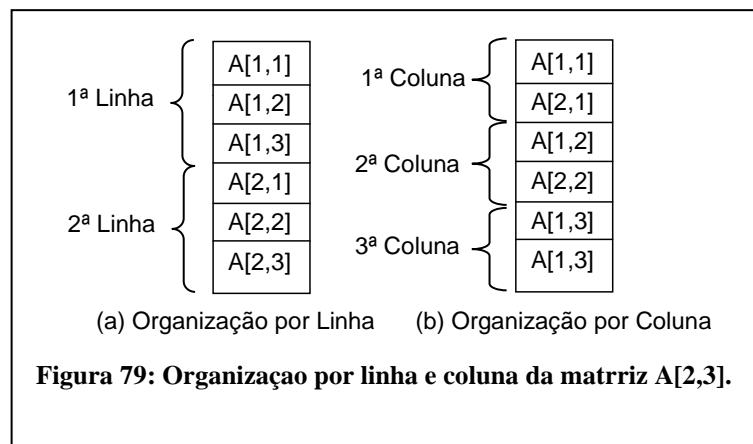


Figura 79: Organização por linha e coluna da matriz  $A[2,3]$ .

No caso de uma matriz bidimensional organizada em linha, o endereço relativo ao elemento  $A[i_1, i_2]$  pode ser calculado por

$$\text{Base} + ((i_1 - LInf_1) * n_2 + i_2 - LInf_2) * w$$

onde  $LInf_1$  e  $LInf_2$  são os limites inferiores aos quais  $i_1$  e  $i_2$  estão subordinados, e  $n_2$  é o número de valores que  $i_2$  pode assumir, ou seja, é o número de elementos da segunda dimensão, o qual pode ser obtido pela expressão  $n_2 = LSup_2 - LInf_2 + 1$ .

Novamente, considerando que os únicos valores desconhecidos em tempo de compilação são  $i_1$  e  $i_2$ , então podemos reescrever a expressão acima como

$$((i_1 * n_2) + i_2) * w + (\text{Base} - ((LInf_1 * n_2) + LInf_2) * w)$$

Novamente, a expressão  $c = w + (\text{Base} - ((LInf_1 * n_2) + LInf_2) * w)$  pode ser calculada em tempo de compilação, sendo armazenada na tabela de símbolos correspondente a declaração de A. Assim, nossa expressão se reduz a

$$((i_1 * n_2) + i_2) * w + c$$

Podemos generalizar esta expressão para a declaração de matrizes de várias dimensões. A expressão abaixo permite uma generalização para o cálculo do endereço relativo de  $A[i_1, \dots, i_k]$

$$((\dots((i_1 * n_2 + i_2) * n_3 + i_3)\dots) * n_k + i_k) * w + c$$

onde,  $c = \text{Base} - ((\dots((LInf_1 * n_2 + LInf_2) * n_3 + LInf_3)\dots) * n_k + LInf_k) * w$ . Visto que para todo  $j$ ,  $n_j = LSup_j - LInf_j + 1$  é fixo, então o valor de  $c$  pode ser calculado em tempo de compilação e salvo na tabela de símbolos na entrada para A.

Para a geração de código para referência a elementos de uma matriz, considere a gramática abaixo.

```

Var → id [ Lista ]
Var → id
Lista → Lista , E
Lista → E
  
```

Esta gramática utiliza atributos herdados para a passagem do ponteiro de **id**. Poderíamos reescrevê-la para utilizar apenas atributos sintetizados, conforme indicado a seguir.

```

Var → Lista ]
Var → id
Lista → Lista , E
Lista → id [ E
  
```

Uma Lista que produza os  $m$  primeiros índices de uma referência a um array  $k$ -dimensional  $A[i_1, i_2, \dots, i_k]$  irá gerar o código de três endereços para calcular

$$((\dots((i_1 * n_2 + i_2) * n_3 + i_3)\dots) * n_k + i_k) * w$$

utilizando a fórmula de recorrência:

$$e_1 = i_1$$

$$e_m = e_{m-1} * n_m + i_m, \text{ para } m = 2, 3, \dots, k$$

A DDS da Figura 80 permite gerar código de três endereços para atribuições que façam referência a elementos de matrizes. Nela, percebemos o uso das funções `c(Lista.array)`, a qual retorna o valor de `c` armazenado na tabela de símbolos para o array `Lista.array`; `w(Lista.array)`, a qual retorna a largura `w` do array dado por `Lista.array`; e `Limite(Lista1.array, nDim)`, a qual retorna o número de elementos da dimensão `nDim` do array dado por `Listा1.array`.

Produções	Ações Semânticas
$S \rightarrow \text{Var} := E$	Se <code>Var.offset</code> = null Então <code>GeraCode(Var.nome ':= E.nome)</code> Senão <code>GeraCode(Var.nome '[' Var.offset ']' ':= E.nome)</code> Fim Se
$E \rightarrow E_1 + E_2$	<code>E.nome := GeraTemporario()</code> <code>GeraCode(E.nome ':= E<sub>1</sub>.nome +' E<sub>2</sub>.nome)</code>
$E \rightarrow ( E_1 )$	<code>E.nome := E<sub>1</sub>.nome</code>
$E \rightarrow \text{Var}$	Se <code>Var.offset</code> = null Então <code>E.nome := Var.nome</code> Senão <code>E.nome := GeraTemporario()</code> <code>GeraCode(E.nome ':= Var.nome '[' Var.offset ''])</code> Fim Se
$\text{Var} \rightarrow \text{Lista} ]$	<code>Var.nome := GeraTemporario()</code> <code>Var.offset := GeraTemporario()</code> <code>GeraCode(Var.nome ':= c(Lista.array))</code> <code>GeraCode(Var.offset ':= w(Lista.array) * Lista.nome)</code>
$\text{Var} \rightarrow \text{id}$	<code>Var.nome := id.nome</code> <code>Var.offset := null</code>
$\text{Lista} \rightarrow \text{Lista}_1 , E$	<code>Tmp := GeraTemporario()</code> <code>nDim := Lista.nDim + 1</code> <code>GeraCode(Tmp ':= Lista<sub>1</sub>.nome '*' Limite(Lista<sub>1</sub>.array, nDim))</code> <code>GeraCode(Tmp ':= Tmp +' E.nome)</code> <code>Lista.array := Lista<sub>1</sub>.array</code> <code>Lista.nome := Tmp</code> <code>Lista.nDim := nDim</code>
$\text{Lista} \rightarrow \text{id} [ E$	<code>Lista.array := id.nome</code> <code>Lista.nome := E.nome</code> <code>Lista.nDim := 1</code>

Figura 80: DDS para geração de código de três endereços para atribuições envolvendo arrays.

Para exemplificar, considere o array de inteiros `A[1..10, 1..20]`. Claramente, `linf1 = linf2 = 1`, `n1 = 10`, `n2 = 20` e `w = 4`. Uma árvore gramatical anotada para a atribuição `x := A[y, z]` é mostrada pela figura. A atribuição é traduzida para a seguinte sequência de instruções de três endereços:

```

$1 := y * 20
$1 := $1 + z
$2 := c           // Constante: c = baseA - 84
$3 := 4 * $1
$4 := $2 [ $3 ]
x := $4

```

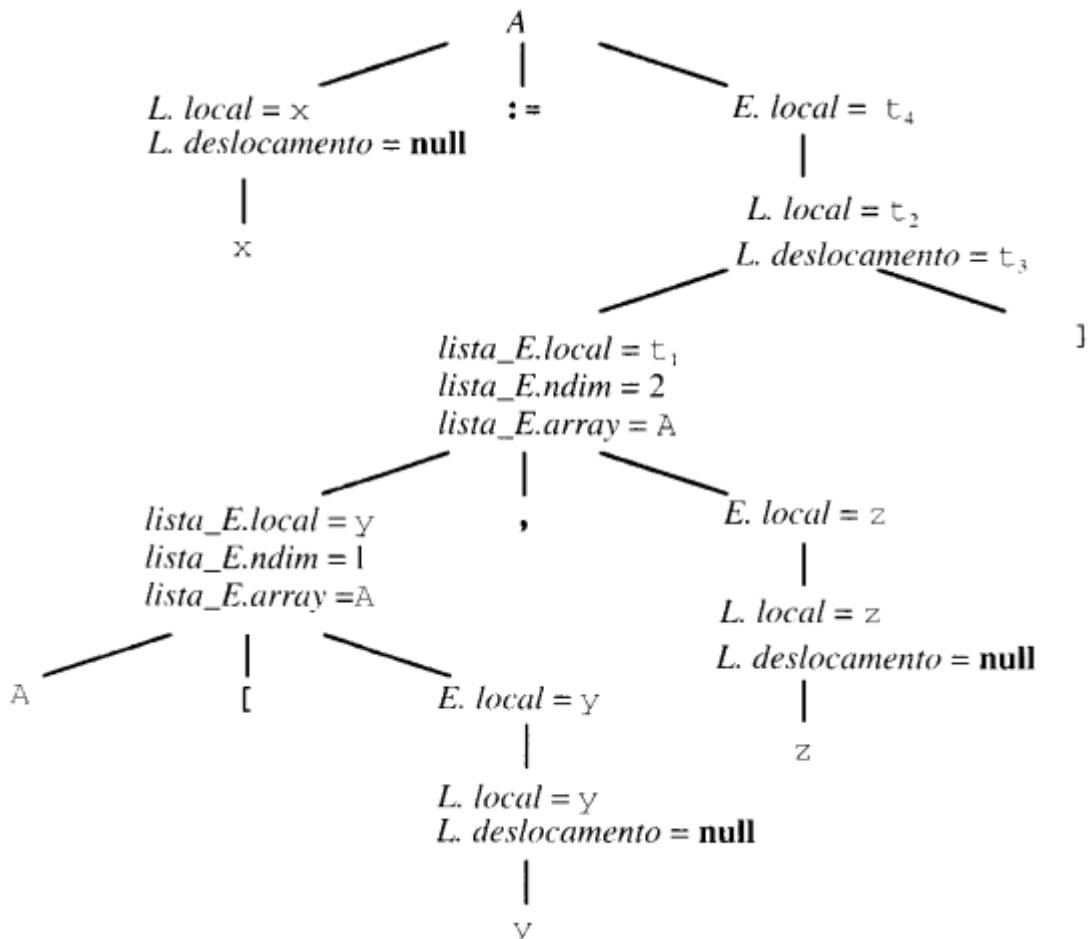


Figura 81: Árvore gramatical para `X := A[y, z]`.

FIGURA 8.18 DO AHO (página 210), substituindo os atributos para nome e offset no lugar de local e deslocamento; e L e Lista\_E por Var e Lista.

## 6.4 Expressões Lógicas e Comandos de Controle

Expressões lógicas são usadas como expressões condicionais de comandos de controle (*if*, *while*, etc.) e em comandos de atribuição lógica. Nesta seção serão apresentados dois métodos de tradução de expressões lógicas:

### 1. Representação numérica:

Este método codifica numericamente as constantes **true** e **false** (por exemplo, 1 para **true** e 0 para **false**) e avalia as expressões lógicas de forma numérica, ficando o resultado da avaliação em uma variável temporária.

### 2. Representação por fluxo de controle:

Este método traduz expressões lógicas para instruções *if* e *goto*, as quais desviam a execução do programa para pontos distintos, caso o resultado da avaliação seja verdadeiro (**true**) ou falso (**false**). Para tanto, são utilizados os atributos **E.true** e **E.false**, os quais conterão rótulos para onde a execução deve ser transferida em caso da avaliação ser verdadeira ou falsa, respectivamente.

### 6.4.1 Representação Numérica

Neste método, as operações lógicas *and*, *or* e *not* são avaliadas numericamente para 1, quando resultarem em verdadeiro (**true**), ou 0, quando resultarem em falso (**false**). A Figura 82 apresenta a DDS capaz de gerar código para expressões lógicas, considerando que estas instruções serão armazenadas em um vetor de quádruplas. Nele, a função *GeraCode()* utiliza a variável *proxQuad* para indicar o índice da próxima quádrupla disponível.. Após armazenar a quádrupla , a função *GeraCode()* incrementa a variável *proxQuad*.

$E \rightarrow E_1 \text{ or } E_2$	<code>E.nome = GeraTemporario() GeraCode(E.nome ':= E<sub>1</sub>.nome 'or' E<sub>2</sub>.nome)</code>
$E \rightarrow E_1 \text{ and } E_2$	<code>E.nome = GeraTemporario() GeraCode(E.nome ':= E<sub>1</sub>.nome 'and' E<sub>2</sub>.nome)</code>
$E \rightarrow \text{not } E_1$	<code>E.nome = GeraTemporario() GeraCode(E.nome ':= 'not' E<sub>1</sub>.nome)</code>
$E \rightarrow ( E_1 )$	<code>E.nome = E<sub>1</sub>.nome</code>
$E \rightarrow id_1 \text{ oprel } id_2$	<code>E.nome = GeraTemporario() GeraCode('if' id<sub>1</sub>.nome oprel.simbolo id<sub>2</sub>.nome 'goto' ProxQuad + 3) GeraCode(E.nome ':= 0') GeraCode('goto' ProxQuad + 2) GeraCode(E.nome ':= 1')</code>

$E \rightarrow \text{true}$	<code>E.nome = GeraTemporario() GeraCode(E.nome ':= 1')</code>
$E \rightarrow \text{false}$	<code>E.nome = GeraTemporario() GeraCode(E.nome ':= 0')</code>

Figura 82: DDS para avaliação numérica de expressões lógicas.

Observe que o atributo `E.nome` contém o nome do temporário que armazena o resultado da avaliação da expressão `E`. O código gerado, o qual avalia a expressão, é colocado no vetor de quádruplas.

Considerando que as variáveis `A`, `B` e `C` são variáveis lógicas e que o código gerado seria armazenado a partir da quádrupla número 100, a avaliação da expressão “`A or B and not C`” seria traduzida para:

```
100: $1 := not C
101: $2 := B and $1
102: $3 := A or $2
```

A expressão relacional `A < B` (onde `A` e `B` são variáveis lógicas) seria traduzida para:

```
100: if A < B goto 103
101: $1 := 0
102: goto 104
103: $1 := 1
```

Observe que o resultado da avaliação da expressão fica sempre no último temporário gerado no processo. O nome desse temporário é armazenado no atributo `E.nome` que representa a expressão e o código gerado é armazenado no vetor de quádruplas.

Como outro exemplo, a avaliação da expressão relacional `A < B or C < D and E < F` (onde `A`, `B`, `C`, `D`, `E` e `F` são variáveis lógicas) geraria as seguintes instruções:

```
100: if A < B goto 103
101: $1 := 0
102: goto 104
103: $1 := 1
```

```

104: if C < D goto 107
105: $2 := 0
106: goto 108
107: $2 := 1
108: if E < F goto 111
109: $3 := 0
110: goto 112
111: $3 := 1
112: $4 := $2 and $3
113: $5 := $1 or $4

```

Note que a gramática da Figura 82 não é de atributos, já que ela gera efeitos colaterais gravando o código no vetor de quádruplas. Para torná-la de atributos, basta modificarmos a função `GeraCode()` para devolver uma *string* contendo o código gerado e armazenarmos este código em um novo atributo, que neste caso chamaremos de `code`. O exemplo a seguir ilustra como realizar esta tarefa.

$E \rightarrow E_1 \text{ or } E_2$	$E.\text{nome} = \text{GeraTemporario}()$ $E.\text{code} := E_1.\text{code}    E_2.\text{code}   $ $\quad \text{GeraCode}(E.\text{nome} ':=', E_1.\text{nome}, 'or', E_2.\text{nome})$
-------------------------------------	--

Utilizando a gramática de atributos construída conforme exemplificado anteriormente, podemos estendê-la para incorporar ações semânticas para gerar o código intermediário para comandos do tipo `while-do`, conforme ilustrado pelo diagrama da Figura 83. O código gerado é uma *string* que instruções armazenada no atributo `code` de `S`. Os atributos `inicio` e `proximo` identificam, respectivamente, o início da iteração e o início do comando seguinte ao `while`, enquanto que a função `GeraRotulo()` retorna um novo rótulo simbólico a cada vez que for chamada.

$S \rightarrow \text{while } E \text{ do } S_1$	$S.\text{inicio} = \text{GeraRotulo}()$ $S.\text{proximo} = \text{GeraRotulo}()$ $S.\text{code} := \text{GeraCode}(S.\text{inicio} ':')    E.\text{code}   $ $\quad \text{GeraCode}('if' E.\text{nome} '= 0' \text{ goto}' S.\text{proximo})   $ $\quad S_1.\text{code}    \text{GeraCode}('goto' S.\text{inicio})   $ $\quad \text{GeraCode}(S.\text{proximo} ':')$
---	---

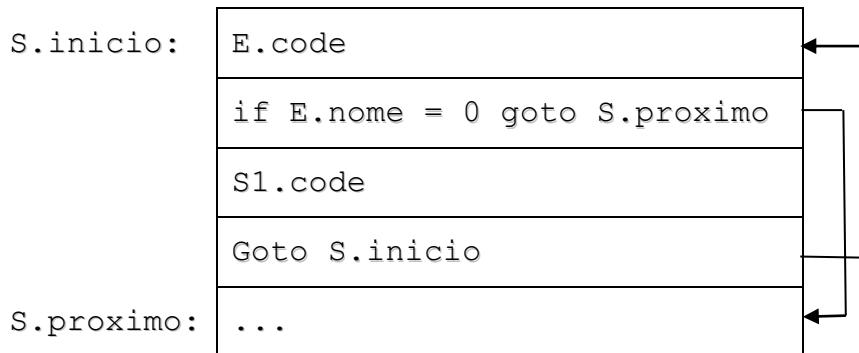


Figura 83: Diagrama do comando while-do.

A avaliação numérica de expressões lógicas adapta-se bem a esquemas *S-atribuídos* (*bottom-up*), pois são usados apenas atributos sintetizados. A avaliação por fluxo de controle, descrita a seguir, adapta-se a esquemas *L-atribuídos*, pois usa atributos herdados e sintetizados. A vantagem da avaliação por fluxo de controle é que o código gerado é mais eficiente.

#### 6.4.2 Representação por Fluxo de Controle

Considera-se a seguir a tradução de expressões lógicas em código de três endereços no contexto de enunciados if-then, if-then-else e while-do, tais como aqueles gerados pela seguinte gramática:

$$\begin{aligned}
 S \rightarrow & \text{ if } E \text{ then } S_1 \\
 | & \text{ if } E \text{ then } S_1 \text{ else } S_2 \\
 | & \text{ while } E \text{ do } S_1
 \end{aligned}$$

Em cada uma dessas produções,  $E$  é uma expressão lógica a ser traduzida. Na tradução, assume-se que um enunciado de três endereços possa ser simbolicamente rotulado.

A uma expressão lógica  $E$  é associado dois rótulos:  $E.\text{True}$ , o rótulo para o qual o controle flui se  $E$  for verdadeiro e  $E.\text{False}$ , o rótulo para o qual flui se  $E$  for falso. As regras semânticas para traduzir um enunciado de fluxo de controle  $S$  permitem que o controle flua da tradução  $S.\text{code}$  para a instrução de três endereços que se segue imediatamente a  $S.\text{code}$ . Em alguns casos, a instrução que se segue a  $S.\text{code}$  é um desvio para algum rótulo  $L$ . Um desvio para  $L$ , a partir de dentro de  $S.\text{code}$ , é evitado utilizando-se um atributo herdado  $S.\text{proximo}$ . O valor de  $S.\text{proximo}$  é um rótulo que é atrelado à primeira instrução de três endereços a ser executada após o código de  $S$ . A inicialização de  $S.\text{proximo}$  não é mostrada.

Ao se traduzir o enunciado condicional  $S \rightarrow \text{if } E \text{ then } S_1$ , um novo rótulo  $E.\text{True}$  é criado e atrelado à primeira instrução de três endereços gerada para o enunciado  $S_1$ . A definição dirigida pela sintaxe aparece na Figura 84. O código para  $E$  gera um desvio para  $E.\text{True}$ , se  $E$  for verdadeiro, e um salto para  $S.\text{proximo}$ . Consequentemente, fazemos  $E.\text{False}$  igual a  $S.\text{proximo}$ .

$S \rightarrow \text{if } E \text{ then } S_1$	$\begin{aligned} E.\text{True} &= \text{GeraRotulo}() \\ E.\text{False} &:= S.\text{proximo} \\ S_1.\text{proximo} &:= S.\text{proximo} \\ S.\text{code} &:= E.\text{code} \mid\mid \text{GeraCode}(E.\text{True} ':') \mid\mid S_1.\text{code} \end{aligned}$
--	--

Figura 84: DDS para avaliação do if-then por fluxo de controle.

Na tradução do enunciado condicional  $S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$ , o código para a expressão lógica  $E$  possui desvios para fora do mesmo, para a primeira instrução do código de  $S_1$ , se  $E$  for verdadeira, e para a primeira instrução do código  $S_2$ , se  $E$  for falsa. Da mesma forma que para o enunciado if-then, o atributo herdado  $S.\text{proximo}$  fornece o próximo rótulo da instrução de três endereços a ser executada em seguida à execução do código para  $S$ . Um `goto` (desvio) explícito para  $S.\text{proximo}$  aparece após o código para  $S_1$ , mas não após  $S_2$ .

$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$	$\begin{aligned} E.\text{True} &= \text{GeraRotulo}() \\ E.\text{False} &:= \text{GeraRotulo}() \\ S_1.\text{proximo} &:= S.\text{proximo} \\ S_2.\text{proximo} &:= S.\text{proximo} \\ S.\text{code} &:= E.\text{code} \mid\mid \text{GeraCode}(E.\text{True} ':') \mid\mid S_1.\text{code} \mid\mid \\ &\quad \text{GeraCode('goto' } S.\text{proximo}) \mid\mid \\ &\quad \text{GeraCode}(E.\text{False} ':') \mid\mid S_2.\text{code} \end{aligned}$
--	---

Figura 85: DDS para avaliação do if-then-else por fluxo de controle.

No código para  $S \rightarrow \text{while } E \text{ do } S_1$  um novo rótulo  $S.\text{inicio}$  é criado e atrelado a primeira instrução gerada para  $E$ . Um outro novo rótulo,  $E.\text{True}$ , é atrelado a primeira instrução de  $S_1$ . O código para  $E$  gera um desvio para esse rótulo se  $E$  for verdadeiro e um desvio para  $E.\text{False}$  se  $E$  for falso; de novo, fazemos  $E.\text{False}$  igual a  $S.\text{proximo}$ . Após a instrução para  $S_1$  colocamos a instrução `goto`  $S.\text{inicio}$ , que provoca um desvio de volta para o início do código para a expressão lógica  $E$ . Note que  $S_1.\text{proximo}$  é estabelecido para esse rótulo  $S.\text{inicio}$ , de forma que os desvios de dentro de  $S_1.\text{code}$  podem se dirigir diretamente para  $S.\text{inicio}$ .

$S \rightarrow \text{while } E \text{ do } S_1$	$\begin{aligned} S.\text{inicio} &:= \text{GeraRotulo}() \\ E.\text{True} &= \text{GeraRotulo}() \\ E.\text{False} &:= S.\text{proximo} \\ S_1.\text{proximo} &:= S.\text{inicio} \end{aligned}$
---	--

	<pre>S.code := GeraCode(S.inicio `:`)    E.code    GeraCode(E.True `:`)    S1.code    GeraCode(`goto' S.inicio)</pre>
--	---

Figura 86: DDS para avaliação do while-do por fluxo de controle.

Discute-se agora a geração do  $E.code$ , o código produzido para expressões lógicas  $E$  apresentadas pela Figura 84, Figura 85 e Figura 86. Como citado anteriormente,  $E$  é traduzido em uma sequência de instruções de três endereços que avalia  $E$  como uma sequência de desvios condicionais e incondicionais para uma ou duas localizações:  $E.True$ , localização que o controle deve desviar se  $E$  for verdadeiro e  $E.False$ , local que deve ser atingido caso  $E$  seja falso.

A ideia básica por traz da tradução é a seguinte. Suponha que  $E$  seja da forma  $a < b$ . O código gerado é, consequentemente, da forma:

```
if a < b then goto E.True
goto E.False
```

Suponha que  $E$  seja da forma  $E_1 \text{ or } E_2$ . Se  $E_1$  for verdadeiro, sabemos imediatamente que  $E$  é verdadeiro, não necessitando avaliar  $E_2$ , de modo que  $E_1.True$  é o mesmo que  $E.True$ . Se  $E_1$  for falso, então  $E_2$  precisa ser avaliado, e, dessa forma, fazemos  $E_1.False$  ser o rótulo do primeiro enunciado no código para  $E_2$ . As saídas verdadeira e falsa de  $E_2$  podem ser feitas iguais as saídas verdadeira e falsa de  $E$ , respectivamente.

Considerações análogas se aplicam à tradução de  $E_1 \text{ and } E_2$ . Nenhum código da forma  $\text{not } E_1$  é necessitado para a expressão  $E$ ; simplesmente intercambiamos as saídas falsas de  $E_1$  para obter as saídas verdadeira e falsa de  $E$ . Uma definição dirigida pela sintaxe que gera código de três endereços para expressões lógicas desta forma é mostrada pela figura. Note-se que os atributos `True` e `False` são herdados.

$E \rightarrow E_1 \text{ or } E_2$	$E_1.True := E.True$ $E_1.False := \text{GeraRotulo}()$ $E_2.True := E.True$ $E_2.False := E.False$ $E.code := E_1.code    \text{GeraCode}(E_1.False `:`)    E_2.code$
$E \rightarrow E_1 \text{ and } E_2$	$E_1.True := \text{GeraRotulo}()$ $E_1.False := E.False$ $E_2.True := E.True$ $E_2.False := E.False$ $E.code := E_1.code    \text{GeraCode}(E_1.True `:`)    E_2.code$
$E \rightarrow \text{not } E_1$	$E_1.True := E.False$ $E_1.False := E.True$

	E.code := E <sub>1</sub> .code
E → ( E <sub>1</sub> )	E <sub>1</sub> .True := E.True E <sub>1</sub> .False := E.False E.code := E <sub>1</sub> .code
E → id <sub>1</sub> oprel id <sub>2</sub>	E.code := GeraCode('if' id <sub>1</sub> .nome relop.simbolo id <sub>2</sub> .nome 'goto' E.True)    GeraCode('goto' E.False)
E → true	E.code := GeraCode('goto' E.True)
E → false	E.code := GeraCode('goto' E.False)

Figura 87: DDS para avaliação e expressões lógicas por fluxo de controle.

Considere de novo a expressão:

```
a < b or c < d and e < f
```

Suponha que as saídas verdadeira e falsa para toda a expressão tenham sido estabelecidas em RTrue e RFalse. Então, usando a DDS da Figura 87 obteríamos o seguinte código:

```
if a < b goto RTrue
    goto R1
R1: if c < d goto R2
    goto RFalse
R2: if e < f goto RTrue
    goto RFalse
```

Considere agora o enunciado:

```
while a < b do
    if c < d then
        x := y + z
    else
        x := y - z
```

Considerando a DDS para tradução dos enunciados while-do e if-then-else juntamente com a DDS para tradução de expressões lógicas, o tradução do

enunciado acima seria:

```
R1: if a < b goto R2
    goto RProximo
R2: if c < d goto R3
    goto R4
R3: $1 := y + z
    x := $1
    goto R1
R4: $2 := y - z
    x := $2
    goto R1
RProximo:
```

### 6.4.3 Backpatching

## 7 Referências Bibliográficas

AHO, Alfred; SETHI, Ravi; ULLMAN, Jeffrey D. Compiladores: princípios, técnicas e ferramentas. LTC Editora, 1995.

PARR, Terence. The Definitive ANTLR 4 Reference. The Pragmatic Bookshelf, 2012.